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 & Heine 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 Pins (Ingrand, Georgeff & Rao 1992), AGENTSPEAK (Rao 1996), 3APL (Hindriks, de Boer, van der Hoek & Meyer 1999), JACK (Busseta, Rönquist, 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 1993b). 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
configurations $C'$ that can be reached within a single execution step from $C$. The relation $\xrightarrow{\star}$ denotes the reflexive transitive closure of $\xrightarrow{\tau}$.

The transition relation $\rightarrow$ 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.

$$\Delta = \{ \psi : \theta \mid P \mid e' : \psi_i : P_i \in I \land \theta = \mu \theta(e, e') \}$$

**Event**

$$\begin{array}{l}
(B, A, \psi) \xrightarrow{\tau} (B, A, \Delta) \\
\psi : P_i \in \Delta \Rightarrow B \equiv \psi \theta
\end{array}$$

**Sel**

$$\begin{array}{l}
(B, A, (\Delta)) \xrightarrow{\tau} (B, A, \theta \circ (\Delta \setminus P_i))
\end{array}$$

**Seq**

$$\begin{array}{l}
(B, A, (P_i \triangleright P_2)) \xrightarrow{\tau} (B', A', (P' \triangleright P_2))
\end{array}$$

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 $\triangleright$ rule states that, as long as there is more to decompose in $P_1$, $P_1$ should be decomposed. On reaching the program $nil \triangleright P$, rule $\triangleright$ states that there is no need for program $P$, as the left hand side of $\triangleright$ has been successfully decomposed. The rule *Seq* handles the execution of two programs in a sequence, and on reaching the sequence $nil \triangleright P$, rule *Seq* removes the program $nil$ and continues with the right hand side of the sequence. The rule that tests for a condition $\phi$ simply succeeds if the condition $\phi$ is met in the current state, and the act rule applies an action to the current state of the world, on the basis that $\psi$ 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 moveEvent, 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.
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.: 

\[
B_0 = \langle \text{occupiedBy}(John, MondayDay) \land \text{availableSlotFor}(John, MondayDay) \land \text{availableSlotFor}(John, TuesdayDay) \land \text{meetingRequest}(David) \rangle
\]

The agent execution will then proceed as follows, where the rules used are Event, Sel, Event, Sel, \( \succ \), and Seq, respectively, from top to bottom:

\[
\begin{align*}
(\langle B_0, A_0, !sm \rangle, \text{Event}) & \rightarrow (\langle B_0, A_0, \{!fss; \text{les}; \text{las}\} \rangle, \text{Event}) \\
(\langle B_0, A_0, \{!fss; \text{les}; \text{las}\} \rangle, \text{Event}) & \rightarrow (\langle B_0, A_0, \{!fss; \text{les}; \text{las} \succ \psi\} \rangle, \text{Event}) \\
(\langle B_0, A_0, \{!fss; \text{les}; \text{las} \succ \psi\} \rangle, \text{Event}) & \rightarrow (\langle B_0, A_0, \{!fss; \text{les}; \text{las} \succ \psi\} \rangle, \text{Sel}) \\
(\langle B_0, A_0, \{!fss; \text{les}; \text{las} \succ \psi\} \rangle, \text{Sel}) & \rightarrow (\langle B_0, A_0, \{!fss; \text{les}; \text{las} \succ \psi\} \rangle, \text{Seq}) \\
(\langle B_0, A_0, \{!fss; \text{les}; \text{las} \succ \psi\} \rangle, \text{Seq}) & \rightarrow \ldots
\end{align*}
\]

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. \( !fss; \text{les}; \text{las} \). The only rule that can work with \( \{\Delta\} \) is Sel, which selects \( !fss; \text{les}; \text{las} \) from \( \{\Delta\} \) and removes the plan body from \( \{\Delta\} \). The first program in \( !fss; \text{les}; \text{las} \), 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 removes \( nil \) from the program \( !fss; \text{les}; \text{las} \), 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 \( D = (II, A) \) consists of a library \( II \) of methods and a library \( A \) of primitive tasks. Each primitive task in \( A \) 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 \( P \) is the triple \( \langle d, B, D \rangle \) where \( d \) is the network of tasks to accomplish, \( B \) is the initial belief state (i.e. a set of all ground atoms that are true in \( B \)), and \( D \) is a planning domain. A plan \( \sigma \) is a sequence \( act_1 \ldots act_n \) of ground actions (that is, ground primitive tasks).

Given a specific planning problem \( P \), HTN planning proceeds by selecting an applicable reduction method from \( D \), 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 \( \sigma \), 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 \( sol(d, B, D) \) that solves a planning problem instance \( P = \langle d, B, D \rangle \).

\[
\begin{align*}
sol_1(d, B, D) &= \text{comp}(d, B, D), \\
sol_{n+1}(d, B, D) &= \bigcup_{d' \in \text{red}(d, \{d', B, D\})} \text{sol}(d', B, D), \\
sol(d, B, D) &= \bigcup_{n<\omega} \text{sol}(d, B, D).
\end{align*}
\]

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

#### 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 \( \text{Plan}(P) \). \( \text{Plan}(P) \) 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 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 \( C \xrightarrow{\text{bdi}} C' \) specifies a single step transition of type bdi, corresponding to a single execution step of an agent. Similarly \( C \xrightarrow{\text{plan}} C' \) specifies a single step transition of type plan, corresponding to a single step of imagined execution within the planner. A transition without a label \( C \xrightarrow{\text{plan}} C' \) specifies that either type applies.

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

\[
\begin{align*}
\langle B, A, \text{Plan}(P) \rangle &\xrightarrow{\text{bdi}} \langle B', A', \text{Plan}(P') \rangle \\
&\xrightarrow{\text{plan}} \langle B'', A'', \text{nil} \rangle
\end{align*}
\]

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

The rule states that a configuration $\langle B, A, Plan(P) \rangle$ can make a single BDI step to $\langle B', A', Plan(P') \rangle$ provided that $\langle B, A, Plan(P) \rangle$ can make a single plan step to $\langle B', A', P' \rangle$, from where it is possible to reach a final configuration $\langle B'', A'', nil \rangle$ 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: interrupting a plan and re-planning is always permissible.

It was proven in (Sardina et al. 2006) that $Plan(P)$ can be seen as an integrated HTN planner, operating on the same domain knowledge as the BDI system. However, $Plan(P)$ 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 $Plan$ $\langle seqActions; \emptyset \rangle$ (Sardina et al. 2006). The event $seqActions$ can be handled by multiple pro-grams, 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. $Plan(P_a) \supset Plan \langle seqActions; \emptyset \rangle$, which specifies that first-principles planning should be tried, only after failure of the HTN planning represented by $Plan(P_a)$.

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 $clearSlot(s)$ occurring in the plan body for $scheduleMeeting(p)$, with $Plan(\langle clearSlot(s) \rangle)$. The behavior of the BDI system on reaching the $Plan(\langle clearSlot(s) \rangle)$ program is shown below.

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

\[ (C, lcs) \rightarrow (C, \langle \{ \emptyset \} \rangle) \rightarrow (C, \langle \{ \emptyset \} \rangle) \rightarrow (C, \langle \{ \emptyset \} \rangle) \rightarrow (C, \langle \{ \emptyset \} \rangle) \rightarrow Plan \]

The antecedent of the above rule is met as a result of applying $Event$, $Sel$, $Event$, $Sel$ and $Seq$ (etc.)

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.
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, signaling 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: \( t, 2t, 4t, \ldots, 2^i t \), 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 \( 2^2t < t_1 < 2^3t \), the solution returned will be the one produced by the contract algorithm with time allocation \( 2^2t \). If the algorithm were interrupted at some time \( t_2 \), where \( 2^2t < t_2 < 2^3t \), the solution returned would be from the contract algorithm with time allocation \( 2^3t \). 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 \( \text{Plan}(P, K) \), where \( K \) is a number specifying the maximum depth for lookahead within \( P \). More specifically, \( \text{Plan}(P, K) \) means “plan for \( P \) offline, searching for a complete hierarchical decomposition within depth \( K \).”

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

\[
P := \text{nil} \mid \text{act} \mid \langle \text{b} \rangle \mid \langle \text{b} > \mid \langle \text{b} = \text{b} \rangle \mid \text{P} ; P_2 \mid P_1 \bowtie P_2 \mid P_1 \parallel P_2 \mid P_1 || \text{P}_2 \mid \text{P}_n.
\]

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

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

The \( \text{Plan}_{\text{Succ}} \) rule states that configuration \( \langle B, A, \text{Plan}(P, K) \rangle \) can evolve to \( \langle B', A', \text{Plan}(P', K) \rangle \) provided that \( \langle B, A, P \rangle \) can evolve to \( \langle B', A', P' \rangle \), 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 \( \text{Plan}(P, K) \) 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 \( \text{Plan}(P, K) \) does not. Hence by making \( \text{Plan}(P, K) \) behave optimistically, there is a guarantee that the agent will behave no worse than the regular execution of \( P \) (shown later).

The optimistic rule \( \text{Plan}_{\text{Opt}} \) states that configuration \( \langle B, A, \text{Plan}(P, K) \rangle \) can evolve to \( \langle B', A', \text{Plan}(P', K) \rangle \) provided that \( \langle B, A, P \rangle \) can evolve to \( \langle B', A', P' \rangle \), from where it is possible to reach some configuration in \( K - 1 \) steps. Hence \( \text{Plan}_{\text{Opt}} \) 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 \( \text{Plan}_{\text{Opt}} \) ensures that this rule is not applicable when \( \text{Plan}_{\text{Succ}} \) is applicable.

The simpler rule \( \text{Plan}_K \) handles planning for program \( \text{nil} \) i.e. when \( P \) has successfully been executed within \( \text{Plan}(P, K) \). The rule simply returns success from \( \text{Plan}(P, K) \), by replacing \( \text{Plan}(\text{nil}, K) \) with \( \text{nil} \). The second simpler rule \( \text{Plan}_P \) handles the \( \text{Plan}(P, K) \) construct from within a planning context. If a \( \text{Plan}(P, K) \) construct is nested within another \( \text{Plan}(P, K) \) 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 \( \text{Plan}_n \) and \( \text{Plan}_p \), defined in (Sardina et al. 2006) for the \( \text{Plan}(P) \) construct.

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

4 Theoretical Properties

In this section we will discuss the formal properties of the \( \text{Plan}(P, K) \) 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 \( \langle N, \Lambda, \Pi, B, A, \Gamma \rangle \) where \( N \) is the name of the agent, \( \Lambda \) is an action library, \( \Pi \) is a library of plan rules, \( B \) is a belief base, \( A \) is a sequence corresponding to actions already executed, and \( \Gamma \) is a set of plan bodies currently being executed. Agent transitions \( \Rightarrow \) are defined by the following three rules.

\[
P \in \Gamma \quad \langle B, A, P \rangle \quad \langle B', A', P' \rangle \quad \langle N, \Lambda, \Pi, B, A, \Gamma \rangle \Rightarrow \langle N, \Lambda, \Pi, B', A', \langle P \setminus \{P'\} \rangle \cup \{P'\} \rangle.
\]

Next, a new external event \( e \) is a new external event \( \langle N, \Lambda, \Pi, B, A, \Gamma \rangle \Rightarrow \langle N, \Lambda, \Pi, B, A, \Gamma \cup \{e\} \rangle \).
Rule $A_{\text{step}}$ performs a single step in a plan body within $\Gamma$, rule $A_{\text{event}}$ appends a new external event $e$ to $\Gamma$, and rule $A_{\text{clean}}$ removes a plan body that is completed (i.e., one which has reached nil, or one from where no transition is possible) from $\Gamma$. 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 $E$ of an agent $C_0 = \langle N, A, II, B, A, \Gamma_0 \rangle$ is a possibly infinite sequence of agent configurations $C_0 \ldots C_n \ldots$ such that $C_i \Rightarrow C_{i+1}$, for every $i \geq 0$. A terminating execution is a finite execution $C_0 \ldots C_n$ where $C_n = \langle N, A, II, B, A, \{\} \rangle$. An environment-free execution is one in which rule $A_{\text{event}}$ has not been used.

**Definition 2 (Intention Execution) Let $E$ be a BDI execution $C_0 \ldots C_n \ldots$ for an agent $C_0 = \langle N, A, II, B, A, \Gamma_0 \rangle$, where $\Gamma_0 = \Gamma_0^0 \cup \{P_0\}$. Intention $P_0$ in $C_0$ has been fully executed in $E$ if $P_n = e$; 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 $i \leq n$; otherwise $P_0$ has failed in $E$ if it has been fully but not successfully executed in $E$.

**Definition 3 Two, possibly derived, agent executions $C_0 \ldots C_n$ and $C'_0 \ldots C'_n$ are equivalent modulo intentions if $C'_0 = \langle N, A, II, B, A, \{\} \rangle$ for every $0 \leq i \leq n$. Also, the two execution plans $P_0 \ldots P_n$ in $C_0$ and $P'_0 \ldots P'_n$ in $C'_0$ are equivalent modulo intentions if they are equivalent modulo intentions and for every $0 \leq i \leq n$, $\Gamma_0 \setminus \{P_i\} = (\Gamma_0^0 \setminus \{P_i\})$ (where $P_i$'s $(P'_i$'s) evolution in configuration $C_i$ $(C'_i)$).

**Definition 4 (Program Simulation) Let $E$ be an execution of $C = \langle N, A, II, B, A, \Gamma \cup \{P\} \rangle$. Program $P'$ simulates program $P$ in execution $E$ if there is an execution $E'$ of configuration $C' = \langle N, A, II, B, A, \Gamma \cup \{P'\} \rangle$ such that (a) $E$ and $E'$ 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 $P$ if $P'$ simulates $P$ in every execution of any configuration.

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

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

**Proof.** The proof relies on the following lemma from (Sardina et al. 2006): if $\langle B, A, P \rangle \longrightarrow^* \langle B_f, A_f, nil \rangle$, then $\langle B, A, Plan(P) \rangle \longrightarrow^* \langle B_f, A_f, nil \rangle$.

Let $C = \langle N, A, II, B, A, \{Plan(P, K)\} \rangle$ and let $C' = \langle N, A, II, B, A, \{Plan(P)\} \rangle$. Contrary to the theorem, suppose there is some successful execution $E$ for $C$ of the form $C_0 = C \ldots C_n$ such that $C_n = \langle B_s, A_n, nil \rangle$ for some $k \leq K$ and that an equivalent modulo Plan($P$, $K$) and Plan($P$) execution $E'$ does not exist for $C'$. From the Plan($P, K$) derivation rules Plan$_{\text{Ksucc}}$ and Plan$_{\text{Kopt}}$, $E$ exists because $\langle B, A, Plan(K, P) \rangle \longrightarrow^* \langle B_f, A_f, Plan(nil, K) \rangle \longrightarrow^* \langle B_k, A_k, nil, nil \rangle$ holds. Therefore the antecedent of Plan$_{\text{Ksucc}}$: $\langle B, A, Plan(nil, nil) \rangle \longrightarrow^* \langle B_f, A_f, nil \rangle$, also has to hold. From the above lemma, if $\langle B, A, P \rangle \longrightarrow^* \langle B', A', nil \rangle$, then $\langle B, A, Plan(P) \rangle \longrightarrow^* \langle B', A', nil \rangle$ holds, using the above lemma and derivation rule Plan$_{\text{Rn}}$. $\langle B, A, Plan(nil) \rangle \longrightarrow^* \langle B_k, A_k, nil \rangle$ also holds. Therefore $E'$ has to exist. ■

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

**Theorem 2** Let $C = \langle N, A, II, B, A, \{Plan(P)\} \rangle$. If for every belief base $B'$, sequence of actions $A'$, program $P'$ and $i \geq K > 0$ such that $\langle B, A, P \rangle \longrightarrow^* \langle B', A', P' \rangle$, it is the case that $\langle B', A', P' \rangle \longrightarrow^* \langle B'', A'', nil \rangle$, then for every value of $K$, Plan($P$, $K$) simulates Plan($P$) in all executions of $C$.

**Proof.** On the contrary, suppose that there is some successful execution $E$ of $C = \langle N, A, II, B, A, \{Plan(P)\} \rangle$ of the form $C_0 \ldots C_k$ where $C_k = \langle B_k, A_k, nil \rangle$, for which there is no equivalent modulo Plan($P$) and Plan($P$, $K$) execution $E'$ of $C' = \langle N, A, II, B, A, \{Plan(P)\} \rangle$. Observe that for $E$ to exist, $\langle B, A, Plan(P) \rangle \longrightarrow^* \langle B_k, A_k, nil \rangle$ holds, and from the antecedent of the Plan rule, $\langle B, A, Plan(P) \rangle \longrightarrow^* \langle B_k, A_k, nil \rangle$ also holds. Furthermore, $0 \leq j \leq K$ holds as the theorem states that there is no successful execution past $K$. From Lemma 3, $\langle B, A, Plan(K) \rangle \longrightarrow^* \langle B_k, A_k, nil \rangle$ holds and therefore the execution $E'$ must exist. ■

In summary, the above two theorems show that Plan($P$, $K$) is not as powerful as Plan($P$) in general, but in certain specific situations, Plan($P$, $K$) is as powerful as Plan($P$). Next we move on to the relationship between 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 $B, A$ and $P$, if $(B, A, P)\xrightarrow{\text{plan}} (B_j, A_j, \text{nil})$, then $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i} (B_j, A_j, \text{nil})$ 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 $n = 0$: $(B, A, P)\xrightarrow{\text{plan}} (B_j, A_j, \text{nil})$ holds for $P = \text{nil}$, $B_j = B$, and $A_j = A$, and $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i} (B_j, A_j, \text{nil})$ also holds using derivation rule PlanK.

Inductive case $n = m + 1$, for $0 < n \leq K$: Then, there exists $(B', A', P')$ such that (a) $(B, A, P)\xrightarrow{\text{plan}} (B', A', P')$, and; (b) $(B', A', P')\xrightarrow{\text{plan}} (B_j, A_j, \text{nil})$. For (a) we can use derivation rule PlanSucc to obtain $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i} (B', A', \text{Plan}(P', K))$. Moreover, using (b) and the induction hypothesis, $(B', A', \text{Plan}(P', K))\xrightarrow{\text{bd}, i} (B_j, A_j, \text{nil})$. Thus $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}, (B_j, A_j, \text{nil})$, for all $0 \leq i \leq K$ follows. ■

The next theorem states that, if failure within $K$ steps is inevitable, 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 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 $C = \langle N, A, I, \Pi, B, A, \{\text{Plan}(P, K)\} \rangle$ such that, $(B, A, \text{Plan}(P, K)) \xrightarrow{\text{bd}, i}$ for every belief base $B'$, sequence of actions $A'$, program $P'$ and $0 \leq i < K$ such that $(B, A, P)\xrightarrow{\text{plan}} (B', A', P')$, and for every belief base $B''$, sequence of actions $A''$ and program $P''$ such that $(B, A, P)\xrightarrow{\text{plan}^{K-i}} (B'', A'', P'')$, it is the case that $(B', A', P')\xrightarrow{\text{plan}} (B'', A'', \text{nil})$ and $(B'', A'', P'')\xrightarrow{\text{plan}}$.

**Proof.** On the contrary, suppose:

1. $(B, A, \text{Plan}(P, K)) \xrightarrow{\text{bd}, i}$ and $(B, A, P)\xrightarrow{\text{plan}} (B', A', \text{nil})$ for some $0 \leq i < K$. However, from Lemma 3, if $(B, A, P)\xrightarrow{\text{plan}} (B', A', \text{nil})$, then it is the case that $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$, for some $0 \leq i < K$. Therefore $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$ cannot hold.

2. $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$ and $(B, A, P)\xrightarrow{\text{plan}^{K}} (B', A', P')$. However, since $(B, A, P)\xrightarrow{\text{plan}^{K}}$, from derivation rule PlanKopt, it is the case that $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$. Therefore $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$ cannot hold. ■

Theorem 5 and 6 shows the situations under which 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 $K$, Plan($P, K$) will take a single step only if that step is one that eventually leads to a solution. Therefore 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 $C = \langle N, A, I, \Pi, B, A, \{\text{Plan}(P, K)\} \rangle$ such that, $(B, A, \text{Plan}(P, K)) \xrightarrow{\text{bd}, i}$ and $(B, A, P)\xrightarrow{\text{plan}} (B', A', \text{nil})$ for some $0 < i < K$. If $E$ is an environment-free agent execution of $C$, then intention Plan($P, K$) is either executing or has been successfully executed in $E$. Moreover, there is an execution $E^*$ of $C$ in which intention Plan($P, K$) has been successfully executed in $E^*$.

**Proof.** This follows directly from Lemma 3. ■

As mentioned before, Plan($P, K$) behaves optimistically, and Theorem 6 confirms that this is indeed the case. If there is no solution within depth $K$, Plan($P, K$) will take a single step, only if there is an execution of length $K$ from $P$. Therefore 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 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 Plan($P, K$), than with regular execution of $P$, as Plan($P, K$) will prune all paths that fail within depth $K$.

**Theorem 6** Let $C = \langle N, A, I, \Pi, B, A, \{\text{Plan}(P, K)\} \rangle$ such that, $(B, A, \text{Plan}(P, K)) \xrightarrow{\text{bd}, i}$ and such that for every belief base $B'$, sequence of actions $A'$, program $P'$ and $0 < i < K$ such that $(B, A, P)\xrightarrow{\text{plan}} (B', A', P')$, it is the case that $(B', A', P')\xrightarrow{\text{plan}} (B'', A'', \text{nil})$. If $E$ is an environment-free agent execution of $C$, then intention Plan($P, K$) is either executing or has executed $K$ steps in $E$.

**Proof.** The proof relies on the following lemma: For every $B, A$ and $P$, if $(B, A, P)\xrightarrow{\text{plan}} (B', A', \text{nil})$ and $(B, A, P)\xrightarrow{\text{plan}} (B', A', P')$, then $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i} (B', A', \text{Plan}(P', K))$, for some $0 \leq i < K$.

On the contrary suppose there is an environment-free execution $E$ of the form $C_0 = C \cdot \ldots \cdot C_k$ such that $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$ for some $K < K$. However, observe that $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i} (B, A, \text{Plan}(P, K))$ will hold. From the antecedent of the derivation rule PlanKopt, we can see that $(B, A, P)\xrightarrow{\text{plan}} (B, A, \text{Plan}(P, K))$, and hence $(B, A, P)\xrightarrow{\text{plan}} (B', A', P')$. However, since $(B, A, P)\xrightarrow{\text{plan}}$(from derivation rule PlanKopt), it is the case that $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$. Therefore $(B, A, \text{Plan}(P, K))\xrightarrow{\text{bd}, i}$ cannot hold. ■

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, Charplillet & 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 Plan(P, K), for limiting the depth of lookahead to K.

The operator Plan(P, K) was shown to have some useful properties, i.e. Plan(P, K) is subsumed by CanPlan’s Plan(P) construct, but Plan(P, K) can find any successful execution that Plan(P) can find, in situations where no solutions lie outside K. Moreover, Plan(P, K) prunes failing paths within a depth of K, hence resulting in a higher chance of success with Plan(P, K) than with regular execution of P. Most importantly, Plan(P, K) will follow a successful path when there is one within depth K, and Plan(P, K) will generally 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 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).

References


