Situational Preferences for BDI Plans

Lin Padgham
RMIT University
Melbourne, Australia
lin.padgham@rmit.edu.au

Dhirendra Singh
RMIT University
Melbourne, Australia
dhirendra.singh@rmit.edu.au

ABSTRACT

In BDI programs it is quite common to find context conditions of plans which are over-constrained in order to ensure that the most preferred plan is selected for use. This is undesirable for at least two reasons. It makes the plan not available for use at all in situations where it could be of value as a back-up plan, and also it requires incorporation of information that conceptually belongs with the preferred plan. The ability to specify directly in a plan specification, aspects of the situation which would make the plan more or less desirable, enables a dynamically calculated preference ordering which removes the need to over-constrain applicability to obtain the desired plan selection. This paper addresses the issue of dynamically assigning a value to a plan instance, based on the current state and the particulars of the plan instance under consideration. The framework uses specifications based on logical formulae which are evaluated dynamically, using the current state and variable bindings provided via the plan's context condition. These provide a simple mechanism for locally specifying the value of plan instances. This can be regarded as providing a degree of applicability for a plan, rather than simply a boolean value.

Categories and Subject Descriptors

I.2.11 [**Distributed AI**]: Intelligent Agents

Keywords

Preferences, BDI, Agent Programming Languages

1. INTRODUCTION

In BDI (Belief, Desire, Intention) agent systems such as JACK [15], JASON [4] or any of the AGENTSPEAK [10] like languages, the number of different ways of achieving a goal are typically represented as abstract procedures, or plans. Each plan has an associated programmed context condition, which is a boolean formula that specifies when that plan is considered to be suitable for use. One of the major strengths of these systems is the ability to specify when a plan is applicable, or valid for use, depending on the particular situation. However they do not provide a similar mechanism for the related issue of specifying how good a particular plan instance is in a specific situation, thus allowing us to select the plan which is preferred, or better, based on the current state.

In this paper we describe a straightforward extension for AGENTSPEAK like languages, that provides a declarative mechanism for capturing priorities or preferences that are based on the specific situation, and/or the specifics of the plan instance (as opposed to simply the plan type). The proposed extension provides support for declarative expressivity of localised preferences and the ability to reason over them which is a highly desirable feature in many domains. Existing AGENTSPEAK like languages already have the infrastucture to support this approach and can do so with minimal effort. We have implemented this proposal as an add-on library for the JADEX [9] agent programming langauge.

While AGENTSPEAK itself does not specify a plan selection mechanism, implemented systems often do provide some approach to specifying a preference ordering over the applicable plans. However this is typically either static, and does not take account of the specifics of either the situation, or the plan instance, or is a function, which is not transparent and does not provide structured information which can be reasoned over. We follow the approach typical in work on planning with preferences, of defining a language for specifying the preferences, and a function for aggregation of multiple preferences [1]. This establishes a framework where priorities are declaratively specified in terms of logical expressions which can be reasoned over. This also supports a mechanism whereby this information can be directly provided by domain experts.

One application that has motivated this work is participatory modelling for agent based simulations, where end-users are intimately involved in developing the behaviour models captured within the (agent based) simulation [6]. We have been exploring technological support for this process [12], whereby users run the simulation under different scenarios, stopping and commenting on an agent's plan choice at various points. Feedback here is typically of the form: "in this situation people would actually use plan X because ...". Currently such information must be processed by a programmer and incorporated into a procedure that does plan selection. This framework provides a declarative approach which facilitates both user understanding of the current specification, as well as direct modification of that specification. Moreover, the framework can be used to reason over the set of preferences collected over time, to highlight cases of interest to the user, such as when different preferences overlap or when one is implied by the other.

Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013), Ito, Jonker, Gini, and Shehory (eds.), May, 6–10, 2013, Saint Paul, Minnesota, USA.*
Copyright © 2013, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

It is quite common when looking at BDI agent programs, to see examples where context conditions are over constrained, as a way of ensuring the desired plan selection, in the absence of any straightforward mechanism for dynamically determining the value of a plan given the current situation. For example, assume there is a plan to fly, and a plan to take the train, for travel between two cities. Appropriate context conditions for the `fly` and `train` plan may well be

`flies($airline,$from,$to)`

and

`station($from)` $\wedge$ `station($to)`

as shown in Figure 1. It may in general be better to fly than to take the train[1], in which case this can be readily specified in most AgentSpeak like platforms, even though AgentSpeak itself does not provide for this within the language. For example both Jack [15] and Goal [7] use the order in which the plan/rule specifications appear in the code to determine priority between plan/rule types. However, assume we now also want to capture that taking the train is a good option if the distance between start and destination is less than 500 km. This requires evaluation of the current situation, namely `distance($from,$to)` which can be accessed from the beliefs. What is often seen in this kind of case is that the context condition of the plan to fly is written as

`flies($airline,$from,$to)`$\wedge$

`(distance($from,$to)` $>$ `500)`

in order to ensure that the train travel plan is selected in such situations. However this is distorting the real applicability of the fly plan, and necessitating an understanding of when it needs to be excluded from prioritisation because of the relative value of some other plan.

In addition, we may consider different instances of the `fly` plan to have different value, depending on the particular airline. We may wish to specify that flying with a 2-star airline has a low value, whereas flying with 4-star or above is valued higher. This requires us to be able to access the variables specific to the plan instance, and bound in the context condition, in order to assign the plan a value. For example, assume that both China Southern and Singapore Airlines fly between `$from` and `$to`, where China Southern is 2-star and Singapore Airlines is 5-star. An ability to assign dynamically a value based on a formula such as

`rating($airline)` $\leq$ `2` *and/or* `rating($airline)` $>$ `2`

allows this differentiation. In current implemented platforms, and formal language specifications, such differentiation is possible only through the use of procedural reasoning.

Our key requirements for plan priorities then are that their specification (*i*) be declarative, (*ii*) allow access to the current world state, (*iii*) provide access to plan instance variables, and (*iv*) be local in scope to the plan.

In effect, the preferences framework proposed here allows us to dynamically allocate a number to each applicable plan instance, which then provides a preference ordering that can be used by a plan selection function to choose the most preferred plan. This number can also be interpreted as a "degree of applicability", and while usually the plan selection mechanism would simply pick the most applicable plan, other reasoning is also possible. For instance, in some domains the cost of trying and failing a plan might be high. Here it may make sense to not select a plan at all if every applicable plan has a low ranking relative to some domain specific threshold. These considerations should give the reader an appreciation for how the framework can be used for plan selection. However our main focus here is on the description of the preferences framework, rather than on the different ways it can be applied in plan selection.

In the following section we first describe how plan selection is done in current BDI systems as well as some related work. We then describe in Section 3 our approach to specification of localised preference information which supports a more powerful approach to plan selection, without the need to use meta-plans or some equivalent where the conditions are hidden in the code. We explore the expressivity of the priority specification language we will use, and then in Section 4 discuss the need for aggregation of multiple preferences and define a couple of useful aggregation functions, while leaving it open for this to be provided by the developer using information specific to the domain.

2. PLAN SELECTION IN BDI

In many BDI languages the plan library is represented as a set of plan rules of the form: $e : \psi \leftarrow P$, where e is an event, ψ is the context condition, and P is the plan-body program. ψ is a boolean formula over the agents beliefs, and P is considered an *applicable* strategy for responding to event/goal e when condition ψ is believed to be true. Plan-body programs are abstract procedures that may contain further goals/events as well as actions.

The context condition ψ associated with a plan, allows the agent execution engine to limit the runtime selection to those plans which are designed to be appropriate for the current situation.[2]

Figure 1 shows a goal plan hierarchy that results from the set of plan rules shown, based on the example discussed earlier. As can be seen, each goal, e.g. `BookTransport`, typically has some number of associated plans, e.g., `fly` and `train`. At runtime, one of these plans is selected for execution. If at some point that plan fails, a BDI agent will generally re-evaluate and choose a different available plan, in order to continue trying to achieve the goal. If no applicable plans remain, the goal fails, thus failing the plan it is part of, e.g. `BookSelf`, leading to the process of finding a new plan one level higher up in the goal-plan tree, e.g. `UseTravelAgent`. It is this in-built failure recovery, common to many BDI platforms, that make it important not to exclude a plan from being applicable, because it is not the highest priority choice. In our example in Figure 1 discussed earlier, although it is better to take the train if the distance is less than 500 km, it may be that this plan fails, due to no available seats left. In this case we would prefer to try booking a flight if one is available. If, as is often done, the context condition of the `fly` plan includes

[1]This refers to generally preferred plans from a design point of view. This is not the same as user-specific preferences, which are not the focus of this framework.

[2]Plans that apply in every situation do not really warrant a context filter. Strictly, such plans have a context condition that always evaluates to true.

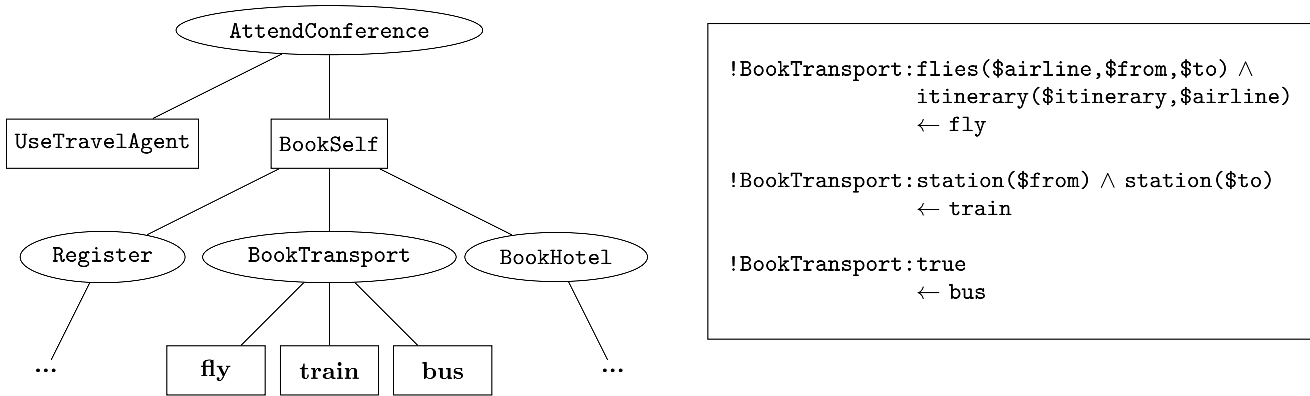

Figure 1: Example goal-plan hierarchy

the clause `distance($from,$to)` $>$ `500`, then this plan will not be applicable, and hence not available for use when the `train` plan fails.

As already mentioned, BDI programming languages such as AgentSpeak [10] or CANPLAN [11] are agnostic as to how a plan is selected from a set of applicable plans. In practice, implementation platforms have one or more mechanisms available for this selection. One common approach is to assume priority based on specification order, so in Figure 1, if the specification order was as shown, then the `fly` plan is prioritised higher than the `train` plan which is in turn higher than the `bus` plan. As noted this is a static prioritisation which does not take account of either the current situation, or of the particular plan instances available. In practice these implemented systems are often flexible enough that there is a mechanism whereby the desired selection can be accomplished. However this is hidden within a programmer defined function, for instance the `PlanChoice` function in Jack[3], often referred to as a meta-plan.

Meta-plans are powerful in that they can encode any desired reasoning. However, because they are not declarative, this information is not available for automated reasoning about the program and its structure. In the case of our tool support for participatory modelling for agent based simulation, where we wish to show users which plan an agent will select, and why, we have no access to the explanation why. The non-declarative nature of the meta-plan also makes it difficult to provide an interface whereby end users can interactively modify the selections made by the meta-plan, either to systematically explore the simulation, or to participate in the design of the agents.

There has been some work on incorporating preferences into plan selection in BDI systems. The work of [14] provides a mechanism for declaring global preferences in a slightly modified version of LPP [2]. These are then applied to plan selection, based on the effects of a plan, or on resource usage. It does not provide a mechanism for defining preferences based on the current situation, nor directly based on the characteristics of plan instances. Like much of the work on preferences in planning, it is oriented towards being able to capture and use the preferences of individual users, which we are not attempting to do in our approach.

Bordini et al. [3] presented extensions to the AgentSpeak language to allow for integration with the DTC scheduler for intention selection purposes. They introduced internal actions, or arbitrary user-provided functions that do not affect the environment, that could be used within a plan's context condition as well as in the plan-body program. Using specific internal actions, this mechanism was used for calculating plan priorities in a way understood by the DTC scheduler, for use in intention selection. This ability to call internal functions from inside context conditions could also be used to calculate plan instance priorities as we wish to do. However internal actions are black-box functions and do not support our agenda of declarative specification and reasoning.

Casali et al. [5] present an approach that extends BDI systems with graded mental attitudes (beliefs, desires and intentions), using modal logic. This enables a more nuanced version of having a goal, belief or intention, and probably this could be used to accomplish dynamic priorities of plans based on graded beliefs and goals. However it does not support a localised direct specification of plan value based on the current situation and plan instance variables.

Myers and Morley [8] are, like us, concerned with specifying preferences for agent strategies in specific situations, rather than the more common individual based preferences. Like us they identify a need to refer to the variables of a particular plan instance, which in their framework are called *roles*. They also introduce *features* that can be used to describe properties of a plan-type, such as "fast-heuristic" or "exact". However, although the properties of the plan are specified locally, the constraints that are used to reason about which plan to select, are specified non-locally. Also, their strategy preference mechanisms are embedded within a larger context of a framework for adjustable autonomy.

Work on preferences in HTN planning [13] is potentially relevant in that HTN planning is very similar to BDI execution [11]. However, the approach here is to determine preferences associated with an entire plan, as opposed to preferences with respect to a single choice point. This reflects the philosophical difference in approach between planning,

[3] `http://www.aosgrp.com/documentation/jack/Agent_Manual_WEB/index.html`

which determines a full course of actions, and BDI execution, which makes decisions as to how to act at each step, allowing the executed plan to unfold depending on potential environmental changes.

Our approach is to assign a quantitative value to each plan instance, using a local preference specification which allows dynamic calculation of this value based on both the current situation and the attributes of the particular plan instance.

3. REPRESENTING THE PREFERENCES

In order to facilitate the calculation of a plan value which can be obtained dynamically based on a specific situation and an individual plan instance, we extend the notation for our plan rule as follows:

$$e : \psi \leftarrow P[\theta]$$

where e is an event, ψ is the context condition, P is the plan body and θ is a *preference description* comprising the tuple $< d, \mathcal{V}, \lambda >$ defined as follows:

d is an integer giving a *default preference measure*;

$\mathcal{V}$ is a set of *value specification* pairs $\{(\varphi_1, \mathcal{V}(\varphi_1)), (\varphi_2, \mathcal{V}(\varphi_2)), \ldots, (\varphi_n, \mathcal{V}(\varphi_n))\}$, where φ is a *preference formula* in the language of first order predicate logic without quantification and $\mathcal{V} : \mathcal{S} \rightarrow \mathbb{R}$ defines a *preference measure* for the set $\mathcal{S}$ of formulae $\varphi \in \mathcal{S}$.

λ is an *aggregation function* $\lambda : [d, \mathcal{V}, \mathcal{S}'] \rightarrow \mathbb{R}$ which specifies how to aggregate the default preference measure d and the preference measures of the evaluated set $\mathcal{S}' = \{\varphi | \varphi \in \mathcal{S}, \forall \varphi = true\}$ of preference formulae that hold true in a given situation.

The plan value is then obtained using λ if both λ and $\mathcal{S}'$ are non-empty, and by using d otherwise.

The preference formulae can access any variables bound within that plan's context condition, or available within the agent's beliefs (essentially any variables accessible within the static scope of the plan definition, if the platform used has scoping mechanisms). We require that the preference formulae in $\mathcal{S}$ be *non-redundant*, i.e., for any two formulae φ_1 and φ_2, that $\neg[(\varphi_1 \models \varphi_2) \wedge (\varphi_2 \models \varphi_1)]$. For example, a non-redundant set $\mathcal{S}$ may only contain one of the formulae $A \wedge B$ or $B \wedge A$, but not both, since they entail each other. Overlapping preference formulae are however allowed, so it is possible to have $A \wedge B \wedge C \in \mathcal{S}$ and $A \wedge B \in \mathcal{S}$ with the same or different preference measures. The aggregation function λ then defines how this will be understood.

The preference formulae can be used both for specifying cases where particular plans are good, but also cases where particular plans are a poor choice. Using this approach our `fly` and `train` plans can now be specified as shown in Figure 2. We see that the `fly` plan has a default value of 5, but if the rating of the airline is `< 2`, then it has a preference measure of 2, and if the rating is `> 4` then it has a preference measure of 7. Moreover, when warnings of volcanic ash are current, the preference for flying is very low or `-10`. Here higher numbers represent higher preference. Similarly the `train` plan has a default value of `3`, with a preference measure of `8` if the distance from the source to destination is less than 500 km.

```
!BookTransport:flies($airline,$from,$to)
        ← fly
        [5,
        {(rating($airline) < 2, 2)
        (rating($airline) > 4, 7),
        (warning(volcanic-ash), -10)},
        λ]

!BookTransport:station($from) ∧ station($to)
        ← train
        [3,
        {(distance($from,$to) < 500, 8)},
        λ]
```

Figure 2: Examples of preference descriptions

Looking at the `fly` plan we can see that at some point we may find ourselves in the situation where we have a plan to fly with some airlines which has a rating of `5`, and a current belief `warning(volcanic-ash)`, leading to preference measure evaluations of `7` and `-10` respectively. Our aggregation function must now determine how this is to be interpreted. There are a number of intuitively reasonable options in a case with multiple true preference formulae, including taking the maximum or the sum. We will explore the aggregation function further in Section 4.

3.1 Expressivity of Value Specifications

Having introduced the framework for locally specifying plan values, we now focus on the preference formulae contained in the set $\mathcal{S}$, and the kinds of things that can and cannot be captured in such formulae. Essentially, preference formulae can specify combinations of the following types of information:

- beliefs about the current state of the world, and
- the bound variables for the particular plan instance.

For instance in Figure 2, in plan `fly` the value specification

```
(warning(volcanic-ash), -10)
```

is an example where the current world state influences the priority of the plan. Here, if the belief representing a current warning about volcanic ash holds true, then we want to assign a very low priority of `-10` to this plan instance, as in such situations we deem flying to be risky as well as prone to delays.

The specification

```
(distance($from,$to) < 500, 8)
```

associated with the `train` plan says that train travel is valued more (i.e., `8` instead of the default `3`) for distances shorter than 500 km, where distance is calculated using the values bound to the `train` plan variables `$from` and `$to`. Similarly the specification

```
(rating($airline) > 4, 7)
```

for the `fly` plan says that if the rating of the airline bound to the variable `$airline` is higher than 4-star, then the plan

receives a fairly high preference measure of 7 compared to the default of 5. It is this type of specification that is able to access the variables of the plan instance and differentiate between instances of the same plan type, that can provide a benefit over more static priority mechanisms available in current agent systems.

Note that in some situations, more than a single preference formula in the set $\mathcal{S}$ will hold true, this subset being $\mathcal{S}'$, in which case the final single value of the plan instance must somehow be decided by considering the individual preference values associated with each formula in $\mathcal{S}'$. As mentioned before, this is done by the provided aggregation function λ that we discuss in the following section.

We now turn our attention to the kinds of things that would potentially be useful but cannot be expressed, or are difficult to represent, in our notation. Consider again the specification

`(rating($airline) > 4, 7)`

from earlier that gives a higher than default measure to higher rated airlines. In this case it may in fact be that what the developer really wishes to capture is that "the higher the airline rating the more preferred the plan instance is." However, this kind of description cannot in general be specified in our framework that assigns discrete values to logical formulae evaluations. In order to capture such a preference, we would need to approximate it by specifying discrete ranges and providing separate preference descriptions for each range. This is clumsy and is also less straightforward for modellers to understand than a single statement. Overall, this kind of comparison requires reasoning across multiple plan instances and not just a single instance as we do here, and is more appropriate at a higher level where this information is available.

Another potential issue is that our framework does not support different plan values being calculated depending on the predecessor (goal or plan) of the goal to which the plan instance is responding. For example in Figure 1 the `fly` plan is in the context of the `BookSelf` plan, which is one of the plans available for realising the goal `AttendConference`. However, the `BookTransport` goal (which is what causes the `fly` plan to potentially be evaluated) may also be used within some other plan within the same agent, where different preference criteria might apply. For example we could imagine a goal `ArrangeHoliday`, which somewhere in its goal-plan tree contains a plan, say `MakeBookings` which also invokes the sub-goal `BookTransport` but would evaluate the various available plan instances differently than in the context shown in Figure 1. In general, to address this issue we would need a language to express the path of plan choices and resulting goals, in the execution so far, of a given intention, and this we do not provide. It would of course be possible to capture this kind of information in the form of beliefs, which could then be referenced using our framework.

A somewhat related issue is that preference descriptions cannot depend on information about later sub-goals to be accomplished and their available plans. For example, the preference to fly with a higher rated airline (that is presumably more expensive) may apply only so long as there are sufficient funds remaining for successfully accomplishing `BookHotel` afterwards. This would require the full (potential) executions to be assessed and then preferences evaluated in light of this information. However this involves planning and does not accord with the BDI approach of making a decision at each step, based on current information. Both these last two limitations are essentially to do with our requirement to specify preferences locally. They also however follow from the modular design philosophy of BDI plan hierarchies that sub-goals and sub-plans should be independent of any higher-level plans in which they are combined.

4. PREFERENCE AGGREGATION

The role of the aggregation function $\lambda(d, \mathcal{V}, \mathcal{S}')$ is to take the set $\mathcal{S}' \subseteq \mathcal{S}$ of preference formulae which are true for the particular plan instance, and use these to determine a single plan value given d and $\mathcal{V}$, which will be compared with the plan values of other applicable plans to select the most preferred one. The preference specifications, and the way in which they are aggregated will determine the preference ordering between all plan instances of the given type. For a full understanding of the preference ordering for the situation it is necessary to compare with the numerical values of other plan types for the same sub-goal. Although in general explicit ordering is very cumbersome, it is often the simplest approach when dealing with quite small numbers. The number of distinct plan types is indeed usually very small, seldom more than five. However numbers of instances can be large, and it is here that our preference specification must correctly order different instances of the same type.

In cases where each formula is essentially capturing a set of independent attributes, i.e., the attributes are *preferentially independent* of each other, and giving them a measure of worth depending on the actual value of the attribute, then it makes sense to have the aggregation function simply sum the measures of the preference descriptions in $\mathcal{S}'$. For example if our `fly` plan has the following set of preference descriptions:

```
(rating($airline) > 3, 2)
(stops($itinerary) = 0, 4)
(warning(volcanic-ash), -10)
```

then it may well make sense to sum the measures associated with the preference descriptions in $\mathcal{S}'$. More generally, summation can make sense also when attributes within a formula are not preferentially independent, as long as the formulae themselves are independent of each other, that is, the conditions captured by them do not overlap.

However, even if the predicates representing the attributes are independent, it is likely, given the language we have specified that we will end up with sentences which are not independent. Take for instance the addition of

`(stops($itinerary) < 2, 2)`

to our set of descriptions above. Now, in a particular plan instance with a bound itinerary, we may have 0 stops. In this case, the formula in both

`(stops($itinerary) < 2, 2)` *and*
`(stops($itinerary) = 0, 4)`

will evaluate to true and be included in $\mathcal{S}'$, but it does not really make sense to add these two values. Of course the values could have been set such that adding them would give the desired answer, for instance by changing the value of `stops($itinerary) = 0` to 2, so that it would sum to 4 when both formulae evalute to true. We could also write the

formulae to be exclusive, e.g.,

`(stops($itinerary) > 0 ∧ stops($itinerary) < 2, 2)`.

However, in order to remove this burden on the designer, we instead use a definition of λ that sums only over the preference descriptions in $\mathcal{S}'$ where the formula is not entailed by the formula of some other preference description in $\mathcal{S}'$. We call this aggregation function *ESum*[4] and define it as:

$$\lambda(d, \mathcal{V}, \mathcal{S}') = \sum_{\varphi \in \mathcal{S}', \forall \varphi' \in \mathcal{S}' \setminus \{\varphi\} : \varphi' \not\models \varphi} \mathcal{V}(\varphi) \quad (1)$$

This kind of reasoning over the formulae highlights one of the key benefits of using a declarative language to capture preferences. Logical reasoning could also be used to ensure consistency such as in systems where preferences are collected over time. For instance, reasoning may be used to identify overlaps in the conditions captured by the formulae in $\mathcal{S}$ and confirm if this was intended or should be corrected. On the other hand, the system can suggest subsets of formulae that could be further consolidated into more compact forms.

In some cases attributes or predicates are not independent, and the preferred value of one attribute is dependent on the value of another, called *conditional preference.* Take the case where we want the agent to prefer flight times under `14` hours regardless of the number of stops. Here the preference for flight-time is preferentially independent of the number of stops. However if the preference is such, that for flights under `14` hours it is best to have `0` stops, but if it is over `14` hours it is better to have `1` stop to stretch the legs, then the preference for stops would be conditional on flight-time. In this case the preference specifications cannot simply be given in terms of simple attributes with value ranges and appropriately chosen preference measures as before. So while we can certainly capture the preference for flight-times as

```
(flight-time($itinerary) < 14, 5)
(flight-time($itinerary) ≥ 14, -2)
```

there is no way then to set preference measures for the number of stops, i.e., for the formulae `stops($itinerary) = 0` and `stops($itinerary) ≥ 0`, in a way that gives the desired ordering of the plan instances with any straightforward aggregation function. However, we can achieve this desired ordering by explicitly capturing the conditional dependence that matters, such as:

```
((flight-time($itinerary) < 14) ∧
 (stops($itinerary) = 0), 6)
((flight-time($itinerary) < 14) ∧
 (stops($itinerary) = 1), 4)
((flight-time($itinerary) ≥ 14) ∧
 (stops($itinerary) = 1), 3)
((flight-time($itinerary) ≥ 14) ∧
 (stops($itinerary) = 0), 2)
```

While this is somewhat cumbersome, the interacting attributes within a single plan are likely to be few, and explicit ordering is straightforward to understand and to specify. Note that while the attributes stops and flight-time are not mutually independent, the formulae above are mutually exclusive and therefore if the other preference specifications in $\mathcal{S}$ are independent of these too, then it may make sense to use the ESum aggregation function. As previously it is appropriate to exclude descriptions with sentences that are entailed by other sentences in $\mathcal{S}'$. So for instance if the specification

```
(flight-time($itinerary) < 14, 5)
```

was added to the above set of preference descriptions (in case stops is unknown), one would not want to add its preference measure to that of the first or second descriptions above.

It is also well known that in many cases attributes are not additive. Indeed, in our simulation work with social science colleagues we have frequently experienced situations where they have wanted to capture the fact that the value of `A` ∧ `B` is not the same as the value of `A` plus the value of `B`. This third case is also straightforward for us to express in our framework, as for example

```
(rating($airline) > 3, 5)
(stops($itinerary) = 0, 3)
((rating($airline) > 3) ∧
 (stops($itinerary) = 0), 10)
```

where we define the value of `0` stops together with a high airline rating as `10`, rather than simply the sum of the two parts, which would give `8`. As before, our aggregation function ESum as defined will exclude the more general, entailed descriptions (the first two in the list) from $\mathcal{S}'$ before summing, allowing this more specfic (third) description to take precedence.

We have discussed cases where one attribute may be conditionally dependent on another (e.g. stops being conditionally dependent on flight-time). However there are also cases where the attributes are not directly dependent, but are obviously correlated. Let us assume that we sometimes have access to further information about an airline, in the form of its food quality and seat comfort rating, and that we have the following preference specifications:

```
(rating($airline) > 3, 4)
(food-quality($airline) > 3, 5)
(seat-comfort($airline) > 3, 6)
```

It is clear from this specification that seat-comfort is more highly valued than food-quality or overall rating. Now consider the case where we have two plan instances: one for Singapore Airlines which is rated 5-star overall, 4-star for food, but 3-star for seat comfort, and another for Indian Airlines which is rated 3-star overall, 3-star for food, and 4-star for seat comfort. If we use ESum the calculated values will be `9` for Singapore Airlines, and `6` for Indian Airlines, leading to Singapore Airlines being preferred, even though its seat-comfort is lower. While this may be the desired interpretation, it is also possible that what is intended is that the formulae capture potentially overlapping information, and that the best overall situation be the determinant for the plan's value.

For such cases we define a new aggregation function called *EMax* that assigns a plan value based on the maximum measure in $\mathcal{S}'$:

[4] for "Excluding Sum" as we exclude descriptions where the formulae are implied by other descriptions.

$$\lambda(d, \mathcal{V}, \mathcal{S}') = \mathbf{max}(\mathcal{V}(\varphi) \mid \varphi \in \mathcal{S}', \forall \varphi' \in \mathcal{S}' \setminus \{\varphi\} : \varphi' \not\models \varphi) \quad (2)$$

We note that, should we wish to do so, we can still capture the combination of good rating and good food quality and give it a value higher than seat-comfort, by simply combining into a single formula. Overall though a decision must be made as to whether the intended interpretation is to give points to particular conditions and then sum them, or to recognise particular conditions and choose the highest valued.

Finally, a slight variation on our aggregation functions is where the individual preference descriptions are really describing situations which make the plan slightly better or worse than the default value. In this case it may make more sense to interpret the measures as offsets from the default. This makes no difference to the preference ordering among the plan instances of the single type, but does affect the calculated plan value, and therefore the preference ordering of the plan in relation to plans of other types. This *Offset* aggregation function for ESum is defined below (the change to EMax is similar):

$$\lambda(d, \mathcal{V}, \mathcal{S}') = d + \sum_{\varphi \in \mathcal{S}', \forall \varphi' \in \mathcal{S}' \setminus \{\varphi\} : \varphi' \not\models \varphi} \mathcal{V}(\varphi) \quad (3)$$

The Offset versions of the aggregation functions have the benefit that they allow us to conceptualise values in relative terms rather than absolute terms. This is arguably more intuitive as the baseline default values can be correlated across all plan types that handle the same event. As mentioned, the final ordering of plan instances of a single type is not impacted.

To summarise the discussion above, our preference framework caters to the following cases:

- Independent attributes: this is straightforward and is done by giving preference measures to the attributes with their particular values.
- Conditionally dependent attributes: this is done by explicitly specifying the combinations that we care about, and giving a preference measure to each such combination.
- Non-additive attributes where the preference value of two attributes (A $\wedge$ B) together is not the same as the value of A plus the value of B separately: this is achieved by logical entailment so that the more specific formulation, i.e., (A $\wedge$ B) takes precedence.
- Non-independent attributes in the sense that they capture overlapping things: this is easy enough to specify in the formulae but needs to be aggregated differently, such as by taking the most preferred condition rather than summing over all conditions that hold.

Our framework also allows for multiple aggregation functions to be used in the agent program, since λ is included in the preference specification θ attached to each plan type. This means that a different aggregation function could be used per plan type if needed. In general, we believe that a choice between ESumOffset and EMaxOffset is sufficient for applications (*i*) when the same aggregation function is to be used for the entire program; (*ii*) when all plan types for a given event type have the same aggregation function; and (*iii*) when a different aggregation function is required per plan type.

5. DISCUSSION AND CONCLUSION

The approach that we have presented allows a straightforward declarative specification of the value of a plan, in a way that can be dynamically calculated depending on the particular state of the environment, as well as the information about the individual plan instance under consideration.

This eliminates the need to use over-constrained context conditions of plans to enable preferred plan selection of a different plan type in some situation. With our framework, we no longer need to have the context condition of our `fly` plan include the clause

`distance($from,$to)` $>$ `500`

in order to ensure that the usually less preferred `train` plan is chosen for shorter distances. Instead we simply add a preference description something like:

`(distance($from,$to)` $<$ `500, 10)`

to the `train` plan, where `10` is sufficient to give the plan a value above that for the `fly` plan in this situation.

The declarative specification at the plan level also makes it straightforward to derive an explanation for a user as to why a particular plan was chosen in a particular situation. This framework supports the development of an interface to allow non-programmers to question and understand the plan choices being made by the agent system, and indeed to provide input into what those plan choices should be.

The aggregation functions defined, where more general specifications are ignored in favour of more specific ones when both are true, provide ease and flexibility in writing preference specifications as it is necessary only to consider whether there are dependencies regarding the combinations of attributes, not whether there are dependencies involving the actual formulae.

As far as the authors are aware this is the first approach to provide a declarative specification to enable plan selection based on the current beliefs and the bindings of the particular plan instance.

A possible weakness of the approach is that the ordering is determined by explicit plan values, and therefore the developer must ensure that these values give the desired ordering. In general this approach is undesirable largely because it is difficult to both assign appropriate values and maintain an overview of the values assigned. However, we would argue that in this particular situation the approach is acceptable as the number of plan types will almost certainly be small, allowing a straightforward overview of relative values. Within a single plan type, the number of instances may be large, but we need only ensure that the preference descriptions are correctly ordered. These will also likely be a limited number, to address some specific situations.

The biggest weakness of the framework presented is that we are unable to order all instances based on one or more preferentially independent variables. However this would require reasoning across multiple plan instances, whereas our current framework assesses only one instance at a time. Such an addition would require a language that allowed comparisons across different plan instances that did not rely on the simple assignment of numerical values that we have now.

One possibility may be to develop an appropriate preference specification framework at the level of goals. This is a possible area of interest for future work. However we first want to evaluate the current approach with users who are not programmers, to establish (*i*) whether it is easy for them to specify domain knowledge in this way, and (*ii*) whether it is easy for them to understand what is specified in a given program. Of course it is still possible to achieve such an ordering using a meta-plan or other procedural mechanism, but it is not declarative.

We believe that the ability to dynamically assess the value of particular plan instances, dependent on current situation and the specifics of the individual plan is extremely important. Making this a part of the declarative specification of the plan, similarly to the context condition, is a straightforward but significant step in improved flexibility of BDI programming languages.

Implementation

We have developed a basic implementation of the proposed extension as an add-on Java library (.jar file) for use with the JADEX[5] agent programming platform. The extension is available for download, together with usage instructions and a working example used in this paper, from: `https://sites.google.com/site/rmitagents/software`.

6. ACKNOWLEDGMENTS

This work is partially supported by the Australian Research Council under Discovery grant DP1093290 and the National Climate Change Adaptation Research Facility under grant EM1105. We also thank Andreas Suekto, Ralph Rönnquist, Sebastian Sardina, Sheila McIlraith, and Sarah Hickmott for their valuable input to this work.

7. REFERENCES

[1] J. A. Baier and S. A. McIlraith. Planning with preferences. *AI Magazine*, 29(4):25–36, 2008.

[2] M. Bienvenu, C. Fritz, and S. A. McIlraith. Planning with qualitative temporal preferences. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 134–144. AAAI Press, 2006.

[3] R. H. Bordini, A. L. C. Bazzan, R. de Oliveira Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): efficient intention selection in bdi agents via decision-theoretic task scheduling. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1294–1302, 2002.

[4] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason.* Wiley, 2007. Series in Agent Technology.

[5] A. Casali, L. Godo, and C. Sierra. A graded BDI agent model to represent and reason about preferences. *Artificial Intelligence*, 175(7-8):1468 – 1478, 2011.

[6] T.-Q. Chu, A. Drogoul, A. Boucher, and J.-D. Jucker. Towards a methodology for the participatory design of agent-based models. In N. Desai, A. Liu, and M. Winikoff, editors, *Principles and Practice of Multi-Agent Systems*, volume 7057 of *Lecture Notes in Computer Science*, pages 428–442. Springer, 2012.

[7] K. V. Hindriks. Programming rational agents in GOAL. *Multi-Agent Tools: Languages, Platforms and Applications*, pages 119–157, 2009.

[8] K. Myers and D. N. Morley. Policy-based agent directability. In H. Hexmoor, C. Castelfranchi, and R. Falcone, editors, *Agent Autonomy*, chapter 9, pages 187–210. Kluwer Academic Publishers, 2003.

[9] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A bdi reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer, 2005.

[10] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55. Springer Verlag, Jan. 1996. LNAI, Volume 1038.

[11] S. Sardiña and L. Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.

[12] D. Scerri, S. Hickmott, and L. Padgham. User understanding of cognitive processes in simulation: A tool for exploring and modifying. In *Winter Simulation Conference (WSC)*, Berlin, Germany, Dec. 2012.

[13] S. Sohrabi, J. A. Baier, and S. A. McIlraith. HTN planning with preferences. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1790–1797, 2009.

[14] S. Visser, J. Thangarajah, and J. Harland. Reasoning about preferences in intelligent agent systems. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 426–431, Barcelona,Spain, July 2011. IJCAI/AAAI.

[15] M. Winikoff. Jack intelligent agents: An industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer, 2005.

[5] `jadex-agents.informatik.uni-hamburg.de`