Leask, Sam and Logan, Brian (2015) Programming deliberation strategies in meta-APL. In: PRIMA 2015, 26-30 Oct 2015, University Residential Centre of Bertinoro (Ce.U.B.), Italy.

Programming Deliberation Strategies in Meta-APL

Sam Leask and Brian Logan

School of Computer Science
University of Nottingham
{svl,bsl}@cs.nott.ac.uk

Abstract. A key advantage of BDI-based agent programming is that agents can deliberate about which course of action to adopt to achieve a goal or respond to an event. However, while state-of-the-art BDI-based agent programming languages provide flexible support for expressing plans, they are typically limited to a single, hard-coded, deliberation strategy (perhaps with some parameterisation) for all task environments. In this paper, we present an alternative approach. We show how both agent programs and the agent's deliberation strategy can be encoded in the agent programming language meta-APL. Key steps in the execution cycle of meta-APL are *reflected* in the state of the agent and can be queried and updated by meta-APL rules, allowing BDI deliberation strategies to be programmed with ease. To illustrate the flexibility of meta-APL, we show how three typical BDI deliberation strategies can be programmed using meta-APL rules. We then show how meta-APL can used to program a novel *adaptive* deliberation strategy that avoids interference between intentions.

1 Introduction

The BDI approach to agent programming has been very successful, and is perhaps now the dominant paradigm in agent language design [8]. In the BDI approach, agents select plans in response to changes in their environment, or to achieve goals. In most BDI-based agent programming languages, plan selection follows four steps. First the set of relevant plans is determined. A plan is relevant if its triggering condition matches a goal to be achieved or a change in the agent's beliefs the agent should respond to. Second, the set of applicable plans are determined. A plan is applicable if its belief context evaluates to true, given the agent's current beliefs. Third, the agent commits to (intends) one or more of its relevant, applicable plans. Finally, from this updated set of intentions, the agent then selects one or more intentions, and executes one (or more) steps of the plan for that intention. This deliberation process then repeats at the next cycle of agent execution.

Current APLs provide considerable syntactic support for steps one and two (determining relevant applicable plans). However, with the exception of some flags, the third and fourth steps can not be programmed in the APL itself. No

single deliberation strategy is clearly 'best' for all agent task environments. It is therefore important that the agent developer has the freedom to adopt the strategy which is most appropriate to a particular problem.

Some languages allow the programmer to over-ride the default deliberation cycle behaviour by redefining 'selection functions' in the host language (the language in which the APL is itself implemented), e.g., [2], or by specifying the deliberation strategy in a different language, e.g., [10]. Clearly, this is less than ideal. If often requires considerable knowledge of how the deliberation cycle is implemented in the host language, for example. Moreover, without reading additional code (usually written in a different language), an agent developer cannot tell how a program will be executed.

An alternative approach is to use procedural reflection. A reflective programming language [11] incorporates a model of (aspects of) the language's implementation and state of program execution in the language itself, and provides facilities to manipulate this representation. Critically, changes in the underlying implementation are reflected in the model, and manipulation of the representation by a program results in changes in the underlying implementation and execution of the program. Perhaps the best known reflective programming language is 3-Lisp [12]. However, many agent programming languages also provide some degree of support for procedural reflection. For example, the Procedural Reasoning System (PRS) [9] incorporated a meta-level, including reflection of some aspects of the state of the execution of the agent such as the set of applicable plans, allowing a developer to program deliberation about the choice of plans in the language itself. Similarly, languages such as *Jason* [2] provide facilities to manipulate the set of intentions. However the support for procedural reflection in current state-of-the-art agent programming languages is often partial, in the sense that it is difficult to express the deliberation strategy of the agent directly in the agent programming language.

In this paper, we show how procedural reflection in the agent programming language meta-APL [7] can be used to allow a straightforward implementation of steps three and four in the deliberation cycle of a BDI agent, by allowing both agent programs and the agent's deliberation strategy to be encoded in the same programming language. By exploiting procedural reflection an agent programmer can customise the deliberation cycle to control which relevant applicable plan(s) to intend, and which intention(s) to execute. To illustrate the flexibility of meta-APL, we show how three typical BDI deliberation strategies can be programmed using meta-APL rules. We then show how meta-APL can used to program a novel *adaptive* deliberation strategy that avoids interference between intentions.

2 Specifying Deliberation Strategies

Many deliberation strategies are possible and it is impossible to consider them all in detail. Instead we focus on three deliberation strategies that are representative of deliberation strategies found in the literature and in current implementations of BDI-based agent programming languages. The strategies are based on those

presented in [1], however the terminology has been changed to be more consistent with usage in the BDI literature (e.g., 'selecting a planning goal rule' becomes 'selecting a plan' etc.), and the restriction in [1] that plans don't have subgoals has been removed.

The simplest deliberation strategy is a *non-interleaved* **(ni)** deliberation strategy that executes a single intention to completion before adopting another intention. Alternatively, the agent may pursue multiple goals in parallel, choosing and executing plans for several (sub)goals at the same time. In [1] two parallel strategies are described: the *alternating (single action)* **(as)** strategy, and the *alternating (multi action)* **(am)** strategy. The **(as)** strategy first selects a plan for an event (e.g., a belief update or (sub)goal) and then executes a single basic action of one of the agent's current intentitons. One common instantiation of the **(as)** strategy is *round robin* scheduling (RR), in which the agent executes one step of each intention at successive deliberation cycles. (RR) is the default deliberation strategy used by *Jason* [2]. The **(am)** strategy first selects a plan for an event and then executes a single basic action from each of the agent's current current intentions (i.e., it executes multiple actions per deliberation cycle). **(am)** is the deliberation strategy used by 3APL [4].

However none of these strategies (or any other single strategy) is clearly "best" for all agent task environments. For example, the **(ni)** strategy has the advantage that it minimises conflicts between intentions. If the preconditions of plans for each top-level goal are disjoint, and the preconditions for all subplans (and actions) are established by preceding steps in the intention, then, if a set of intentions can be executed at all in a given environment, they can be executed by a **(ni)** strategy. However it has the disadvantage that the agent is unable to respond to new goals until (at least) the intention for the current goal has been executed. In many situations it is desirable that the agent progresses all its intentions at approximately the same rate (i.e., achieves its top-level goals in parallel). For example, if goals correspond to requests from users, we may wish that no user's request is significantly delayed compared to those of other users. Conversely, the **(as)** and **(am)** strategies allow an agent to pursue multiple goals at the same time, e.g., allowing an agent to respond to an urgent, short-duration task while engaged in a long-term task. For example, round robin scheduling attempts to ensure 'fairness' between intentions by executing a one step of each intention in turn at each deliberation cycle. However these strategies can increase the risk that actions in plans in different intentions will interfere with each other. Such conflicts between intentions can sometimes be avoided by using *atomic* constructs that prevent the interleaving of actions in a plan in one intention with actions from plans in other intentions. However, it is difficult for the programmer to ensure that all potential interactions between plan steps are encapsulated within atomic constructs, and excessive use of atomic constructs may reduce the responsiveness of the agent to an unacceptable degree.[1] It is therefore important that the agent developer has the freedom to choose the strategy that is most appropriate to a particular problem.

[1] We return to the problem of interference in Section 5.

3 Meta-APL

Meta-APL is a BDI-based agent programming language in which a programmer, in addition to being able to write normal agent programs, can also specify the deliberation cycle. This is achieved by adding to the language the ability to query the agent's plan state and actions which manipulate the plan state.

There are two key goals underlying the design of meta-APL:

- it should be possible to specify a wide range of deliberation cycles, e.g., the deliberation cycles of current, state-of-the art agent programming languages
- it should be simple and easy to use, e.g., it should be easy to specify alternative deliberation strategies

The ability to express deliberation strategies (and other language features) in a clear, transparent and modular way is a flexible tool for agent design. By expressing a deliberation strategy in meta-APL, we provide a precise, declarative operational semantics for the strategy which does not rely on user-specified functions. Even low level implementation details of a strategy, such as the order in which rules are fired or intentions are executed, can be expressed if necessary.

In this section, we briefly introduce meta-APL [7].[2] A meta-APL agent consists of an *agent program* and the *agent state* which is queried and manipulated by the program. The agent program consists of an ordered sequence of sets of rules. The agent's state consists of two main components: the *mental state*, which is a collection of *atom instances*, and the *plan state* which consists of a collection of *plan instances* and their properties. Atom instances are used to represent beliefs, goals, events etc. Plan instances play a role similar to relevant, applicable plans in conventional BDI agent programming languages.

3.1 Meta-APL Syntax

The syntax of Meta-APL is built from atoms, plans, clauses, macros, object rules, and meta-rules, and a small number of primitive operations for querying and updating the mental state and plan state of an agent.

Atoms Atoms are built of terms. Terms are defined using the following disjoint sets of symbols: *IDs* which is a non-empty totally ordered set of ids, *Pred* which is a non-empty set of predicate symbols, *Func* which is a non-empty set of function symbols, and *Vars* which is a non-empty set of variables.

[2] A preliminary version of meta-APL was presented in [6]. The main differences between the verson of meta-APL presented in [6] and that presented here, are that the belief and goal bases have been merged into a single 'mental state', plan instances are automatically deleted if any of the atoms forming the *justification* for the plan (see below) are deleted, and plan instances must be explicitly scheduled for execution (in [6] a single step of the root plan in each intention was executed at each cycle).

The syntax of terms t and atoms a is given by:

$$t =_{def} x \mid f(t_1, \ldots, t_m)$$
$$a =_{def} p(t_1, \ldots, t_n)$$

where $f \in Func$,[3] $p \in Pred$, $x \in Vars \cup IDs$, $n \geq 0$, and $m \geq 0$. For example, a domestic robot (cf. [2]) may represent a belief that its supply of beer has been depleted as:

$$\text{belief}(\text{stock}(\text{beer}, 0))$$

To distinguish between different instances of syntactically identical atoms (e.g., two instances of the same event), each atom instance is associated with a unique $id \in IDs$.

The atom instances comprising the agent's mental state can be queried and updated using the following primitive operations:

- $\mathsf{atom}(i, a)$: an instance of the atom a has id i
- $\mathsf{add\text{-}atom}(i, a)$: create a new instance of the atom a and bind its id to i
- $\mathsf{delete\text{-}atom}(i)$: delete the atom instance with id i

For brevity, queries may be expressed in terms of atoms rather than atom instances where the id is not important, i.e., the query a is true if the query $\mathsf{atom}(\_, a)$ is true.

Plans A *plan* is a textual representation of a sequence of actions the agent can execute in order to change its environment or its mental state. Plans are built of external actions, mental state tests, reified mental state actions and subgoals composed with the sequence operator ';'. A plan π is defined as:

$$\pi =_{def} \epsilon \mid (ea \mid mt \mid ma \mid sg)\,;\pi$$

where ϵ denotes the empty plan, ea is an external action of the form $\mathrm{e}(t_1, \ldots, t_n)$, $\mathrm{e} \in ActionNames$ and $t_1, \ldots t_n$, $n \geq 0$ are ground terms, mt is a mental state test of the form $?\,q$ where q is a (primitive or defined) mental state query, ma is a (primitive or defined) mental state action, and sg is a subgoal of the form $!\,g(u_1, \ldots, u_m)$ where $g(u_1, \ldots, u_m)$ is an atom and $u_1, \ldots u_m$, $m \geq 0$ are (possibly non-ground) terms. For example, the domestic robot may employ the following plan to scold its owner:

!at(robot, owner);
say("You should drink no more than 3 units of alcohol per day!")

Meta-APL distinguishes between generic plans, which are a static part of the agent program, and plan instances — specific substitutions of generic plans generated during the execution of the program. The plan state of a meta-APL agent may contain multiple instances of the same plan (e.g., if a plan is used to achieve

[3] In addition to standard functors, we assume Prolog-style list syntax.

different subgoals). Each plan instance has a unique *id*, a current *suffix* (the part of the instance still to be executed), one or more justifications, a substitution and (at most) one active subgoal. A *justification* is an atom instance *id*. Informally a justification is a 'reason' for executing (this instance of) the plan, e.g., an atom representing a belief or goal. In general, a plan instance may have multiple justifications, and a justification may be the reason for adopting multiple plan instances. The *substitution* $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$ specifies the current binding of variables in the plan instance to terms. A *subgoal* is created by the execution of a subgoal step $!\,g(u_1, \ldots, u_m)$, and is an instance of the atom $g(u_1, \ldots, u_m)$ which shares variables with the subgoal in the plan instance. Each plan instance also has a set of *execution state flags* σ. σ is subset of a set of flags *Flags* which includes at least `intended`, `scheduled`, `stepped` and `failed`, and may contain additional user-defined flags, e.g., some deliberation strategies may require a `suspended` execution state. The `scheduled` flag indicates that the plan instance is selected to be executed at the current deliberation cycle. The `stepped` flag indicates that the plan instance was executed at the last cycle. Finally, the `failed` flag indicates that attempting to execute the plan instance failed, e.g., a mental state test returned false, or attempting to execute an external action failed.

The plan instances comprising the plan state of an agent can be queried and updated using the following primitive operations:

- $\mathsf{plan}(i, \pi)$: i is the *id* of an instance of the plan π
- $\mathsf{plan\text{-}remainder}(i, \pi)$: π is the textual representation of the (unexecuted) suffix of the plan instance with *id* i
- $\mathsf{justification}(i, j)$: the plan instance with *id* i has the atom instance with *id* j as a justification
- $\mathsf{substitution}(i, \theta)$: the plan instance with *id* i has substitution θ
- $\mathsf{subgoal}(i, j)$: j is the *id* of the subgoal of the plan instance with *id* i, i.e., $\mathsf{plan\text{-}remainder}(i, !\,g; \pi)$ and $atom(j, g)$ such that j is the *id* of the instance of g created by executing $!\,g$ in i
- $\mathsf{state}(i, \sigma)$: the plan instance with *id* i has execution state flags σ
- $\mathsf{set\text{-}remainder}(i, \pi)$ set the (unexecuted) suffix of the plan instance with *id* i to π
- $\mathsf{set\text{-}substitution}(i, \theta)$: set the substitution of the plan instance with *id* i to θ, where θ may be an implicit substitution resulting from the unification of two terms $t(x) = t(a)$
- $\mathsf{set\text{-}state}(i, \sigma)$ set the execution state flags of the plan instance with *id* i to σ
- $\mathsf{delete\text{-}plan}(i)$: delete the plan instance with *id* i, together with its suffix, substitution and subgoal (if any)
- $\mathsf{cycle}(n)$: the current deliberation cycle is n

Clauses & Macros Additional mental state and plan state queries can be defined using Prolog-style Horn *clauses* of the form:

$$q \leftarrow q_1, \ldots, q_n$$

where $q_1, \ldots, q_n$ are mental or plan state queries or their negation. Negation is interpreted as negation as failure, and we assume that the set of clauses is always *stratified*, i.e., there are no cycles in predicate definitions involving negations. Clauses are evaluated as a sequence of queries, with backtracking on failure.

Additional mental state and plan state actions can be defined using *macros*. A macro is a sequence of mental state and/or plan state queries/tests and actions. Macros are evaluated left to right, and evaluation aborts if an action or query/test fails. For example, the mental state action add-atom(a) which does not return an instance id can be defined by the macro: add-atom(b) = $\mathsf{add\text{-}atom}(\_, b)$. Macros can also be used to define *type specific* mental state actions, e.g., to add an instance of the atom b as a belief and signal a belief addition event as in *Jason* [2], we can use the macro

$$\text{add-belief}(b) = \text{add-atom}(\text{belief}(b)), \text{add-atom}(+\text{belief}(b))$$

Object Rules To select appropriate plans given its mental state, an agent uses *object rules*. Object rules correspond to plan selection constructs in conventional BDI agent programming languages, e.g., plans in *Jason* [2], or PG rules in 3APL [4]. The syntax of an object rule is given by:

$$\mathit{reasons}\ [\ :\ \mathit{context}\ ] \rightarrow \pi$$

where *reasons* is a conjunction of non-negated primitive mental state queries, *context* is boolean expression built of mental state queries and π is a plan. The context may be null (in which case the ":" may be omitted), but each plan instance must be justified by at least one reason. The reason and the context are evaluated against the agent's mental state and both must return true for π to be selected. Firing an object rule gives rise to a new instance of the plan π that forms the right hand side of the rule which is justified by the atom instances matching the *reasons*. For example, the following object rule selects a plan to brings beer the robot's owner:

$$\begin{aligned}&\text{has}(\text{owner}, \text{beer})\ :\ \text{available}(\text{beer}, \text{fridge}),\ \text{not drunk}(\text{owner}) \rightarrow\\ &\quad !\text{at}(\text{robot}, \text{fridge});\ \text{open}(\text{fridge});\ \text{get}(\text{beer});\ \text{close}(\text{fridge});\\ &\quad !\text{at}(\text{robot}, \text{owner});\ \text{give}(\text{beer});\ ?\text{date}(d);\ \text{add-atom}(\_, \text{consumed}(\text{beer}, d))\end{aligned}$$

Meta-rules To update the agent's state, specify which plan instances to adopt as intentions and select which intentions to execute in a given cycle an agent uses *meta-rules*. The syntax of a meta-rule is given by:

$$\mathit{meta\text{-}context} \rightarrow m_1;\ \ldots\ ; m_n$$

where *meta-context* is a boolean expressions built of mental state and plan state queries and $m_1, \ldots, m_n$ is a sequence of mental state and/or plan state actions. When a meta-rule is fired, the actions that form its right hand side are immediately executed.For example, the following meta-rule selects a 'root' plan for an intention when the agent has no current intention

$$\text{not intention}(\_),\ \text{plan}(i, \_) \rightarrow \text{add-intention}(i)$$

Meta-APL Programs A meta-APL program $(D, \mathcal{R}_1, \ldots, \mathcal{R}_k, A)$ consists of a set of clause and macro definitions D, a sequence of rule sets $\mathcal{R}_1, \ldots, \mathcal{R}_k$, and a set of initial atom instances A. Each *rule set* $\mathcal{R}_i$ is a set of object rules or a set of meta-rules that forms a component of the agent's deliberation cycle. For example, rule sets can be used to update the agent's mental and plan state, propose plans or create and execute intentions.

3.2 Meta-APL Core Deliberation Cycle

The meta-APL core deliberation cycle consists of three main phases. In the first phase, a user-defined *sense* function updates the agent's mental state with atom instances resulting from perception of the agent's environment, messages from other agents etc. In the second phase, the rule sets comprising the agent's program are processed in sequence. The rules in each rule set are run to quiescence to update the agent's mental and plan state. Each rule is fired once for each matching set of atom and/or plan instances. Changes in the mental and plan state resulting from rule firing directly update the internal (implementation-level) representations maintained by the deliberation cycle, which may allow additional rules to match in the same or subsequent rule sets. Finally, in the third phase, the next step of all `scheduled` object-level plans is executed. The deliberation cycle then repeats. Cycles are numbered starting from 0 (initial cycle), and the cycle number is incremented at each new cycle.

In the remainder of this section, we briefly summarise how the key steps in the execution of meta-APL are reflected in the agent state — full details are given in [5]. The firing of an object-level rule creates a new plan instance, together with a justification associating the atom instances matching each mental state query in the *reasons* of the object-level rule with the plan instance. The initial substitution of the plan instance is the result of evaluating the mental state queries in the *reason* and *context* of the corresponding object-level rule with the mental state of the agent. The execution of a plan instance updates the agent's mental and plan state. The execution of a mental state query may update the substitution of the plan instance. The execution of a mental state action may add or remove an atom instance from the agent's state. Deleting an atom instance with *id* i also deletes all atom instances that have i as argument. In addition, if any of the justifications of a plan instance are deleted, the plan instance, together with any subgoal of the plan instance are also deleted (recursively). The evaluation of a subgoal creates a new instance of the goal atom (with the substitution of the plan instance applied to any variables in the goal), together with a subgoal relation associating the plan instance and the new instance of the goal atom. Executing an action in a plan instance also updates the plan-remainder of the instance. The firing of a meta-rule immediately executes the meta-actions on the RHS of the rule. The meta actions may add or delete atom instances, set the state or substitution of a plan instance, or delete it.

4 Encoding Deliberation Strategies

In this section, we show how to encode the deliberation strategies given in Section 2 in meta-APL. We assume that we are given a user program expressed as a set of meta-APL object rules $\mathcal{R}_2$, and we show how to execute this program under the three different strategies. The encoding of each strategy takes the form of a meta-APL program $(D, \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, A)$, where D and $\mathcal{R}_1$ are a set of clause and macro definitions and a set of meta-rules common to all deliberation strategies, $\mathcal{R}_2$ is the user program, and $\mathcal{R}_3$ is the encoding of the deliberation strategy itself.

We first define D and $\mathcal{R}_1$. D contains the following clause-definable plan state queries:

- intention(i): the plan instance with *id* i is intended

$$\text{intention}(i) \leftarrow \text{state}(i, \sigma),\ \text{member}(\texttt{intended}, \sigma)$$

- intended-plan(j, i): the plan instance with *id* i is the intended means for the reason (e.g., belief or goal event) with *id* j

$$\text{intended-plan}(j, i) \leftarrow \text{justification}(i, j),\ \text{intention}(i)$$

- executable-intention(i): the intention with *id* i has no subgoal (hence no subintention)

$$\text{executable-intention}(i) \leftarrow \text{intention}(i),\ \text{not subgoal}(i, \_)$$

- scheduled(i): a step of the plan instance with *id* i will be executed at the current deliberation cycle

$$\text{scheduled}(i) \leftarrow \text{state}(i, \sigma),\ \text{member}(\texttt{scheduled}, \sigma)$$

and the macro-definable plan state actions:

- add-intention(i): add the `intended` flag to the plan instance with *id* i

$$\text{add-intention}(i) = \text{state}(i, \sigma),\ \text{set-state}(i, \sigma \cup \{\texttt{intended}\})$$

- schedule(i): add the `scheduled` flag to the plan instance with *id* i

$$\text{schedule}(i) = \text{state}(i, \sigma),\ \text{set-state}(i, \sigma \cup \{\texttt{scheduled}\})$$

$\mathcal{R}_1$ contains meta-rules to remove non-intended plan instances from the previous cycle and to remove completed intentions:

$$\begin{aligned}
\mathcal{R}_1 = {} & \text{plan}(i, _),\ \text{not intended}(i) \rightarrow \text{delete-plan}(i) \\
& \text{executable-intention}(i),\ \text{plan-remainder}(i, \epsilon),\ \text{justification}(i, j), \\
& \quad \text{not subgoal}(_, j) \rightarrow \text{delete-atom}(j) \\
& \text{executable-intention}(i),\ \text{plan-remainder}(i, \epsilon),\ \text{justification}(i, j), \\
& \quad \text{subgoal}(k, j),\ \text{substitution}(k, s),\ \text{substitution}(i, s') \\
& \quad \rightarrow\ \text{set-substitution}(k, s \cup s'),\ \text{delete-atom}(j)
\end{aligned}$$

The first rule removes non-intended plan instances generated at the previous cycle. The second rule removes the 'root' plan of a completed intention and its associated triggering event (reason). The plan must have finished execution (have an empty remainder), and be executable, i.e., have no pending subgoal. (In the case that the plan body is empty but has an active child plans, the plan instance should not be removed because any child plans would also be removed.) The third rule removes a completed 'leaf' plan of an intention. The substitution of the parent plan instance is extended with the substitution of the completed plan instance, and the subgoal justifying the completed plan instance is deleted.

We can now define the deliberation strategies.

4.1 Non-interleaved (ni)

In the non-interleaved strategy **(ni)**, the agent executes a single intention at a time. It can be encoded as the following set of meta-rules:[4]

$$\begin{aligned}\mathcal{R}_3 \;= \;&\text{not intention}(\_),\ \text{plan}(i,\_) \rightarrow \text{add-intention}(i)\\ &\text{subgoal}(\_,j),\ \text{not intended-plan}(j,\_),\ \text{justification}(i,j)\\ &\quad\rightarrow \text{add-intention}(i)\\ &\text{executable-intention}(i) \rightarrow \text{schedule}(i)\end{aligned}$$

The first rule selects a 'root' plan for an intention when the agent has no current intention. The plan is selected non-deterministically from the plan instances that are generated by the execution of the program's object-level rules. The second rule extends an intention by adding a new 'leaf' plan for a subgoal for which there is currently no intended plan. The third rule re-enables the (leaf) intention for execution at the current cycle. Together, these rules ensure that the agent progresses a single intention to completion, even though the meta-APL core deliberation cycle generates all relevant applicable plan instances at each cycle.

4.2 Alternating (Single Action) (as)

In the **(as)** strategy, a plan instance from the set of plan instances generated by the object-level rules is intended, and a single intention is `scheduled` for execution. For the **(as)** strategy defined in [1], in which the intention to be executed is chosen non-deterministically from the set of intentions, two meta-rules suffice for implementation.

$$\begin{aligned}\mathcal{R}_3 \;= \;&\text{cycle}(c),\ \text{not}(\text{selected-plan}(\_,c),\ \text{intended-plan}(j,\_)),\ \text{justification}(i,j)\\ &\rightarrow\ \text{add-intention}(i), \text{add-atom}(\_, \text{selected-plan}(i,c))\\ &\text{not scheduled}(\_),\ \text{executable-intention}(i) \rightarrow \text{schedule}(i)\end{aligned}$$

[4] Encodings of variants of the **(ni)** and round robin strategies are given in [5].

The first rule selects a plan instance for a reason (e.g., (sub)goal) j for which there is no current intention, and adds the plan instance as an intention. To ensure that at most one plan instance is intended at each deliberation cycle, we also record the fact that a plan has been selected at this cycle by adding a selected-plan atom to the agent's mental state. The second rule non-deterministically selects an executable intention and schedules it.

For a round robin (RR) strategy, the implementation is slightly more involved. To ensure fairness, we must keep track of which intention has been *least recently executed*. There are several ways in which this could be done. We adopt the the most straightforward approach, which is to explicitly record the cycle at which each intention was last executed in the agent's mental state. We extend D with the clause and macro definitions

- least-recently-executed(i): the intention with *id* i is least recently executed

$$\begin{aligned}\text{least-recently-executed}(i) \leftarrow\ & \text{intention}(i),\ \text{intention}(i'),\ \text{not } i = i',\\ & \text{last-executed}(i, c),\ \text{last-executed}(i', c'),\ \text{not } c' < c\end{aligned}$$

- executed(i, c): record that the intention with *id* i was executed at cycle c

$$\begin{aligned}\text{executed}(i, c)\ =\ & \text{atom}(j, \text{last-executed}(i, \_)),\ \text{delete-atom}(j),\\ & \text{add-atom}(\_, \text{last-executed}(i, c))\end{aligned}$$

The round robin strategy can then be encoded as:

$$\begin{aligned}\mathcal{R}_3\ =\ & \text{cycle}(c),\ \text{not}(\text{selected-plan}(\_, c),\ \text{intended-plan}(j, \_)),\ \text{justification}(i, j)\\ & \rightarrow\ \text{add-intention}(i), \text{add-atom}(\text{selected-plan}(i, c))\\ & \text{not scheduled}(\_),\ \text{least-recently-executed}(i),\ \text{cycle}(c)\\ & \rightarrow \text{schedule}(i),\ \text{executed}(i, c)\end{aligned}$$

The first rule non-deterministically selects a plan for a reason for which there is no current intention, and is the same as in the simple **(as)** strategy above. The second rule schedules the least recently executed intention, and records the fact that it was last executed at the current deliberation cycle.

4.3 Alternating (Multi-action) (am)

In the alternating multiple **(am)** strategy, a plan instance from the set of plan instances generated by the object-level rules is intended, and a single step of all current intentions are executed at each cycle. This strategy can be encoded as:

$$\begin{aligned}\mathcal{R}_3\ =\ & \text{cycle}(c),\ \text{not}(\text{selected-plan}(\_, c),\ \text{intended-plan}(j, \_)),\ \text{justification}(i, j)\\ & \rightarrow\ \text{add-intention}(i), \text{add-atom}(\text{selected-plan}(i, c))\\ & \text{executable-intention}(i),\ \text{not scheduled}(i) \rightarrow \text{schedule}(i)\end{aligned}$$

The first rule non-deterministically selects a plan for a reason for which there is no current intention and is the same as in the **(as)** strategy. The second rule simply schedules all executable intentions.

5 An Adaptive Deliberation Strategy

It is straightforward to encode variations of the 'standard' deliberation strategies considered in the previous section in meta-APL. For example, it is possible to encode strategies that take preferences regarding plans into account as in [14].

However, in this section we illustrate the flexibility of meta-APL by presenting a novel adaptive deliberation strategy that combines features of both the **(ni)** and **(as)** (or **(am)**) strategies. As noted in Section 2 the **(ni)** strategy has the advantage that it minimises conflicts between intentions. However it has the disadvantage that the agent is unable to respond to new goals until (at least) the intention for the current goal has been executed. Conversely, the **(as)** and **(am)**) strategy allows an agent to pursue multiple goals at the same time. However it can increase the risk that actions in plans in different intentions will interfere with each other. In this section we define an adaptive strategy that interleaves steps in intentions where this does not result in conflicts between intentions. If conflicts are inevitable, it defers execution of one or more intentions in an attempt to avoid the conflict. As such it avoids the need for the programmer to insert *atomic* constructs that prevent the interleaving of actions in a plan in one intention with actions from plans in other intentions.[5]

The adaptive strategy checks for conflicts between the postconditions of the next action in each of the agent's current intentions. If the postconditions conflict, e.g., if the next action in one intention would cause the agent to move to the left while the next action in another intention would cause it move to the right, execution of one of the intentions is deferred. At each deliberation cycle, the strategy:

- computes the effects of the first action in the remainder of each intention
- checks the effects to identify conflicts
- if there are conflicts, defers the execution of one or more intentions

We will now explain how to encode this strategy in meta-APL, starting with techniques for computing the effects of each type of plan element. Mental state tests have no effects on the mental state or environment, and therefore cannot cause a conflict of the kind described above. Similarly subgoals can't give rise to conflicts. Mental state actions are considered to conflict if they add and delete the same atom. (For simplicity, we consider only addition and deletion here. It would be straightforward to extend the check to simple forms of logical inconsistency.) The effects of mental state actions can be determined by inspection of the code and so require no additional information from the programmer. External actions are considered to conflict if they result in incompatible states of the environment, e.g., the fridge being open and it being closed. As this information can't be inferred from the program text, it must be provided by the programmer.

[5] As with the use of atomic constructs, the approach we present here does not guarantee that a set of intentions can be executed successfully, e.g., where one intention destroys a precondition required for the execution of another intention and which can't be regenerated.

There are several ways in which this could be done. For simplicity, we assume that each external action has a single postcondition, i.e., actions are deterministic. (This could be extended to, for example, make the postcondition dependent on the agent's current beliefs.) We further assume that the programmer specifies the effect of each external action as an 'effect' unit clause, and provides a definition for a predicate 'conflict' that returns true if two postconditions denote incompatible states of the environment. In the example below, 'conflict' is defined using a set of unit clauses, but again more complex approaches are possible. Lastly, we assume that any macros appearing in plans in the user program are inlined at the call site recursively when the plan suffix is returned by the plan-remainder primitive.

An adaptive deliberation strategy can be encoded as follows. We extend D with the clause definitions

- conflicting(i, i'): the next step in intentions with *id* i, i' have incompatible effects

$$\begin{aligned}\text{conflicting}(i, i') \leftarrow\ & \text{intention}(i),\ \text{intention}(i'),\ \text{not } i = i', \\ & \text{plan-remainder}(i, s\,; \ldots),\ \text{plan-remainder}(i', s'\,; \ldots), \\ & \text{conflicts}(s, s')\end{aligned}$$

- conflicts(s, s'): the plan steps s and s' have incompatible effects

$$\begin{aligned}\text{conflicts}(s, s') &\leftarrow \text{effect}(s, e),\ \text{effect}(s, e'),\ \text{conflict}(e, e') \\ \text{conflicts}(s, s') &\leftarrow \text{effect}(s, e),\ \text{effect}(s, e'),\ \text{conflict}(e', e)\end{aligned}$$

We assume that the definitions of the predicates 'effect' and 'conflict' for external actions are given in the ruleset $\mathcal{R}_2$ containing the user program. The adaptive deliberation strategy itself is encoded as

$$\begin{aligned}\mathcal{R}_3 = \ & \text{executable-intention}(i),\ \text{not}\ (\text{scheduled}(i),\ \text{conflicting}(i, \_)) \\ & \quad \rightarrow\ \text{schedule}(i) \\ & \text{executable-intention}(i),\ \text{not scheduled}(i),\ \text{conflicting}(i, i'),\ \text{not } i' < i \\ & \quad \rightarrow\ \text{schedule}(i)\end{aligned}$$

The first rule schedules all intentions that do not conflict with any other intention. The second rule schedules a single intention from a set of conflicting intentions (chosen arbitrarily to be the one with lowest *id*). By permitting the execution of one conflicting intention per cycle, we avoid deadlock.

The approach above essentially implements single step lookahead. However in many cases, an external action in a plan establishes a precondition for a later step in the same plan (these are called *p-effects* in [13]). For example, the action of going to a particular location such as the fridge may establish the precondition for a later action which must be performed at that location such as opening the fridge. We can extend the adaptive strategy to avoid this kind of conflict by

taking the preconditions of actions into account when evaluating conflicts. We briefly sketch one way in which this can be done below.

The only changes required are to modify the definitions of the predicate 'conflicting' to consider plan suffices rather than the next steps of intentions, and of the predicate 'conflicts' to consider both pre- and postconditions. (For simplicity, we assume that each external action has a single precondition.)

$$\begin{aligned}
\text{conflicting}(i, i') \;&\leftarrow \text{intention}(i),\ \text{intention}(i'),\ \text{not } i = i', \\
&\quad\ \text{plan-remainder}(i, \pi),\ \text{plan-remainder}(i', \pi'), \\
&\quad\ \text{conflicts}(\pi, \pi') \\
\text{conflicts}(\pi, \pi') \;&\leftarrow \text{effects}(\pi, es),\ \text{conditions}(\pi', cs), \\
&\quad\ \text{member}(e, es),\ \text{member}(c, cs),\ \text{conflict}(e, c) \\
\text{conflicts}(\pi, \pi') \;&\leftarrow \text{effects}(\pi', es),\ \text{conditions}(\pi, cs), \\
&\quad\ \text{member}(e, es),\ \text{member}(c, cs),\ \text{conflict}(e, c)
\end{aligned}$$

The predicates 'effects' and 'conditions' return the set of postconditions and the set of pre- and postconditions of a plan suffix π respectively, and are omitted due to lack of space.

6 Related Work

The Procedural Reasoning System (PRS) [9] had a meta-level, namely the ability to program deliberation about the choice of plans in the language itself. Since PRS, there have been several attempts to make the deliberation cycle of agent programming languages programmable. For example, 3APL enables the programmer to modify 3APL interpreter deliberation cycle [4]. It provides a collection of Java classes for each mental attitude, where each class has a collection of methods representing operations for manipulating this attitude. In order to implement a particular deliberation cycle, the programmer should essentially modify the interpreter to call the methods of those Java classes in a particular order. This idea of extending 3APL with a set of programming constructs which allowed the deliberation cycle to be programmed was proposed in an earlier paper [3], where the authors explicitly consider the option of adding meta-actions to 3APL as basic actions, and programming the interpreter in 3APL itself. However they argued against this approach on the grounds that it would give "too much" expressive power to the programmer and would make the meta-level hard to program. They opted instead for providing a simple separate language for programming deliberation cycle which uses a small set of primitives. The language is imperative and extends that proposed in [10], mainly by adding a primitive to call a planner and generate a new plan. Plans can be compared on the grounds of cost with the gain from achieving the goal, and a plan which has a cost less than gain selected. We share the motivation for providing an ability for the programmer in an agent programming language to change the deliberation cycle on a program-by-program basis, but believe that it is more natural and elegant to do this in the same language, rather than joining together two different languages.

In [7] Doan et al show how *Jason* and 3APL programs (and their associated deliberation strategies) can be translated into meta-APL to give equivalent behaviour under weak bisimulation equivalence.

There has also been a considerable amount of work on avoiding conflicts between intentions in agent programming languages, e.g., [13,15,16]. While such approaches can avoid more conflicts than the approach we present here, this work focus on developing a single deliberation strategy that is 'hardwired' into the the deliberation cycle of an agent programming language, and which cannot be easily adapted by an agent developer to meet the needs of a particular application.

7 Conclusion

In this paper, we showed how procedural reflection in the agent programming language meta-APL [7] can be used to allow a straightforward implementation of the deliberation strategy of a BDI agent. To illustrate the flexibility of meta-APL, we showed how three typical BDI deliberation strategies from [1] can be programmed using meta-APL rules. We also showed how meta-APL can used to program a novel *adaptive* deliberation strategy combines features of both a non-interleaved and an alternating strategy to avoid interference between intentions.

By exploiting procedural reflection an agent programmer can customise the deliberation cycle to control when to deliberate (as in the non-interleaved strategy), which relevant applicable plan(s) to intend, and which intention(s) to execute. We argue this brings the advantages of the BDI approach to the problem of selecting an appropriate deliberation strategy given the agent's current state, and moreover, facilitates a modular, incremental approach to the development of deliberation strategies. In future work, we plan to explore more sophisticated approaches to the selection of plan instances as in, e.g., [14], and intention reconsideration.

References

1. Natasha Alechina, Mehdi Dastani, Brian Logan, and John-Jules Ch. Meyer. Reasoning about agent deliberation. *Autonomous Agents and Multi-Agent Systems*, 22(2):1–26, 2010.
2. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley, 2007.
3. M. Dastani, F. de Boer, F. Dignum, and J.J. Ch. Meyer. Programming agent deliberation: an approach illustrated using the 3APL language. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, pages 97–104. ACM, 2003.
4. Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Programming multi-agent systems in 3APL. In *Multi-Agent Programming: Languages, Platforms and Applications*, pages 39–67. Springer, 2005.
5. Thu Trang Doan. *Meta-APL: A general language for agent programming*. PhD thesis, School of Computer Science, University of Nottingham, 2013.

6. Thu Trang Doan, Natasha Alechina, and Brian Logan. The agent programming language meta-APL. In Louise A. Dennis, Olivier Boissier, and Rafael H. Bordini, editors, *Proceedings of the Ninth International Workshop on Programming Multi-Agent Systems (ProMAS 2011)*, pages 72–87, Taipei, Taiwan, May 2011.
7. Thu Trang Doan, Yuan Yao, Natasha Alechina, and Brian Logan. Verifying heterogeneous multi-agent programs. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 149–156, Richland, SC, 2014. International Foundation for Autonomous Agents and Multiagent Systems.
8. Michael P. Georgeff, Barney Pell, Martha E. Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In Jörg P. Müller, Munindar P. Singh, and Anand S. Rao, editors, *Intelligent Agents V, Agent Theories, Architectures, and Languages, 5th International Workshop, (ATAL'98), Paris, France, July 4-7, 1998, Proceedings*, volume 1555 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1999.
9. M.P. Georgeff and A.L. Lansky. Reactive reasoning and planning. In *Proceedings of AAAI-87*, pages 677–682, 1987.
10. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
11. Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 331–347, New York, NY, USA, 1984. ACM.
12. B. C. Smith. Reflection and semantics in lisp. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.
13. John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & avoiding interference between goals in intelligent agents. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 721–726, Acapulco, Mexico, August 2003. Morgan Kaufmann.
14. Simeon Visser, John Thangarajah, James Harland, and Frank Dignum. Preference-based reasoning in bdi agent systems. *Autonomous Agents and Multi-Agent Systems*, pages 1–40, 2015.
15. Max Waters, Lin Padgham, and Sebastian Sardina. Evaluating coverage based intention selection. In Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns, editors, *Proceedings of the 13th International Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2014)*, pages 957–964. IFAAMAS, 2014.
16. Yuan Yao, Brian Logan, and John Thangarajah. SP-MCTS-based intention scheduling for BDI agents. In *Proceedings of the 21st European Conference on Artificial Intelligence*, Prague, Czech Republic, August 2014. ECCAI, IOS Press.