Planning with Time Limits in BDI Agent Programming Languages

Lavindra de Silva
RMIT University,
Melbourne, Australia
ldesilva@cs.rmit.edu.au

Anthony Dekker
DSTO[*]*,*
Canberra, Australia
dekker@ACM.org

James Harland
RMIT University,
Melbourne, Australia
jah@cs.rmit.edu.au

Abstract

This paper provides a theoretical basis for performing time limited planning within Belief-Desire-Intention (BDI) agents. The BDI agent architecture is recognised as one of the most popular architectures for developing agents for complex and dynamic environments, in addition to which they have a strong theoretical foundation. Recent work has extended a BDI agent specification language to include HTN-style planning as a built-in feature. However, the extended semantics assume that agents have an unlimited amount of time available to perform planning, which is often not the case in many dynamic real world environments. We extend previous research by using ideas from anytime algorithms, and allow programmer control over the amount of time the agent spends on planning. We show that the resulting integrated agent specification language has advantages over regular BDI agent reasoning.

Keywords: BDI Agents, Operational Semantics, Time Limited Planning

1 Introduction

An Intelligent Agent is piece of software that is (Jennings, Sycara & Wooldridge 1998): *situated* in an environment, capable of *autonomous* behaviour, i.e. with little or no intervention from humans, and *flexible*. Flexibility means the agent is: responsive to the environment (i.e. able to perceive and respond to changes in a timely manner), proactive (i.e. able to exhibit goal directed behaviour), and social (i.e. able to interact with other agents and humans). Intelligent agents are used in numerous real world applications such as Unmanned Autonomous Vehicles (UAV) (Karim & Heinze 2005) and autonomous spacecraft control (Chien, Knight, Stechert, Sherwood & Rabideau n.d.).

The BDI (Belief-Desire-Intention) architecture, based on Bratman's theory of practical reasoning (Bratman 1987) and Dennett's theory of intentional systems (Dennett 1987), is a popular and well studied model for intelligent agents situated in complex and dynamic environments. *Beliefs* represent an agent's knowledge of its environment, *Desires* represent an agent's goals or its desired outcomes, and an *Intention* is a chosen desire that the agent commits to. The purpose of the BDI architecture is to make agents capable of behaving in a more human-like manner. There are numerous BDI agent programming and specification languages such as PRS (Ingrand, Georgeff & Rao 1992), AGENTSPEAK (Rao 1996), 3APL (Hindriks, de Boer, van der Hoek & Meyer 1999), JACK (Busetta, Rönnquist, Hodgson & Lucas 1999) and CAN (Winikoff, Padgham, Harland & Thangarajah 2002).

BDI systems execute as they reason and are robust in handling failure, therefore well suited for complex and dynamic environments requiring real-time reasoning capabilities. However, reasoning for longer durations before acting is also advantageous in certain situations where time is less critical, and with the recent advances in planning technology, *planning* or *lookahead* within the BDI architecture has become feasible.

In (Sardina, de Silva & Padgham 2006), Hierarchical Task Network (HTN) style planning (Erol, Hendler & Nau 1994) capabilities were integrated into the CAN BDI specification language introduced in (Winikoff et al. 2002). The integration involved (1) extending the CAN operational semantics with planning capabilities to form CANPLAN; (2) substantially improving and simplifying the CAN operational semantics; and (3) exploring the theoretical properties of the new CANPLAN language.

CANPLAN is a good first step towards formalising the type of planning that is appropriate for the BDI architecture. However, a notable limitation in the semantics is in not having control over the amount of time to be spent on planning – CANPLAN will attempt to find a complete solution for a goal, before executing the first action. Control over the planning duration is necessary in many domains, as it is often the case that agents only have (or should only spend) a limited amount of time for (on) planning.

Planning with time limits is important when solving real-time problems in a moderately fast-changing world (Korf 1990, Dean, Kaelbling, Kirman & Nicholson 1993*b*). In such situations, performing a relatively poor action in time may be preferable to performing a planned action which is too late. Planning with time limits can be achieved by giving the planner an explicit time limit, and falling back on BDI-style reaction if necessary.

Planning with time limits is also useful in agent-based modelling and simulation of real human organisations. Under extreme time pressure, human beings react by curtailing planning which cannot provide a solution in time (Newell & Simon 1972, Gigerenzer & Selten 2002). The CAVALIER-NPA framework (Dekker & de Silva 2006) investigates the impact of planning time, communications delays, and workload sharing in military organisations through modelling with BDI agents augmented by time-limited planning.

Therefore, it is important that work with time-limited planning of this kind be placed on a solid

[*]Defense Science and Technology Organisation

theoretical basis. In this paper, we extend the CanPlan operational semantics with the ability to plan for a controlled depth, and we explore the theoretical properties of the new operational semantics. For example, we show that limited planning will lead to early failure detection, not possible with regular BDI execution. Our work also forms a theoretical basis for the experimental work done with CAVALIER-NPA (Dekker & de Silva 2006), and for work in augmenting agents with a time-limited version of the Metric-FF (Hoffmann 2003) planner.

2 Background

In this section we will introduce the work done on formalising the BDI architecture, i.e. the Can (Conceptual Agent Notation) language (Winikoff et al. 2002), and we will also introduce CanPlan (Sardina et al. 2006), which is an extension of Can to capture planning from within the BDI architecture. Finally we will discuss past work on planning under time constraints.

2.1 The CAN Language

Can (Conceptual Agent Notation) (Winikoff et al. 2002) is a high level plan language most similar to AgentSpeak (Rao 1996), but also similar to other languages such as 3APL (Hindriks et al. 1999, van Riemsdijk, Dastani & Meyer 2005). Unlike other languages, Can combines a declarative and procedural view of *goals* and includes semantics for BDI failure recovery and goal persistence (Sardina et al. 2006).

According to Can, BDI agents consist of a set of beliefs $\mathcal{B}$ and a plan library Π. There are three operations possible on $\mathcal{B}$; (1) check whether a condition ϕ follows from $\mathcal{B}$ ($\mathcal{B} \models \phi$), (2) add beliefs to $\mathcal{B}$ ($\mathcal{B} \cup \{b\}$), and; (3) delete beliefs from $\mathcal{B}$ ($\mathcal{B} \setminus \{b\}$).

The plan library Π consists of *predefined* plan rules provided by the programmer, and is therefore different to the plans (or solutions) generated at runtime by a planner. More specifically, Π consists of a set of rules of the form $e : \psi \leftarrow P$, where e represents an event handled by the system, and P represents a corresponding plan body that could be used to achieve e. There can be one or more plan bodies that handle the same event, but a single plan body can only handle a single event. The condition under which a certain P is applicable for e is encoded in ψ. On receiving an event e' from the environment, Can rules search for a P' that handles e', but at the same time, a P' whose precondition ψ' is met. P' is then executed. If there is no applicable P', the handling of e' fails.

The execution of a plan body P involves executing the contents of P, which may include primitive actions *act*, operations to add $+b$ and delete $-b$ beliefs, tests for conditions $?\phi$, etc. P can include multiple such programs with the sequencing construct ; (i.e. $P_1; P_2$) or execute programs in parallel using the paralellism construct $||$ (i.e. $P_1||P_2$)[1]. The full language of P is shown below[2].

$$P ::= nil \mid act \mid ?\phi \mid +b \mid -b \mid !e \mid P_1; P_2 \mid P_1 \rhd P_2 \mid P_1 \| P_2 \mid \mathsf{Goal}(\phi_s, P_1, \phi_f) \mid (\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!).$$

The operational semantics of Can are as follows. A *transition* $C \longrightarrow C'$ specifies that some *configuration* C yields some configuration C' in a single execution step. Similarly, $C \longrightarrow$ specifies that there is some configuration C' that can be reached within a single execution step from C. The relation $\xrightarrow{*}$ denotes the reflexive transitive closure of $\longrightarrow$.

The *transition relation* $\longrightarrow$ on a configuration is defined using one or more derivation rules. Derivation rules have an antecedent and consequent. The antecedent can be empty, but otherwise consists of transitions and auxiliary conditions. The conclusion consists of a single transition. See Plotkin's (Plotkin 1981) Structural Operational Semantics for a full account of the operational semantics in Can.

Some of the main derivation rules of Can are shown and explained below. The usage of some of these rules will be shown in the examples to follow.

$$\frac{\Delta = \{\psi_i\theta : P_i\theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{\langle \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle} \; Event$$

$$\frac{\psi_i : P_i \in \Delta \quad \mathcal{B} \models \psi_i\theta}{\langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P_i\theta \rhd (\!|\Delta \setminus P_i|\!)\rangle} \; Sel$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1 \rhd P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P' \rhd P_2)\rangle} \; \rhd$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, (nil \rhd P)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle} \; \rhd_t$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \ \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1; P_2)\rangle \longrightarrow \langle \mathcal{B}', \ \mathcal{A}', (P'; P_2)\rangle} \; Seq$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, (nil; P)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P\rangle} \; Seq_t$$

$$\frac{\mathcal{B} \models \phi\theta}{\langle \mathcal{B}, \mathcal{A}, ?\phi\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle} \; ?$$

$$\frac{a : \psi \leftarrow \Phi^+; \Phi^- \in \Lambda \quad a\theta = act \quad \mathcal{B} \models \psi\theta}{\langle \mathcal{B}, \mathcal{A}, act\rangle \longrightarrow \langle (\mathcal{B} \setminus \Phi^-\theta) \cup \Phi^+\theta, \mathcal{A} \cdot act, nil\rangle} \; act$$

The *Event* rule collects all the relevant plan bodies for the event, along with their corresponding preconditions, and stores them in $(\!|\Delta|\!)$. The *Sel* rule selects a plan body from $(\!|\Delta|\!)$ whose precondition meets the current state of the world, and schedules the plan body for execution. The $\rhd$ rule states that, as long as there is more to decompose in P_1, P_1 should be decomposed. On reaching the program $nil \rhd P$, rule $\rhd_t$ states that there is no need for program P, as the left hand side of $\rhd$ has been successfully decomposed. The rule *Seq* handles the execution of two programs in a sequence, and on reaching the sequence $nil; P$, rule Seq_t removes the program *nil* and continues with the right hand side of the sequence. The rule that tests for a condition ? simply succeeds if the condition ϕ is met in the current state, and the *act* rule applies an action to the current state of the world, on the basis that ψ is met in the current state, by adding and/or deleting facts from the current state of the world.

2.1.1 An Example

As an example consider a simple meeting scheduler (personal assistant) agent written in Can. The agent's task is to handle the scheduling of meetings for a single person. When meeting requests arrive, the agent will try and schedule the request into an appropriate slot in the person's diary.

The set of possible plan rules for such an agent is shown in Figure 1. The events that the agent can handle are $scheduleMeeting$, $findSuitableSlot$, $clearSlot$ and $moveEntry$, where $scheduleMeeting$ is an external event received from the environment and the rest are internal events. Some events, such as $findSuitableSlot$, have more than one plan rule. Some of the plan bodies are simple and always succeed, provided the precondition holds, and some other plan bodies are complex and can lead to a sequence of further operations, including posting of internal events and modifications to the agent's beliefs.

[1] Refer to (Sardina et al. 2006) for a description of the complete language of P.

[2] Program $P_1 \rhd P_2$ executes P_1 and then executes P_2 only if P_1 failed. The guarded set of plan bodies $(\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!)$ is also represented as $(\!|\Delta|\!)$ for short.

According to the plan rules in Figure 1, when the agent receives a *scheduleMeeting* event requesting the scheduling of some person p into the diary, the internal events $findSuitableSlot(p, s)$, $clearSlot(s)$ and $addEntryToEmptySlot(p, s)$ are posted in that sequence. The first internal event finds an appropriate slot for the person p, and on successful completion, a binding for slot s will have been found. $clearSlot(s)$ then clears slot s, which may include doing nothing in the situation where slot s is already clear. The final event $addEntryToEmptySlot(p, s)$ will add the person p to slot s.

The following example shows briefly how the scheduling of a meeting works in CAN. A different, more detailed example can be found in (Winikoff et al. 2002).

Assume the initial state is such that *John* is already scheduled into the diary, and *David* wants to be scheduled into the diary, i.e.:

$$\begin{aligned}\mathcal{B}_0 \models (&occupiedBy(John, Monday9am) \wedge \\ &availableSlotFor(John, Monday9am) \wedge \\ &availableSlotFor(John, Tuesday9am) \wedge \\ &meetingRequest(David) \wedge \\ &availableSlotFor(David, Monday9am))\end{aligned}$$

The agent execution will then proceed as follows[3], where the rules used are *Event*, *Sel*, *Event*, *Sel*, $\rhd_t$, and Seq_t respectively, from top to bottom.

$$\begin{aligned}\langle \mathcal{B}_0, \mathcal{A}_0, !sm\rangle &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, (\!|!fss; !cs; !as|\!)\rangle \\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, !fss; !cs; !as \rhd (\!|\,|\!)\rangle \\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, (\!|nil|\!); !cs; !as \rhd (\!|\,|\!)\rangle \\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, nil \rhd (\!|\,|\!); !cs; !as \rhd (\!|\,|\!)\rangle \\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, nil; !cs; !as \rhd (\!|\,|\!)\rangle \\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, !cs; !as \rhd (\!|\,|\!)\rangle \\ &\longrightarrow ...\end{aligned}$$

The *Event* derivation rule is applied first as a result of its antecedent being met. The consequent of the rule selects the set of applicable $\psi : P$ pairs for $!sm$, to create $(\!|\Delta|\!)$. In this example, $(\!|\Delta|\!)$ only contains a single plan body, i.e. $!fss; !cs; !as$. The only rule that can work with $(\!|\Delta|\!)$, is *Sel*, which selects $!fss; !cs; !as$ from $(\!|\Delta|\!)$, and removes the plan body from $(\!|\Delta|\!)$. The first program in $!fss; !cs; !as$, i.e. $!fss$, results in the *Event* and *Sel* rules being used again, but this time for selecting the applicable set, and resulting plan body (respectively), for $!fss$. Finally, Seq_t removes *nil* from the program $nil; !cs; !as$ to yield $!cs; !as$, and execution continues in this manner until $!sm$ evaluates to *nil*.

2.2 HTN Planning

We can gain some guidance on formalising planning for BDI agents by looking at the semantics of HTN planning. HTN planning has several similarities to BDI execution. Although BDI systems execute as they look for a solution, and HTN planners find a complete solution before execution begins, they use similar techniques for composing a solution(de Silva & Padgham 2004).

The rest of this subsection is based on (Sardina et al. 2006). An HTN *planning domain* $\mathcal{D} = (\Pi, \Lambda)$ consists of a library Π of methods and a library Λ of primitive tasks. Each primitive task in Λ is a STRIPS (Fikes, Hart & Nilsson 1972) style action with corresponding preconditions and effects in the form of *add* and *delete* lists. An HTN *planning problem* $\mathbf{P}$ is the triple $\langle d, \mathcal{B}, \mathcal{D}\rangle$ where d is the network of tasks to accomplish, $\mathcal{B}$ is the initial belief state (i.e. a set of all ground atoms that are true in $\mathcal{B}$), and $\mathcal{D}$ is a planning domain. A *plan* σ is a sequence $act_1 \cdot ... \cdot act_n$ of ground actions (that is, ground primitive tasks).

Given a specific planning problem $\mathbf{P}$, HTN planning proceeds by selecting an applicable reduction method from $\mathcal{D}$, and applying the method to some compound task in d. The resulting task network d' is then reduced further, until no compound tasks remain. On every application of a reduction method, compound tasks become less abstract (i.e. more primitive), eventually producing a solution σ, composed only of primitive tasks. If an applicable reduction method does not exist for a compound task at any stage, the planner backtracks and tries an alternative reduction for a different compound task.

The following operational semantics for HTN planning is taken from (Erol et al. 1994). The semantics define the set of plans $sol(d, \mathcal{B}, \mathcal{D})$ that solves a planning problem instance $\mathbf{P} = \langle d, \mathcal{B}, \mathcal{D}\rangle$.

$$\begin{aligned}sol_1(d, \mathcal{B}, \mathcal{D}) &= comp(d, \mathcal{B}, \mathcal{D}), \\ sol_{n+1}(d, \mathcal{B}, \mathcal{D}) &= sol_n(d, \mathcal{B}, \mathcal{D}) \cup \bigcup_{d' \in red(d, \mathcal{B}, \mathcal{D})} sol_n(d', \mathcal{B}, \mathcal{D}), \\ sol(d, \mathcal{B}, \mathcal{D}) &= \bigcup_{n<\omega} sol_n(d, \mathcal{B}, \mathcal{D}),\end{aligned}$$

In the above semantics, $comp(d, \mathcal{B}, \mathcal{D})$ is the set of primitive tasks, corresponding to all plan competitions of d, and $red(d, \mathcal{B}, \mathcal{D})$ is the set of all reductions of d in $\mathcal{B}$ by methods in $\mathcal{D}$.

2.3 The CANPlan Language

In (Sardina et al. 2006), the CANPLAN language was introduced. CANPLAN is an extension to CAN with support for; (STRIPS like) actions with preconditions and effects, multiple variable bindings, a more simplified account of declarative goals, and planning capabilities.

The main feature of CANPLAN is the (local) planning construct $\mathsf{Plan}(P)$. $\mathsf{Plan}(P)$ is a *planning-program* that ensures P can be fully decomposed, before execution can begin within the BDI execution engine. The first step in the plan resulting from the decomposition is then executed within the BDI execution engine. Hence $\mathsf{Plan}(P)$ encodes finding the right choices at decision points within P, so that the BDI engine may make the right choices when faced with the decision points during execution.

The main derivation rule for Plan is shown below. The rule relies on two types of transitions on configurations: bdi and plan transitions. The relation $C \xrightarrow{\mathsf{bdi}} C'$ specifies a single step transition of type bdi, corresponding to a single execution step of an agent. Similarly $C \xrightarrow{\mathsf{plan}} C'$ specifies a single step transition of type plan, corresponding to a single step of imagined execution within the planner. A transition without a label: $C \longrightarrow C'$, specifies that either type applies.

The Plan derivation rule states the conditions under which a planning-program $\mathsf{Plan}(P)$ in the context of the current belief base $\mathcal{B}$ can *legally* make a single-step transition, with $\mathsf{Plan}(P')$ being the remaining program to be executed and $\mathcal{B}'$ being the new belief base ($\mathcal{A}$ and $\mathcal{A}'$ stand for the actions executed so far):

$$\frac{\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P'\rangle \quad \langle \mathcal{B}', \mathcal{A}', P'\rangle \xrightarrow{\mathsf{plan}^*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle}\ \mathsf{Plan}$$

[3] where sm = scheduleMeeting(David), fss = findSuitableSlot(David,s), cs = clearSlot(s) and as = addEntryToEmptySlot(David,s). Note that the preconditions have been excluded for clarity, when showing the contents of $(\!|\Delta|\!)$.

$scheduleMeeting(p) : meetingRequest(p) \leftarrow !findSuitableSlot(p,s)\ ;\ !clearSlot(s)\ ;\ !addEntryToEmptySlot(p,s)$
$findSuitableSlot(p,s) :\ availableSlotFor(s,p) \leftarrow nil$
$clearSlot(s) :\ \nexists(p) occupiedBy(s,p) \leftarrow nil$
$clearSlot(s) :\ occupiedBy(s,p) \wedge availableSlotFor(s2,p) \wedge\ \nexists(p2) occupiedBy(s2,p2) \leftarrow !moveEntry(p,s2)$
$clearSlot(s) :\ occupiedBy(s,p) \wedge availableSlotFor(s2,p) \wedge occupiedBy(s2,p2) \wedge s \neq s2 \leftarrow !clearSlot(s2)\ ;\ !moveEntry(p,s2)$
$moveEntry(p,s) : occupiedBy(s2,p) \wedge availableSlotFor(s,p) \wedge\ \nexists(p2) occupiedBy(s,p2) \leftarrow -occupiedBy(s2,p)\ ; +occupiedBy(s,p)$
$addEntryToEmptySlot(p,s) : \neg occupiedBy(s,p) \leftarrow +occupiedBy(s,p)$

Figure 1: The set of plan rules in Π for a meeting scheduler

The rule states that a configuration $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle$ can make a single BDI step to $\langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle$ provided that $\langle \mathcal{B}, \mathcal{A}, P\rangle$ can make a single plan step to $\langle \mathcal{B}', \mathcal{A}', P'\rangle$, from where it is possible to reach a *final* configuration $\langle \mathcal{B}'', \mathcal{A}'', nil\rangle$ in a finite number of planning steps (Sardina et al. 2006). Note that, to make a single BDI transition, the only requirement is for at least one plan to exist. The rules *permit*, but do not *require* the agent to commit to a plan when one is found: interrrupting a plan and re-planning is always permissible.

It was proven in (Sardina et al. 2006) that $\mathsf{Plan}(P)$ can be seen as an integrated HTN planner, operating on the *same domain knowledge* as the BDI system. However, $\mathsf{Plan}(P)$ does not merely lookahead on the BDI execution cycle, as some types of execution, such as failure handling, is only used at the BDI level.

Since HTN planning is more expressive than first-principles planning (Erol et al. 1994), CanPlan can also be used to encode a first principles planner (Fikes et al. 1972), by writing a program of the form $\mathsf{Plan}\ (!seqActions; ?\phi)$ (Sardina et al. 2006). The event *seqActions* can be handled by multiple programs, each (except for a terminating program) containing a single action followed by a recursive call to *seqActions*, to simulate a first principles planner trying different combinations of existing actions until a terminating condition is reached. Additionally, CanPlan can be used to encode more general planning strategies, e.g. $\mathsf{Plan}(P_a) \rhd \mathsf{Plan}\ (!seqActions; ?\phi)$, which specifies that first-principles planning should be tried, only after failure of the HTN planning represented by $\mathsf{Plan}(P_a)$.

2.3.1 An Example

In the previous example, scheduling a new meeting into the diary may sometimes involve rescheduling existing meetings. The rescheduling is handled by the plan body of $clearSlot(s)$ which has a recursive call to $clearSlot(s)$. However, an alternative and possibly better approach is to remove the plan rule with the recursive call, and to replace the call to $clearSlot(s)$ occuring in the plan body for $scheduleMeeting(p)$, with $\mathsf{Plan}(!clearSlot(s))$. The behavior of the BDI system on reaching the $\mathsf{Plan}(!clearSlot(s))$ program is shown below [4].

The rule Plan is the first applicable rule. The first step on reaching $\mathsf{Plan}(!clearSlot(Monday9am))$ is shown below (note that programs P and P' from the original Plan rule have been replaced with the corresponding programs of this example).

$$\frac{\langle C, !cs\rangle \xrightarrow{\mathsf{plan}} \langle C, (\!|!me|\!)\rangle \quad \langle C, (\!|!me|\!)\rangle \xrightarrow{\mathsf{plan}^*} \langle \mathcal{B}', \mathcal{A}', nil\rangle}{\langle C, \mathsf{Plan}(!cs)\rangle \longrightarrow \langle C, \mathsf{Plan}((\!|!me|\!))\rangle}\ \mathsf{Plan}$$

The antecedent of the above rule is met as a result of applying *Event*, *Sel*, *Event*, *Sel* and Seq_t (etc.) rules, respectively. The example below shows these initial steps.

$$\begin{aligned}\langle \mathcal{B}_0, \mathcal{A}_0, \mathsf{Plan}(!cs)\rangle &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, \mathsf{Plan}((\!|!me|\!))\rangle\\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, \mathsf{Plan}(!me \rhd (\!|\,|\!))\rangle\\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, \mathsf{Plan}((\!|-ob\ ; +ob|\!) \rhd (\!|\,|\!))\rangle\\ &\longrightarrow \langle \mathcal{B}_0, \mathcal{A}_0, \mathsf{Plan}(-ob\ ; +ob \rhd (\!|\,|\!) \rhd (\!|\,|\!))\rangle\\ &\longrightarrow \langle \mathcal{B}_1, \mathcal{A}_0, \mathsf{Plan}(+ob \rhd (\!|\,|\!) \rhd (\!|\,|\!))\rangle\\ &\longrightarrow ...\end{aligned}$$

Note that the last step results in the belief base changing from $\mathcal{B}_0$ to $\mathcal{B}_1$. Execution is continued in this manner, until the Plan rule succeeds, resulting in the slot $Monday9am$ being cleared.

2.4 Planning with Time Limits

The $\mathsf{Plan}(P)$ construct is an important addition to the Can language, as it allows lookahead deliberation before acting. The construct ensures that the agent finds a complete solution for P, before executing even the first step of P. However, it is not always feasible to find a complete solution – if the environment is rapidly changing, the agent will need to act with little or no deliberation. Alternatively, the agent may have enough time for deliberation, but may perform better if the time spent on deliberation was balanced carefully with execution (Dekker & de Silva 2006).

Limited deliberation has been studied for many years in the field of anytime algorithms (Korf 1990, Russell & Wefald 1991, Dean & Boddy 1988, Briggs & Cook 1999, Goodwin 1994, Hansen & Zilberstein 2001, Dean, Kaelbling, Kirman & Nicholson 1993*a*, Drummond, Swanson, Bresina & Levinson 1993, Dean et al. 1993*b*, Miura & Shirai 2001, Atkins, Durfee & Shin 1996) but without a formal operational semantics. Anytime algorithms provide a tradeoff between time spent on deliberation and the quality of the solution, by ensuring that the quality of the solution gradually increases as more time is spent on deliberation.

There are two broad categories of anytime algorithms (Zilberstein 1995): *contract algorithms* and *interruptible algorithms*.

2.4.1 Contract Algorithms

Contract algorithms (e.g. (Korf 1990)) require a time limit to be known in advance, before the algorithm can begin. Moreover, the algorithm cannot be interrupted for the specified duration. The time limit can either be programmer specified, or calculated based on the state of the world (Goodwin 1994, Atkins et al. 1996).

Many existing planning algorithms can be considered contract algorithms, and depth-limited planning is one of them (Zilberstein & Russell 1995, Russell & Zilberstein 1991). Hence, we have essentially introduced a contract algorithm into the BDI architecture. This mapping of our work to a contract algorithm is important, as future work can focus on using existing techniques for converting contract algorithms into interruptible ones, to extend our operational semantics.

[4] where -ob = -occupiedBy(John,Monday9am), +ob = +occupiedBy(John,Tuesday9am), me = moveEntry(John,-Tuesday9am), cs = clearSlot(Monday9am), and C = $\mathcal{B}_0, \mathcal{A}_0$.

2.4.2 Interruptible Algorithms

Unlike contract algorithms, interruptible algorithms do not require a predefined time limit. The algorithm can be interrupted at any time during the deliberation process, and a solution obtained. Due to the ability to interrupt the algorithm, interruptible algorithms are more difficult to construct than contract algorithms.

Interruptible algorithms are suitable for domains where the amount of time available for deliberation is not known in advance. The agent will therefore deliberate until an event is received from the environment, signalling the need to stop deliberation and start execution.

2.4.3 Converting a Contract Algorithm into an Interruptible Algorithm

Interestingly, given any contract algorithm, a corresponding interruptible algorithm can be composed with a small penalty (Zilberstein 1995). The conversion is done by initially allocating some time period t to the contract algorithm, and repeatedly calling the contract algorithm with an increased time allocation on each subsequent call. According to (Russell & Zilberstein 1991), the optimal time allocation for each subsequent call is an exponential increase, in the sequence: t, $2t$, $4t$, ... $2^i t$, where t is a value arbitrarily chosen, representing the smallest time allocation which produces a significant improvement in the quality of the solution (Russell & Zilberstein 1991).

When the newly composed interruptible algorithm is interrupted at some point in time, the result obtained from the most recently completed contract algorithm is returned. For example, if the interruptible algorithm were interrupted at some time t_1, where $2^k t < t_1 < 2^{k+1}t$, the solution returned will be the one produced by the contract algorithm with time allocation $2^k t$. If the algorithm were interrupted at some time t_2, where $2^{k+2}t < t_2 < 2^{k+3}t$, the solution returned would be from the contract algorithm with time allocation $2^{k+2}t$. Hence the solution at interruption point t_2 is guaranteed to be no worse than the solution at interruption point t_1.

3 Planning with Time Limits in BDI Systems

We now extend CanPlan with a new construct $\mathsf{Plan}(P, K)$, where K is a number specifying the maximum depth for lookahead within P. More specifically, $\mathsf{Plan}(P, K)$ means "*plan for P offline, searching for a complete hierarchical decomposition within depth K.*"

The new construct $\mathsf{Plan}(P, K)$ is intended to be used in a similar manner to $\mathsf{Plan}(P)$, i.e. from anywhere within a CanPlan program. Hence we add $\mathsf{Plan}(P, K)$ into the language of P in CanPlan, to give the following new language for P:

$$P ::= nil \mid act \mid ?\phi \mid +b \mid -b \mid !e \mid P_1; P_2 \mid P_1 \rhd P_2 \mid P_1 \| P_2 \mid \mathsf{Plan}(P) \mid \mathsf{Plan}(P, K) \mid \mathsf{Goal}(\phi_s, P_1, \phi_f) \mid (\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!).$$

Next we specify the necessary behaviour when the new construct $\mathsf{Plan}(P, K)$ is encountered during execution. The specification is done by introducing derivation rules for $\mathsf{Plan}(P, K)$. We provide four simplified derivation rules that are sufficient for dealing with $\mathsf{Plan}(P, K)$: two main rules PlanK_{Suc} and PlanK_{Opt}, and two simpler rules PlanK_t and $\mathsf{PlanK}_{\,\mathsf{P}}$.

The first main rule PlanK_{Succ} handles the case where there is a solution for P of length less than or equal to K, and the second main rule PlanK_{Opt} handles the case where P does not have a solution of length less than or equal to K. The simpler rule PlanK_t handles the case where planning has been completed, and $\mathsf{PlanK}_{\,\mathsf{P}}$ handles the case where a $\mathsf{Plan}(P, K)$ construct is nested within another $\mathsf{Plan}(P, K)$ construct. All new rules are shown in Figure 2, and a more detailed description of each follows.

The PlanK_{Succ} rule states that configuration $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle$ can evolve to $\langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P', K)\rangle$ provided that $\langle \mathcal{B}, \mathcal{A}, P\rangle$ can evolve to $\langle \mathcal{B}', \mathcal{A}', P'\rangle$, from where it is possible to reach a final (successful) configuration within a depth of K. Therefore the consequent of this rule is only applicable in situations where P can be decomposed completely, in K steps or less.

The alternatives when there is no solution within a depth of K is to either behave optimistically or to behave pessimistically. If $\mathsf{Plan}(P, K)$ behaves pessimistically, and fails when no solution can be found within depth K, regular agent execution, i.e. without the use of planning, may sometimes find a successful solution when $\mathsf{Plan}(P, K)$ does not. Hence by making $\mathsf{Plan}(P, K)$ behave optimistically, there is a guarantee that the agent will behave no worse than the regular execution of P (shown later).

The optimistic rule PlanK_{Opt} states that configuration $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle$ can evolve to $\langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P', K)\rangle$ provided that $\langle \mathcal{B}, \mathcal{A}, P\rangle$ can evolve to $\langle \mathcal{B}', \mathcal{A}', P'\rangle$, from where it is possible to reach some configuration in $K - 1$ steps. Hence PlanK_{Opt} behaves optimistically by executing a step, hoping that some step after K will lead to a final configuration. Note that the first (left) conjunction in the antecedent of PlanK_{Opt} ensures that this rule is not applicable when PlanK_{Succ} is applicable.

The simpler rule PlanK_t handles planning for program *nil*, i.e. when P has successfully been executed within $\mathsf{Plan}(P, K)$. The rule simply returns success from $\mathsf{Plan}(P, K)$, by replacing $\mathsf{Plan}(nil, K)$ with *nil*. The second simpler rule $\mathsf{PlanK}_{\,\mathsf{P}}$ handles the $\mathsf{Plan}(P, K)$ construct from within a planning context. If a $\mathsf{Plan}(P, K)$ construct is nested within another $\mathsf{Plan}(P, K)$ construct, lookahead and depth limit within the nested construct is avoided, and control on lookahead and depth-limit will be maintained by the meta-level construct. These two simpler rules are similar to the secondary rules Plan_t and $\mathsf{Plan}_{\,\mathsf{P}}$, defined in (Sardina et al. 2006) for the $\mathsf{Plan}(P)$ construct[5].

We now use the derivation rules to show a number of interesting properties about the behaviour of $\mathsf{Plan}(P, K)$.

4 Theoretical Properties

In this section we will discuss the formal properties of the $\mathsf{Plan}(P, K)$ construct. First we show some rules related to *agent executions*, along with four useful definitions, both taken from (Sardina et al. 2006).

An agent is a tuple of the form $\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma\rangle$ where $\mathcal{N}$ is the name of the agent, Λ is an action library, Π is a library of plan rules, $\mathcal{B}$ is a belief base, $\mathcal{A}$ is a sequence corresponding to actions already executed, and Γ is a set of plan bodies currently being executed. Agent transitions $\Longrightarrow$ are defined by the following three rules.

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma\rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \cup \{P'\}\rangle}\ A_{step}$$

$$\frac{e \text{ is a new external event}}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma\rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{!e\}\rangle}\ A_{event}$$

[5] Two more rules are necessary, but are trivial and therefore left out. These rules should be similar to $\mathsf{PlanK}_{\,\mathsf{P}}$, and used for dealing with situations where $\mathsf{Plan}(P)$ is encountered within $\mathsf{Plan}(P, K)$, and vice versa. Again, control will be left to the construct encountered first.

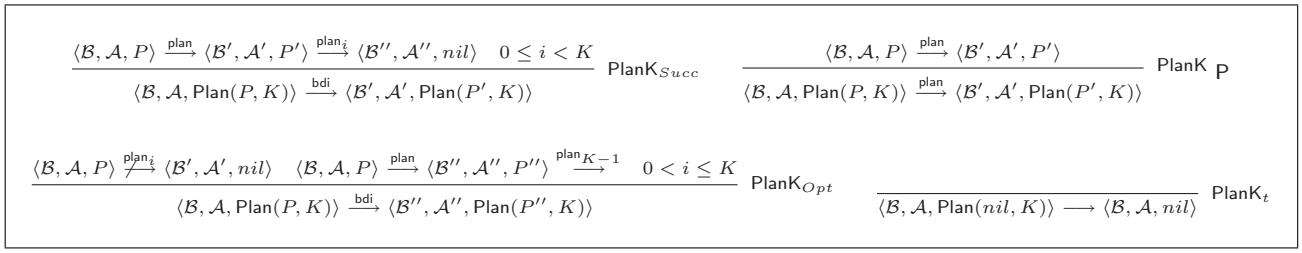

Figure 2: The four new rules for planning with time constraints

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P\rangle \not\longrightarrow}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma\rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\}\rangle}\ A_{clean}$$

Rule A_{step} performs a single step in a plan body within Γ, rule A_{event} appends a new external event e to Γ, and rule A_{clean} removes a plan body that is completed (i.e. one which has reached *nil*, or one from where no transition is possible) from Γ. Next, some important definitions used in the rest of the paper are shown (see (Sardina et al. 2006) for more details).

Definition 1 (BDI Execution) *A BDI execution of an agent $C_0 = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_0, \mathcal{A}_0, \Gamma_0\rangle$ is a, possibly infinite, sequence of agent configurations $C_0 \cdot C_1 \cdot \ldots \cdot C_n \cdot \ldots$ such that $C_i \Longrightarrow C_{i+1}$, for every $i \ge 0$. A terminating execution is a finite execution $C_0 \cdot \ldots \cdot C_n$ where $C_n = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_n, \mathcal{A}_n, \{\}\rangle$. An environment-free execution is one in which rule $\mathcal{A}_{event}$ has not been used.*

Definition 2 (Intention Execution) *Let E be a BDI execution $C_0 \cdot C_1 \cdot \ldots \cdot C_n$ for an agent $C_0 = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_0, \mathcal{A}_0, \Gamma_0\rangle$, where $\Gamma_0 = \Gamma_0' \cup \{P_0\}$. Intention P_0 in C_0 has been fully executed in E if $P_n = \epsilon$; otherwise P_0 is currently executing in E. In addition, intention P_0 in C_0 has been successfully executed in E if $P_i = nil$, for some $i \le n$; intention P_0 has failed in E if it has been fully but not successfully executed in E.*

Definition 3 *Two, possibly derived, agent executions $C_0 \cdot \ldots \cdot C_n$ and $C_0' \cdot \ldots \cdot C_n'$ are equivalent modulo intentions iff $C_i = \langle \mathcal{N}_i, \Lambda_i, \Pi_i, \mathcal{B}_i, \mathcal{A}_i, \Gamma_i'\rangle$, for every $0 \le i \le n$. Also, the two executions are equivalent modulo intentions $P_0 \in \Gamma_0$ and $P_0' \in \Gamma_0'$ if they are equivalent modulo intentions and for every $0 \le i \le n$, $(\Gamma_i' \setminus \{P_i'\}) = (\Gamma_i \setminus \{P_i\})$ (where P_i (P_i') is P_0's (P_0''s) evolution in configuration C_i (C_i')).*

Definition 4 (Program Simulation) *Let E be an execution of $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P\}\rangle$. Program P' simulates program P in execution E iff there is an execution E' of configuration $C' = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P'\}\rangle$ such that (a) $\overline{E}$ and $\overline{E'}$ are equivalent modulo P and P'; and (b) if P has been successfully executed in E, so has P' in E'. We say that P' simulates P iff P' simulates P in every execution of any configuration.*

Next we discuss the theorems. The first two theorems are based on the relationship between the $\mathsf{Plan}(P)$ and $\mathsf{Plan}(P, K)$ constructs. The first theorem shows the intuitive result that $\mathsf{Plan}(P)$ can, in any situation, find all the successful executions that $\mathsf{Plan}(P, K)$ can find. Moreover, in some situations, $\mathsf{Plan}(P)$ will find more successful executions than $\mathsf{Plan}(P, K)$.

Theorem 1 *For every program P and every value of K, $\mathsf{Plan}(P)$ simulates $\mathsf{Plan}(P, K)$.*

Proof. The proof relies on the following lemma from (Sardina et al. 2006): if $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$, then $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$.

Let $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P, K)\}\rangle$ and let $C' = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P)\}\rangle$. Contrary to the theorem, suppose there is some successful execution E for C of the form $C_0 = C \cdot \ldots \cdot C_k$ such that $C_k = \langle \mathcal{B}_k, \mathcal{A}_k, nil\rangle$ for some $k \le K$ and that an equivalent modulo $\mathsf{Plan}(P, K)$ and $\mathsf{Plan}(P)$ execution E' does not exist for C'. From the $\mathsf{Plan}(P, K)$ derivation rules PlanK_{Succ} and PlanK_t, E exists because $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}^{k-1}} \langle \mathcal{B}_j, \mathcal{A}_j, \mathsf{Plan}(nil, K)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}_k, \mathcal{A}_k, nil\rangle$ holds. Therefore the antecedent of PlanK_{Succ}; $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}^{k-1}} \langle \mathcal{B}_j, \mathcal{A}_j, nil\rangle$, also has to hold. From the above lemma, if $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}', \mathcal{A}', nil\rangle$, then $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}_*} \langle \mathcal{B}', \mathcal{A}', nil\rangle$. Since $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}^{k-1}} \langle \mathcal{B}_j, \mathcal{A}_j, nil\rangle$ holds, using the above lemma and derivation rule Plan_t, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}^{k-1}} \langle \mathcal{B}_j, \mathcal{A}_j, \mathsf{Plan}(nil)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}_k, \mathcal{A}_k, nil\rangle$ also holds. Therefore E' has to exist. ■

The second theorem says that, in situations where no solutions lie outside depth K, $\mathsf{Plan}(P, K)$ can also find all the successful executions that $\mathsf{Plan}(P)$ can find.

Theorem 2 *Let $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P)\}\rangle$. If for every belief base $\mathcal{B}'$, sequence of actions $\mathcal{A}'$, program P' and $i \ge K > 0$ such that $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', P'\rangle$, it is the case that $\langle \mathcal{B}', \mathcal{A}', P'\rangle \not\xrightarrow{\mathsf{plan}} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$, then for every value of K, $\mathsf{Plan}(P, K)$ simulates $\mathsf{Plan}(P)$ in all executions of C.*

Proof. On the contrary, suppose that there is some successful execution E of $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P)\}\rangle$, of the form $C_0 \cdot \ldots \cdot C_k$ where $C_k = \langle \mathcal{B}_k, \mathcal{A}_k, nil\rangle$, for which there is no equivalent modulo $\mathsf{Plan}(P)$ and $\mathsf{Plan}(P, K)$ execution E' of $C' = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P, K)\}\rangle$. Observe that for E to exist, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}_j} \langle \mathcal{B}_k, \mathcal{A}_k, nil\rangle$ holds, and from the antecedent of the Plan rule, $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}_j} \langle \mathcal{B}_k, \mathcal{A}_k, nil\rangle$ also holds. Furthermore, $0 \le j \le K$ holds as the theorem states that there is no successful execution past K. From Lemma 3, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_j} \langle \mathcal{B}_k, \mathcal{A}_k, nil\rangle$ holds and therefore the execution E' must exist. ■

In summary, the above two theorems show that $\mathsf{Plan}(P, K)$ is not as powerful as $\mathsf{Plan}(P)$ in general, but in certain specific situations, $\mathsf{Plan}(P, K)$ is as powerful as $\mathsf{Plan}(P)$. Next we move on to the relationship between $\mathsf{Plan}(P, K)$ and regular BDI execution, and show that the former does have advantages over

the latter. But first, we introduce a new lemma that the next two theorems rely on.

Lemma 3 *For every $\mathcal{B}, \mathcal{A}$ and P, if $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$, then $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_i} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$, for all $0 \leq i \leq K$.*

Proof. We prove this by induction on n, representing the number of derivation steps using transitions of type plan.

Base case $n = 0$: $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_0} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$ holds for $P = nil$, $\mathcal{B}_f = \mathcal{B}$, and $\mathcal{A}_f = \mathcal{A}$, and $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_0} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$ also holds using derivation rule PlanK_t.

Inductive case $n = m + 1$, for $0 < n \leq K$: Then, there exists $\langle \mathcal{B}^j, \mathcal{A}^j, P^j\rangle$ such that (a) $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}^j, \mathcal{A}^j, P^j\rangle$, and; (b) $\langle \mathcal{B}^j, \mathcal{A}^j, P^j\rangle \xrightarrow{\mathsf{plan}_m} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$. For (a) we can use derivation rule PlanK_{Succ} to obtain $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}^j, \mathcal{A}^j, \mathsf{Plan}(P^j, K)\rangle$. Moreover, using (b) and the induction hypothesis, $\langle \mathcal{B}^j, \mathcal{A}^j, \mathsf{Plan}(P^j, K)\rangle \xrightarrow{\mathsf{bdi}_m} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$. Thus $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_i} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$, for all $0 \leq i \leq K$ follows. ■

The next theorem states that, if failure within K steps is inevitable, $\mathsf{Plan}(P, K)$ will detect this failure, and fail (i.e. by not taking a step). This is an important result, as it shows that $\mathsf{Plan}(P, K)$ does indeed provide a benefit over regular BDI execution, as the latter will waste time executing up to the point at which it fails.

Theorem 4 *Let $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P, K)\}\rangle$ such that, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \not\xrightarrow{\mathsf{bdi}}$. Then for every belief base $\mathcal{B}'$, sequence of actions $\mathcal{A}'$, program P' and $0 \leq i < K$ such that $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', P'\rangle$, and for every belief base $\mathcal{B}''$, sequence of actions $\mathcal{A}''$ and program P'' such that $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_{K-1}} \langle \mathcal{B}'', \mathcal{A}'', P''\rangle$, it is the case that $\langle \mathcal{B}', \mathcal{A}', P'\rangle \not\xrightarrow{\mathsf{plan}} \langle \mathcal{B}''', \mathcal{A}''', nil\rangle$ and $\langle \mathcal{B}'', \mathcal{A}'', P''\rangle \not\xrightarrow{\mathsf{plan}}$.*

Proof. On the contrary, suppose;

1. $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \not\xrightarrow{\mathsf{bdi}}$ and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', nil\rangle$ for some $0 \leq i \leq K$. However, from Lemma 3, if $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', nil\rangle$ then it is the case that $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_i} \langle \mathcal{B}', \mathcal{A}', nil\rangle$, for some $0 \leq i \leq K$. Therefore $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \not\xrightarrow{\mathsf{bdi}}$ cannot hold.
2. $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \not\xrightarrow{\mathsf{bdi}}$ and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_K} \langle \mathcal{B}', \mathcal{A}', P'\rangle$. However, since $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_K}$, from derivation rule PlanK_{Opt}, it is the case that $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}}$. Therefore $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \not\xrightarrow{\mathsf{bdi}}$ cannot hold. ■

Theorem 5 and 6 shows the situations under which $\mathsf{Plan}(P, K)$ will take a single step, and what it means to take a single step.

As hinted by Theorem 2, in a situation where there is a solution within depth K, $\mathsf{Plan}(P, K)$ will take a single step only if that step is one that eventually leads to a solution. Therefore $\mathsf{Plan}(P, K)$ will guide BDI execution along a successful execution, when a successful execution is visible. This is again a useful result and contrasts with default BDI behaviour.

Theorem 5 *Let $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P, K)\}\rangle$ such that, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}}$ and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', nil\rangle$ for some $0 < i \leq K$. If E is an environment-free agent execution of C, then intention $\mathsf{Plan}(P, K)$ is either executing or has been successfully executed in E. Moreover, there is an execution E^s of C in which intention $\mathsf{Plan}(P, K)$ has been successfully executed in E^s.*

Proof. This follows directly from Lemma 3. ■

As mentioned before, $\mathsf{Plan}(P, K)$ behaves optimistically, and Theorem 6 confirms that this is indeed the case. If there is no solution within depth K, $\mathsf{Plan}(P, K)$ will take a single step, only if there is an execution of length K from P. Therefore $\mathsf{Plan}(P, K)$ will execute the first step of P, hoping that a successful path to a solution exists beyond K steps.

If these K steps are not the prefix of a successful execution, then $\mathsf{Plan}(P, K)$ will eventually fail. However, there is also a chance that these K steps lead to success. Most importantly, there is a higher chance that success will be reached with $\mathsf{Plan}(P, K)$, than with regular execution of P, as $\mathsf{Plan}(P, K)$ will prune all paths that fail within depth K.

Theorem 6 *Let $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\mathsf{Plan}(P, K)\}\rangle$ such that, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}}$ and such that for every belief base $\mathcal{B}'$, sequence of actions $\mathcal{A}'$, program P' and $0 \leq i < K$ such that $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', P'\rangle$, it is the case that $\langle \mathcal{B}', \mathcal{A}', P'\rangle \not\xrightarrow{\mathsf{plan}} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$. If E is an environment-free agent execution of C, then intention $\mathsf{Plan}(P, K)$ is either executing or has executed K steps in E.*

Proof. The proof relies on the following lemma: For every $\mathcal{B}, \mathcal{A}$ and P, if $\langle \mathcal{B}, \mathcal{A}, P\rangle \not\xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', nil\rangle$ and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_i} \langle \mathcal{B}', \mathcal{A}', P'\rangle$, then $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_i} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P', K)\rangle$, for some $0 \leq i \leq K$.

On the contrary suppose there is an environment-free execution E of the form $C_0 = C \cdot \ldots \cdot C_k$ such that $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k, K)\rangle \not\xrightarrow{\mathsf{bdi}}$ for some $k < K$. However, observe that $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_k} \langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k, K)\rangle$ will hold. From the antecedent of the derivation rule PlanK_{Opt}, we can see that $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_k} \langle \mathcal{B}_k, \mathcal{A}_k, P_k\rangle \xrightarrow{\mathsf{plan}_{K-k}} \langle \mathcal{B}', \mathcal{A}', P'\rangle$ and hence $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}_K} \langle \mathcal{B}', \mathcal{A}', P'\rangle$ applies. By using the above lemma, we get that $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}_K} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P', K)\rangle$. Next, since $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \not\xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P', K\rangle$, there exist $\mathcal{B}'', \mathcal{A}'', P''$ such that $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}'', \mathcal{A}'', P''\rangle \xrightarrow{\mathsf{bdi}_{K-1}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P', K)\rangle$. Thus, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P, K)\rangle \xrightarrow{\mathsf{bdi}}$ and the execution E cannot exist. ■

The last theorem conforms to the hypothesis that prefix plans can provide guidance to the reactive system at critical choice points, thus helping the reactive component avoid dead-ends during execution (Drummond et al. 1993).

5 Conclusion

Time limited planning is important in real world agent applications. On some occasions, agents do not have time to plan for a complete solution (Zilberstein, Charpillet & Chassaing 1999), while on other occasions, agents can benefit by being able to control the amount of time spent on planning (Dekker & de Silva 2006).

In this work we have extended previous work on planning in BDI systems, with ideas from contract anytime algorithms (Zilberstein & Russell 1995). In particular, we have extended the CanPlan semantics to allow programmer control over planning duration, by introducing a new program construct $\mathsf{Plan}(P, K)$, for limiting the depth of lookahead to K.

The operator $\mathsf{Plan}(P, K)$ was shown to have some useful properties, i.e. $\mathsf{Plan}(P, K)$ is subsumed by CanPlan's $\mathsf{Plan}(P)$ construct, but $\mathsf{Plan}(P, K)$ can find any successful execution that $\mathsf{Plan}(P)$ can find, in situations where no solutions lie outside K. Moreover, $\mathsf{Plan}(P, K)$ prunes failing paths within a depth of K, hence resulting in a higher chance of success with $\mathsf{Plan}(P, K)$ than with regular execution of P. Most importantly, $\mathsf{Plan}(P, K)$ will follow a successful path when there is one within depth K, and $\mathsf{Plan}(P, K)$ will generally[6] detect failure earlier than regular BDI execution. Our theorems also compliment work on anytime algorithms, as shown with the connection between Theorem 6, and the hypothesis in (Drummond et al. 1993).

The operational semantics and theorems form the basis for an implemented system that extends the popular Metric-FF (Hoffmann 2003) planner. Moreover, they provide a theoretical framework for experimental work on simulating how humans behave in real organisations (Dekker & de Silva 2006).

A possible direction for future work involves extending our semantics to suit interruptible anytime algorithms, so that the planner can be interrupted to obtain a partial solution with some guaranteed quality. Basically, the steps involved would be as follows: (i) call the existing $\mathsf{Plan}(P, K)$ construct multiple times, starting with an arbitrary value for K, and: (ii) on each successive call, increase K by one. The first step will create the interruptible algorithm, allowing the solution at the most recently completed depth to be returned on interruption, and the second step is equivalent to the ideal scenario of exponentially increasing the contract algorithm's time allocation (see (Russell & Zilberstein 1991) for a similar example using the RTA^* algorithm). The results from (Zilberstein 1995) may help as a starting point in the formalisation. The semantics could also be extended to capture interleaved planning and execution, i.e. continue to plan for the next K steps, while actions from the first K steps are being executed (e.g. (Miura & Shirai 2001)).

6 Acknowledgements

We would like to thank Sebastian Sardina for feedback on the operational semantics, and for helpful discussions during the initial stages of this work. We would also like to thank the anonymous reviewers for their useful feedback, and the Australian Research Council for their support through grant "Learning and Planning in BDI Agents" (number LP0560702).

[6] $\mathsf{Plan}(P, K)$ will not detect failure earlier than regular execution of P, if failure happens on the $K + 1^{th}$ step.

References

Atkins, E. M., Durfee, E. H. & Shin, K. G. (1996), Building a plan with real-time execution guarantees, *in* 'AAAI-96 Workshop on Structural Issues in Planning and Temporal Reasoning', pp. 1–6.

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

Briggs, W. & Cook, D. J. (1999), Anytime planning for optimal tradeoff between deliberative and reactive planning, *in* 'Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference', AAAI Press, pp. 367–370.

Busetta, P., Rönnquist, R., Hodgson, A. & Lucas, A. (1999), 'JACK Intelligent Agents - Components for Intelligent Agents in Java, AgentLink News Letter, Agent Oriented Software Pty. Ltd., Melbourne'.

Chien, S., Knight, R., Stechert, A., Sherwood, R. & Rabideau, G. (n.d.), Integrated Planning and Execution for Autonomous Spacecraft, *in* 'Proceedings of the IEEE Aerospace Conference (IAC), Aspen, USA', Vol. 1, pp. 263–271.

de Silva, L. P. & Padgham, L. (2004), A comparison of bdi based real-time reasoning and htn based planning, *in* 'Proc. of Australian Joint Conference on AI', pp. 1167–1173.

Dean, T. & Boddy, M. S. (1988), An analysis of time-dependent planning., *in* 'AAAI', pp. 49–54.

Dean, T., Kaelbling, L. P., Kirman, J. & Nicholson, A. (1993*a*), Deliberation scheduling for time-critical sequential decision making, *in* 'Uncertainty in Artificial Intelligence', pp. 309–316.

Dean, T., Kaelbling, L. P., Kirman, J. & Nicholson, A. (1993*b*), Planning with deadlines in stochastic domains, *in* R. Fikes & W. Lehnert, eds, 'Proceedings of the Eleventh National Conference on Artificial Intelligence', AAAI Press, Menlo Park, California, pp. 574–579.

Dekker, A. & de Silva, L. (2006), Investigating organisational structures with networks of planning agents, *in* 'Proceedings of International Conference on Intelligent Agents, Web Technologies and Internet Commerce (To Appear)', Sydney, Australia.

Dennett, D. (1987), *The Intentional Stance*, MIT Press.

Drummond, M., Swanson, K., Bresina, J. & Levinson, R. (1993), Reaction-first search, *in* 'Proc. of the Int. Joint Conf. on Artificial Intelligence', Chambery, France, pp. 1408–1414.

Erol, K., Hendler, J. & Nau, D. S. (1994), HTN Planning: Complexity and Expressivity, *in* 'Proc. of AAAI-94', pp. 1123–1228.

Fikes, R. E., Hart, P. E. & Nilsson, N. J. (1972), 'Learning and executing generalized robot plans', *Artificial Intelligence* **3**, 251–288.

Gigerenzer, G. & Selten, R. (2002), *Bounded Rationality*, MIT Press.

Goodwin, R. (1994), Reasoning about when to start acting, *in* K. Hammond, ed., 'Proceedings of 2nd International Conference on AI Planning Systems', American Association for Artificial Intelligence, Menlo Park, California, pp. 86–91.

Hansen, E. A. & Zilberstein, S. (2001), 'Monitoring and control of anytime algorithms: A dynamic programming approach', *Artificial Intelligence* **126**(1-2), 139–157.

Hindriks, K. V., de Boer, F. S., van der Hoek, W. & Meyer, J.-J. C. (1999), 'Agent Programming in 3APL.', *Autonomous Agents and Multi-Agent Systems* **2**(4), 357–401.

Hoffmann, J. (2003), 'The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables', **20**, 291–341.

Ingrand, F. F., Georgeff, M. P. & Rao, A. S. (1992), 'An Architecture for Real-Time Reasoning and System Control', *IEEE Expert: Intelligent Systems and Their Applications* **7**(6), 34–44.

Jennings, N. R., Sycara, K. & Wooldridge, M. (1998), 'A roadmap of agent research and development', *Journal of Autonomous Agents and Multi-Agent Systems* **1**(1), 7–38.

Karim, S. & Heinze, C. (2005), Experiences with the design and implementation of an agent-based autonomous uav controller., *in* 'AAMAS Industrial Applications', pp. 19–26.

Korf, R. E. (1990), 'Real-time heuristic search', *Artif. Intell.* **42**(2-3), 189–211.

Miura, J. & Shirai, Y. (2001), Parallelizing planning and action of a mobile robot based on planning-action consistency, *in* 'IEEE Conf. on Robotics and Automation', pp. 1750–1756.

Newell, A. & Simon, H. (1972), *Human Problem Solving*, Prentice Hall, Englewood Cliffs.

Plotkin, G. (1981), A Structural Approach to Operational Semantics, Technical Report DAIMI-FN-19, Dept. of Computer Science, Aarhus University, Denmark.

Rao, A. S. (1996), AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, *in* W. V. Velde & J. W. Perram, eds, 'Agents Breaking Away (LNAI)', Vol. 1038 of *LNAI*, Springer-Verlag, pp. 42–55.

Russell, S. J. & Zilberstein, S. (1991), Composing real-time systems, *in* J. Mylopoulos & R. Reiter, eds, 'Proceedings of the Twelfth International Conference on Artificial Intelligence (IJCAI-91)', Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, pp. 212–217.

Russell, S. & Wefald, E. (1991), *Do the right thing : studies in limited rationality*, Artificial intelligence, MIT Press, Cambridge, Mass.

Sardina, S., de Silva, L. & Padgham, L. (2006), Hierarchical Planning in BDI Agent Programming Languages, *in* 'Proc. of AAMAS-06', pp. 1001–1008.

van Riemsdijk, M. B., Dastani, M. & Meyer, J.-J. C. (2005), Semantics of Declarative Goals in Agent Programming, *in* 'Proc. of AAMAS-05', pp. 133–140.

Winikoff, M., Padgham, L., Harland, J. & Thangarajah, J. (2002), Declarative & Procedural Goals in Intelligent Agent Systems, *in* 'Proc. of KR-02', pp. 470–481.

Zilberstein, S. (1995), 'Operational rationality through compilation of anytime algorithms', *AI Magazine* **16**(2), 79–80.

Zilberstein, S., Charpillet, F. & Chassaing, P. (1999), Real-time problem-solving with contract algorithms, *in* 'IJCAI', pp. 1008–1015.

Zilberstein, S. & Russell, S. (1995), Approximate reasoning using anytime algorithms, *in* 'Natarajan, S. (ed.), Imprecise and Approximate Computation', Kluwer Academic Publishers.