Planning on demand in BDI systems

Lavindra de Silva and Lin Padgham

{*ldesilva,linpa*}*@cs.rmit.edu.au*
School of Computer Science and Information Technology
RMIT University, Melbourne 3000, VIC, Australia

1   Introduction

The BDI (Belief, Desire, Intention) model of agency is a popular architecture based on Bratman's (Bratman 1987) theory of practical reasoning. There are numerous implementations based on the BDI architecture such as PRS (Georgeff & Ingrand 1989) and JACK[1], which are used for both academic and industrial purposes. An important aspect of BDI style systems is that they execute as they reason, and so avoid the possibility of the reasoning being outdated, due to environmental change, by the time execution happens. This makes them useful for many complex and dynamic environments, such as Unmanned Autonomous Vehicles (UAVs) and Air Traffic Management, due to their ability to cope well with changes, making adjustments as they go in terms of the steps chosen. They are also very fast, and therefore well suited to systems needing to operate in real time, or close to real time environments. However, there are no generic mechanisms in BDI systems to do any kind of look-ahead, or planning. In some situations this would be desirable.

The primary goals and contributions of our work are: 1) incorporating planning at specific points in a BDI application, on an as needed basis, under control of the programmer; 2) planning using only limited subsets of the application, making the planning more efficient, and; 3) incorporating the plan generated back into the BDI system, for regular BDI execution, identifying plan steps that could be pursued in parallel.

Other features of our approach include: 1) minimising the programming overhead, as the program to be run by the planner is derived from the existing BDI program; 2) allowing the use of regular functions in planning; 3) extracting planning effects from existing code, and; 4) incorporating aspects of both HTN (Hierarchical Task Networks) planning and classical planning.

There is some previous work that deals with using planning capabilities to guide the execution of BDI-like systems. Some of the research closely related to ours is Propel (Levinson 1995), Propice-Plan (Despouys & Ingrand 1999), and RETSINA (Paolucci *et al.* 1999). Propice-Plan (and other similar systems) use a reactive system for execution, and a separate planning component to produce plans. There are three important differences between our system and Propice-plan : 1) the planner provides an alternative means of operation; it is not smoothly integrated into the BDI architecture and algorithms. In particular, the planner does not provide guidance to the reactive-planner, on how to use its hierarchy to achieve the goal, instead, the reactive planner only executes the operators in the final plan produced; 2) Propice-plan does not allow the programmer to specify points when the planner should be called, and to provide extra operators for each planning point; 3) Propice-plan does not support the use of programmer provided functions in planning, i.e. where the outcome cannot be specified as an effect. Propel and RETSINA are different to Propice-plan in that planning and execution are tightly integrated within a single framework. However, in Propel, the planner is called every time a runtime failure occurs, whereas our system allows the programmer to specify when to call the planner, and what part of the hierarchy to use for planning. In RETSINA, planning is done when all the information required for planning is available, therefore not leaving the decision of when to plan and when to execute to the programmer.

A simplified view of the BDI architecture is in terms of *goals* and *recipes*, where a goal has one or more recipes that can be used to achieve it. The recipes to achieve goals are stored in a library provided by the programmer. When an agent has a goal to achieve, it looks for a recipe that can achieve the goal in the current state of the world. If a suitable recipe is found, it is executed. If it fails during execution, the agent looks for other suitable recipes to achieve the goal. The goal fails if none of its recipes could be executed to completion, or if none of them were suitable for the current state of the world.

In achieving a goal, the agent typically executes a number of steps, or *subgoals/subtasks*. In some situations there can be multiple options (recipes) at each step, but for a given state, only certain combinations of choices will lead to success of the overall goal. However, it may not be possible to (easily) encode information enabling successful choices, based only on knowledge of the current state. If steps are cheap, reversible, and there is little or no cost to doing them, then a BDI system can easily be used to find a solution in these situations. However, in many cases of this sort in real applications, actually doing the actions rather than only rea-

[1] JACK Intelligent Agents : http://www.agent-software.com.au
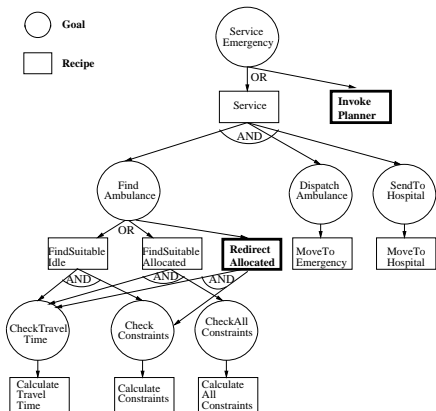
Figure 1: The Design of the BDI Program. Additional planner recipes are highlighted in bold.

soning about them (planning), results in wasting time and resources in ineffectual behaviour. At times this can even lead to inability to achieve the goal. While it is always possible for planning to be explicitly programmed into a BDI application (and in fact this is often done), it would be advantageous to have a simple mechanism that incorporates planning in a generic way, at particular points where it is needed.

2 Example

Let us present an example to illustrate the inclusion of a planning facility within a BDI application. Consider an ambulance dispatch system: as ambulance requests arrive, a dispatching agent ascertains the best ambulance to dispatch and provides ongoing directions during the process of dealing with the emergency.

The steps that are part of servicing the emergency are *FindAmbulance*, *DispatchAmbulance* and *SendToHospital*, in that order, as shown in Figure 1 (ignore recipes in bold for now). Normally the FindAmbulance goal is achieved by executing the *FindSuitableIdle* recipe. This is a recipe type representing a set of recipe instances. These instances are determined at runtime, one for each idle ambulance. Each of these recipe instances will be tried in turn, attempting to successfully complete the subtasks of *CheckTravelTime* and *CheckConstraints*. If no recipe instance of this type completes successfully, instances of the alternative recipe type *FindSuitableAllocated* will be tried. This recipe attempts to schedule the new emergency to follow one of the jobs that is already allocated.

However, if all instances of all applicable plan types fail, i.e. are unable to be done in a way that meets the time constraints for the new emergency, the *ServiceEmergency* goal will fail. Since at this point the BDI system cannot find a solution, it could be useful to call a planner to see if there is a way to successfully achieve this *ServiceEmergency* goal, while also servicing all other such goals and maintaining all timing constraints. The planner could attempt to develop a plan that reallocates some of the currently allocated emergencies, so that the new emergency can also be serviced.

For example, the situation may be that there are two ambulances and two hospitals, placed on the grid as shown in Figure 2. Three emergencies occur at different times, at locations also shown on the grid. The table beside the grid

Figure 2: Ambulance servicing example

gives the number of time units after the start that each emergency happens and the urgency of the emergency. Urgency is in terms of maximum time units acceptable between the emergency happening, and the ambulance having serviced the emergency arriving at the hospital. We assume that the ambulances can move right, left, up and down, and that each move takes one time unit.

Using the BDI recipes in Figure 1, the first two emergencies will be allocated to the two ambulances. Assume E1 is allocated to ambulance A1, and to hospital H1, while E2 is allocated to ambulance A2, and also to hospital H1. It will take eight time units to service each of E1 and E2.

When emergency E3 occurs, the ambulances are busy, so the recipe *FindSuitableAllocated* will be tried, to see if E3 can be scheduled with either A1 or A2, after their current job, in a way that meets the urgency constraints. In this case A1 and A2 arrive at H1 at time units 8 and 10 respectively (from the start time). The trip from H1, to E3, and then to H2 takes 10 units. As it is necessary to service E3 by 20 time units from the start (occurrence time plus urgency allowance) it is possible to allocate E3 to be done by A1 after it has reached H1. This will have E3 at the hospital at time unit 18, 2 time units before the maximum for its urgency level.

However, if E3 had had an urgency level of 10 units, then this allocation would not have been successful. By calling a planner, and exploring possible re-allocations, it would be possible to allocate A1 to service E1 and E2, while A2 services E3. To arrive at this set of allocations the planner must find an appropriate set of bindings to meet all constraints for the three *ServiceEmergency* goals. I.e. find a set of bindings for the first goal, project it's effects, then find a set of bindings for the next goal based on the projected state, and continue in this manner. If a set of bindings could not be found for a goal, an alternative set of bindings would need to be found for the previous goal.

3 Overview

What we propose in this work is a mechanism whereby a BDI programmer can indicate that runtime planning should be applied. This may be on failure of a more simple approach (as in our example), or in a situation where planning may always be appropriate for a particular task or subtask. Our approach is to use the information already available within the BDI program as the knowledge which must be provided to the planner, and to thereby relieve the programmer of the responsibility of specifying information specifically for planning. The planner can then make choices about which recipe instances to use and how these are sequenced.

In order to provide the required information to the planner, the relevant BDI goals and/or recipes, as well as the relevant information about the current state as captured by agent beliefs, must be translated into a representation suit-

able for some planning system. Due to the similarity of problem representation between BDI systems and HTNs (de Silva & Padgham 2004), we decided to use an HTN planner which allows representation of the information encoded in the hierarchical BDI structure. We have decided to use the Java version of the SHOP planner called JSHOP[2] as it can handle numeric computations and also allows user defined Java functions to be used during planning. The BDI platform we are using is JACK, BDI agent platform, with research licensing available to universities.

The approach that we use is to have the programmer include in the BDI program a generic recipe that invokes the HTN planner at the desired point. At compile time our system then automatically converts the relevant goals and recipes[3] into a JSHOP program, which can be accessed at runtime, if the recipe invoking the planner is instantiated. After invoking the planner, the recipe executes the plan returned. If desired, additional recipes can be provided within the BDI program, for the express purpose of being available for use in planning. A suitable context condition can ensure that they are not instantiated at other times.

The process of converting information from the JACK program, into a suitable JSHOP representation, is a straightforward one-one mapping of JACK's goals and recipes (including each recipe's precondition and effect), into JSHOP's syntax (i.e. *methods*, and *precondition* and *tail* pairs). Due to space constraints, the mapping process will not be discussed here. The reader can refer to http://www.cs.rmit.edu.au/Īdesilva/research/publications/ for a more detailed paper.

The bold recipes in Figure 1 show the recipe *InvokePlanner* which calls the planner, and also recipe *RedirectAllocated* provided specifically for use by the planner.

4 Invoking the Planner

As indicated in section 3, a recipe is placed at whatever point the programmer wishes the planner to be invoked. This recipe will be chosen according to normal BDI principles. Therefore it can have a context condition which captures the situation in which it should be used, or it can be a last priority, if other options have failed (as in our example situation), or it can be prioritised with respect to alternative options in the same way that other alternative plans are prioritised (in JACK using either a precedence attribute or order of declaration).

The recipe for invoking the planner has a generic form, and therefore most of the details are added automatically, based on a template. There are four subtasks which are included within this recipe. These are to: 1) create the initial set of beliefs, corresponding to the current state when the planner is to be invoked; 2) create the desired goal state; 3)

instantiate and call the planner, and; 4) execute the plan.

The list of beliefs that are relevant, and whose values should be accessed at runtime, and provided to JSHOP, must be specified by the programmer (via a GUI). (These could be extracted automatically, but this has not currently been done.) In our ambulance example, relevant beliefs are *AmbulanceState*, *EmergencyState* and *HospitalState* and values obtained at runtime include *A1* is at *4,3* and *A2* is at *2,3* from AmbulanceState, *E1* is at *5,3*, *E2* is at *2,4*, and *E3* is at *2,1* for EmergencyState and so on. When the *InvokePlanner* recipe is executed, the current values are obtained and the initial state for JSHOP is built.

The desired goal state must be provided by the programmer in the form of a recipe which can produce the desired goal state for the specific situation. For the ambulance example, the desired goal state is: *EmergencyState E1 == Serviced*, *EmergencyState E2 == Serviced*, and *EmergencyState E3 == Serviced*. This is produced using a recipe which obtains all current emergencies, including the new one, and creates a beliefset instance which has all of them as being serviced. An interface is currently being developed to support creation of most of this recipe automatically.

The third subtask is to call the appropriate JSHOP program with the initial state and goal state produced by the first two subtasks. The final task is to execute the plan which is returned. This is done using a recipe supplied by our system, and is explained in section 6.

The planner is able to do both planning by reduction within a particular goal, and also planning with multiple instances of top level goals, in a manner similar to how first principles planners plan with multiple "instances" of operators. The latter is possible because HTNs are more expressive than STRIPS style planners, as described in (Erol, Hendler, & Nau 1996). The ability to combine these techniques allows for considerable flexibility, allowing user defined functions, and a lossless representation of the BDI hierarchy.

5 Producing a Suitable Partial-Order Plan

Certain planners, including JSHOP, produce a totally ordered sequential plan. However BDI systems like JACK are intended to be able to pursue multiple goals in parallel. In order to take advantage of this aspect of BDI systems, it is desirable to parallelise the resulting JSHOP plan before beginning execution within JACK.

An alternative would be to directly use a partial-order planner. However, none of the partial-order HTN planners are Java based and would therefore not facilitate the direct integration of functions that we require. Following the algorithm described in (Veloso, Pérez, & Carbonell 1991), with minor changes, we have modified JSHOP to have a final processing step which creates a partial-order plan.

In its original form, JSHOP produces a sequence of primitive actions as the outcome of planning. These are parts of JACK recipes. We now need to execute the plan found, and we do that by incorporating it back into JACK, as there may be other aspects of the original recipes that require execution, as well as those parts that were provided to the planning process. As JACK executes recipes as a result of goals being instantiated, we need our resulting plan to be in terms

---

[2]JSHOP is a state of the art planner being used by the *Naval Research Laboratory for Noncombatant Evacuation Operations* http://www.cs.umd.edu/projects/shop/download.html

[3]The relevant goals and recipes are the recipes which are siblings of the planning recipe in the BDI hierarchy, and all their children recipes and goals. Other goals and recipes in the program are excluded because they are not responsible for achieving the task for which runtime planning is desired.

of goals. Consequently we have modified JSHOP to provide information on the reduction process itself; i.e. for each operator/action in the plan, which recipe and goal instance were selected to lead to that action[4].

6   Executing the JSHOP Plan

The partial-order plan returned from the planner consists of a partial order of nodes. Each node contains the top level goal, and all information necessary for binding variables and making choices of recipes as that goal is executed.

The recipe provided by our system posts the top level goals in the appropriate order. It initially posts asynchronously, all goals at the start of the plan which can be run in parallel. As each top level goal completes, any immediate successor, for which all predecesors have completed, is posted. In our example, the goal instances *SeviceEmergency E1* and *ServiceEmergency E3* are posted initially. When *ServiceEmergency E1* completes, *ServiceEmergency E2* is posted, as it is dependent only on *ServiceEmergency E1* in the partial order. If the ordering was such that *ServiceEmergency E2* was after both *ServiceEmergency E1* and *ServiceEmergency E3*, then it would not be posted until both completed.

When each goal is posted (both the top level goal and the subsequent subgoals), the BDI system must decide the appropriate recipe to use. This is based on the plan that has been returned by the planner. We require firstly that the recipe instance chosen is of the same type as that indicated by the plan. Secondly it must contain the same bindings in the context condition as that indicated in the plan.

Recipe selection is handled transparently to the programmer, by extra code added to each recipe's *context()* condition at the compilation stage. The code ensures that, when appropriate, the BDI program selects recipes based on plans returned by JSHOP, and at other times selects recipes using normal BDI recipe selection.

If at any point in the execution it is not possible to match a recipe from what JACK considers is available with what the planner considers should be executed, then this indicates that there is a problem, probably resulting from some environmental change. If at this stage execution continues, using the recipe chosen by the planner, this is likely to cause problems. JACK context conditions are written to ensure that appropriate plans for the situation are the ones that are considered. If a recipe is used which is intended for a different situation than the one existing, then it cannot be expected to succeed. If on the other hand we allow JACK to choose a recipe outside the plan which has been produced, we invalidate the plan.

In such cases, a recipe will not be selected, causing the goal it handles to fail, therefore causing the top level goal called within InvokePlanner (used as a generic term here to represent any plan that invokes JSHOP) to fail. When InvokePlanner realises the goal state has not been achieved, instead of calling the planner to replan, the InvokePlanner recipe will also fail. At this point the BDI system's failure handling will take over.

---

[4]Refer to http://www.cs.rmit.edu.au/˜desilva/research/publications for more information.

7   Conclusions

BDI systems are robust in dealing with complex and dynamic environments, and work with a recipe library provided by the programmer. In some situations it can be desirable to do some planning, either as a result of other approaches failing, or in order to look ahead to guide choices at a particular point. The planner would ideally be able to use information about the existing BDI program to simulate the behaviour of the system, and provide advice on the choices the system should take during execution. We have implemented a system that does this, by using an efficient HTN planner. Our focus is different to past work in interleaving planning and execution, in that we cater for the intrinsic needs of the BDI architecture. In particular, we leave the choice of when planning should be done, and with what information, to the BDI programmer. Executing the plan is done using regular BDI execution, using the advice from the planner on what recipes to choose, and what bindings to use in context conditions. Furthermore, our plan execution model is unique, in that it is possible for the BDI system to maintain control on plan failure, and resume normal BDI execution.

We are currently working on creating formalisms to define and evaluate our framework.

References

Bratman, M. E. 1987. Intention, Plans and Practical Reason, Havard University Press, Cambridge, MA, ISBN (Paperback): 1575861925.

de Silva, L. P., and Padgham, L. 2004. A Comparison of BDI Based Real-Time Reasoning and HTN Based Planning. In *17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia*.

Despouys, O., and Ingrand, F. F. 1999. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *European Conference on Planning (ECP)*, 278–293.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18(1):69–93.

Georgeff, M., and Ingrand, F. 1989. Decision making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Aritificial Intelligence (IJCAI)*, 972–978.

Levinson, R. 1995. A general programming language for unified planning and control. *Artifial Intellgence* 76(1-2):319–375.

Paolucci, M.; Shehory, O.; Sycara, K. P.; Kalp, D.; and Pannu, A. 1999. A Planning Component for RETSINA Agents. In *Agent Theories, Architectures, and Languages*, 147–161.

Veloso, M. M.; Pérez, M. A.; and Carbonell, J. G. 1991. Nonlinear Planning with Parallel Resource Allocation. In *Workshop on Innovative Approaches to Planning, Scheduling and Control*, 207–212.