Atomic Intentions in Jason$^+$

Daniel Kiss, Neil Madden, and Brian Logan

School of Computer Science
University of Nottingham, UK.
bsl@cs.nott.ac.uk

Abstract. We consider interactions between *atomic intentions* and plan failures in the *Jason* BDI-based agent programming language. Atomic intentions allow the agent developer to control the execution of intentions in situations where a sequence of actions must be executed 'atomically' in order to ensure the success of a plan. However, while atomic intentions in *Jason* enforce mutual exclusion, they are not atomic operations in the sense understood in conventional programming or in databases, and failure of an atomic plan can leave the agent's belief and plan bases in an inconsistent state. In this paper we present a new approach to atomic intentions which provides a transactional 'all-or-nothing' semantics, and describe its implementation in a new version of *Jason*, *Jason*$^+$. We argue that *Jason*$^+$ offers a more predictable semantics for atomic plans in the face of plan failure and can reduce the load on the agent developer by automating simple cases of failure handing, leading to the development of more robust agent programs.

1 Introduction

Jason [1] is a Java-based interpreter for an extended version of AgentSpeak(L). AgentSpeak(L) is a high-level agent-oriented programming language [2] which incorporates ideas from the BDI (belief-desire-intention) model of agency. Since it was first introduced in [3], *Jason* has evolved into a rich platform for the development of multi-agent systems. It has many sophisticated features to facilitate the development of complex agent-based systems, including control over the scheduling of intentions, support for plan exchange between agents, and facilities for handling plan failures. In this paper, we investigate some of the interactions between these features (which are also found in other BDI-based agent programming languages), and highlight some of the potential problems that can arise from their interaction.

We focus on a key feature of *Jason*, *atomic intentions*, and how these interact with *Jason*'s facilities for handling plan failures. Atomic intentions allow the agent developer to control the execution of intentions in situations where a sequence of actions must be executed 'atomically' in order to ensure the success of a plan. However, while atomic intentions enforce mutual exclusion, they are not atomic operations in the sense understood in conventional programming or in databases. In particular, we show that failure of an atomic plan can leave the agent's belief and plan bases in an inconsistent state, and further that the existing failure handling mechanisms implemented in *Jason* make it hard to recover from such failures.

To overcome these problems we have developed *Jason*$^+$, an extension of the *Jason* agent programming language which adopts a *transactional* approach to atomic plans. If

an action in a *Jason*$^+$ atomic plan fails, the whole atomic plan fails and the agent's belief and plan bases and event list are left in the state they were before the execution of the atomic plan. However if an atomic plan succeeds, all updates to the agent's belief and plan bases made by the atomic plan are visible to any subsequent actions in the intention containing the atomic plan and to actions in other intentions, and all events generated by the atomic plan are added to the event list. *Jason*$^+$ also implements a version of the module system described in [4] which is used to encapsulate dynamically loaded plans. With the exception of minor changes required by the module system, the syntax of *Jason*$^+$ is backwards compatible with the current release of *Jason*. We believe that the extensions implemented in *Jason*$^+$ offer a more predictable semantics for atomic plans in the face of plan failure and can reduce the load on the agent developer by automating simple cases of failure handing, leading to the development of more robust agent programs.[1]

The remainder of this paper is structured as follows. In the next section we briefly summarise the syntax and semantics of *Jason*. In section 3 we motivate the need for atomic intentions, and highlight some of the problems of the current implementation of atomic intentions in *Jason*. In section 4 we present a new approach to atomic intentions which provides a transactional 'all-or-nothing' semantics, and describe its implementation in a new version of *Jason*, *Jason*$^+$. In section 5 we briefly discuss how our approach could be generalised to other BDI-based agent programming languages, and outline plans for future work.

2 *Jason*

Jason is loosely based on the logic programming paradigm, but with an operational semantics based on plan execution in response to events and beliefs rather than SLD resolution as in Prolog.

A *Jason* agent is specified in terms of beliefs, goals and plans. An agent's beliefs represent its information about its environment, e.g., sensory input, information about other agents, etc. Beliefs are represented as ground atomic formulas. For example, the agent may believe that John is a customer: `customer(John)`. A goal is a state the agent wishes to bring about or a query to be evaluated. An achievement goal, written $!g(t_1, \ldots, t_n)$ where $t_i, \ldots, t_n$ are terms, specifies that the agent wishes to achieve a state in which $g(t_1, \ldots, t_n)$ is a true belief. A test goal, written $?g(t_1, \ldots, t_n)$, specifies that the agent wishes to determine if $g(t_1, \ldots, t_n)$ is a true belief. For example, an agent may have a goal `!order(widget, 10)` to process a purchase order for 10 widgets. (As in Prolog, constants are written in lower case and variables in upper case, and all negations must be ground when evaluated.) *Jason* extends AgentSpeak(L) with support for more complex beliefs, default and strong negation, and arbitrary internal actions implemented in Java. The belief base of AgentSpeak(L) consists simply of a set of ground literals, whereas *Jason* supports a sizeable subset of Prolog for the belief base, including universally-quantified rules (Horn clauses).

Changes in the agent's beliefs or the acquisition of new achievement goals give rise to *events*. An addition event, denoted by $+$, indicates the addition of a belief or an

[1] *Jason*$^+$ is available at `http://code.google.com/p/jasonp`.

achievement goal. A deletion event, denoted by $-$, indicates the retraction of a belief or goal. *Plans* specify sequences of actions and subgoals an agent can use to achieve its goals or respond to events (changes in its beliefs and goals). The head of a plan consists of a triggering event which specifies the kind of event the plan can be used to respond to, and a belief context which specifies the beliefs that must be true for the plan to be applicable. The body of a plan specifies a sequence of actions and (sub)goals to respond to the triggering event. Actions are the basic operations an agent can perform to change its environment (or its internal state) in order to achieve its goals. Plans may also contain achievement and test (sub)goals. Achievement subgoals allow an agent to choose a course of action as part of a larger plan on the basis of its current beliefs. An achievement subgoal $!g(t_1, \ldots, t_n)$ gives rise to a internal goal addition event $+!g(t_1, \ldots, t_n)$ which may in turn trigger subplans at the next execution cycle. Test goals are evaluated against the agent's belief base, possibly binding variables in the plan.[2] For example, an agent which forms part of an e-Commerce application might use the simple plan shown if Figure 1 to processes orders.

```
/* Initial stock */
stock(widget, 1).

@processOrder
+!order(Item, Num)[source(Customer)] <-
    ?stock(Item, Stock);            /* test plan */
    Num <= Stock;                   /* test plan */
    -+stock(Item, Stock - Num);     /* modify stock */
    +purchased(Customer, Item, Num);
    !dispatch(Customer, Item, Num).
```

Fig. 1. Example *Jason* plan.

The current stock level of items is stored as a ground fact in the belief base. The plan checks that there is sufficient stock to process the order, updates the agent's beliefs to reflect the reduced stock level and record the customer's order and then generates a subgoal to dispatch the order to the customer.

At each reasoning cycle, *Jason* first processes any events and updates the agent's belief base. The interpreter then selects a single event to process and matches it against the plan library to select one or more plans to handle the event. Of these plans, a single plan is then selected to become an intention. Finally, one of the currently active intentions is selected and the next step in that intention is executed, before the cycle repeats.

Jason has been extended in a variety of ways, for example, to support decision theoretic scheduling of intentions [5]. Of particular interest here is the support provided in core *Jason* for cooperative plan exchange as described in [6, 7]. In the cooperative BDI paradigm, agents can retrieve plans from other agents. Such plans can be dynamically

[2] In the interests of brevity, we have slightly simplified the presentation of *Jason* syntax and semantics.

loaded into the agent's plan base to extend its capabilities at run time. For example, the *Jason* interpreter can be configured to invoke the user-defined `selectOption` function when the set of relevant and applicable plans for an event is empty. Internal actions are also provided to allow plans to be added and removed from the agent's plan base at run time. These facilities can be used to implement the kind of plan exchange described in [7].

3 Atomic Intentions

The basic *Jason* execution strategy allows an agent to pursue multiple goals at the same time, e.g., an agent can respond to an urgent, short-duration task while engaged in a long-term task, and such flexibility is a key characteristic of an agent-oriented approach to software development. However it can increase the risk that actions in different plans will interfere with each other. For example, if an agent has an intention containing a "go right" action and another intention containing a "go left" action, their interleaved execution may result in the agent returning to its original location. (The default *Jason* intention selection function implements a form of 'round robin' scheduling: each intention is selected in turn and a single step of that intention is executed, resulting in fully-interleaved execution of plans.) More generally, the successful execution of an action in a plan (and ultimately achievement of the agent's goals) typically requires that preconditions established by actions earlier in the plan continue to hold when the action is eventually executed. This can be difficult or impossible to guarantee with unconstrained interleaving of agent plans. For example, the example *Jason* plan shown if Figure 1 may result in incorrect behaviour. If two orders for a widget arrive at about the same time, an agent using the default *Jason* intention selection function may attempt to dispatch the same widget to both customers. Such race conditions are well known in multi-threaded conventional programs, and it is not surprising that agent programming languages that support the parallel execution of intentions should face similar challenges.

It is therefore important that the agent developer can control which intentions are selected for execution. For example, there may be situations where a sequence of actions must be executed 'atomically' in order to ensure the success of a plan. *Jason* allows such control to be exercised in two ways: by over-riding the default intention selection function and by use of `atomic` plan annotations (both are optional). In what follows, we focus on `atomic` annotations, as these provide built-in support for atomic execution of intentions.

3.1 Atomic Plans in Jason

An `atomic` plan annotation indicates that a plan should be executed 'atomically'. If an intention containing a plan with an `atomic` annotation is selected for execution by the agent's intention selection function, it will continue to be selected for execution in subsequent execution cycles until the plan completes. In effect, an `atomic` annotation over-rides the normal operation of the event and intention selection functions during the execution of the atomic plan, and guarantees that the execution of actions in other intentions will not be interleaved with the steps of an atomic plan. (Note that unlike mutual

exclusion in conventional programming languages, e.g., `synchronized` methods in Java, an atomic plan in *Jason* is mutually exclusive with *all* other intentions, not just those containing atomic plans.) For example, adding an `atomic` annotation to the example order processing plan above ensures that all of its actions are run to completion before another intention is selected, and so avoid the possibility of attempting to dispatch the same item twice.

```
@processOrder[atomic]
+!order(Item, Num)[source(Customer)] <-
    ?stock(Item, Stock);            /* test plan */
    Num <= Stock;                   /* test plan */
    -+stock(Item, Stock - Num);     /* modify stock */
    +purchased(Customer, Item, Num);
    !dispatch(Customer, Item, Num).
```

Fig. 2. Example *Jason* atomic plan.

While `atomic` annotations are essential for many applications, they have to be used with care to maintain the reactivity of the agent, as the agent is unable to respond to external events during the execution of an atomic plan. However of greater interest here is the way in which atomic annotations interact with plan failure. While the `atomic` annotation ensures mutual exclusion (no other intention can interfere with the execution of an atomic plan), atomic plans in *Jason* are not atomic operations in the sense understood in conventional programming or in databases. In conventional programming, an 'atomic' action is effectively uninterruptible—any intermediate states which arise during the execution of an atomic operation are not visible to other processes even if the operation fails. In contrast, intermediate, possibly inconsistent, states resulting from the execution of a *Jason* atomic plan may be visible to other intentions if the plan fails.

3.2 Failure of Atomic Plans

In the open environments characteristic of MAS applications, it is impossible to guarantee that an agent's actions will always be successful. Of the six types of formulas which can appear in *Jason* plans (internal and external actions, achievement and test goals, expressions and mental notes), five can fail. Only mental notes are guaranteed to succeed. When a step in a plan fails, the plan itself is said to have failed. For example, the example plan in Figure 2 can fail either because there is insufficient stock to process the order, or if attempting to achieve the `dispatch` goal fails.

Jason provides support for dealing with plan failures in the form of *failure handling plans*. Failure handling plans allow the developer to 'clean up' following a plan failure, e.g., by specifying that the agent should perform additional actions to 'undo' the effects of internal or external actions performed prior to the failure of a plan. When a plan π in an intention i fails, the *Jason* interpreter searches for a goal g in i that has a relevant

failure handling plan. If one is found, the intention is suspended and a goal deletion event $-!g$ is added to the event list. If no failure handling plan is found, the intention containing π is dropped. Execution of the agent then continues. If a goal deletion event was generated it may be selected at a future cycle and trigger a failure handling plan. If the failure handling plan is applicable in the current context, it is pushed onto the intention i, on top of the failed plan π, and i is unsuspended. This allows the failure handling to plan to inspect the current state of the failed plan (using *Jason* internal actions) and take appropriate recovery steps depending on which external actions were executed prior to failure. For example if the order processing plan fails because there is insufficient stock to process the order, a failure handling plan could send a message to the customer to this effect, and perhaps suggest a similar alternative item that is in stock.

In the case of plans with an `atomic` annotation, this basic scheme is extended in two ways. First, if the plan which gives rise to a goal deletion event $-!g$ has an `atomic` annotation, the intention retains the atomic property and will be scheduled at the next reasoning cycle. Second, the event selection function is over-ridden (including any user customisations) to ensure that the goal deletion event $-!g$ is processed at the next cycle. In effect, the failure handling plan extends the critical section created by the failed atomic plan π, allowing recovery actions to be performed before any other intention is scheduled.

While this scheme allows (atomic) recovery from the failure of an atomic plan, it suffers from a number of disadvantages. First, and most obviously, it requires that the agent developer anticipate possible failures and write appropriate failure handling plans. For external and internal actions which have complex side effects that must be undone for recovery to be possible this is inevitable. However undoing changes made to the agent's belief and plan base as a result of plan execution should not require programmer intervention if the plan fails. Such automated reversion of the belief and plan bases is currently not supported in *Jason*.

Perhaps more surprising, given the current architecture of *Jason*, it can be difficult for the agent developer to cleanly undo the effects of belief or plan base changes in a failure handling plan. For example, if the `dispatch` goal in the `processOrder` plan cannot be achieved, e.g., because it is impossible to deliver the item to the customer's delivery address, or because delivery is handled by an external organisation/agent which rejects the delivery request, the changes to the agent's beliefs to reflect the reduced level of stock and the customer's purchase of the item should be reverted. However, simply reverting the belief changes in a failure handling plan does not remove any belief change events resulting from belief changes made by the failed plan. Moreover, belief deletions and additions made by a failure handling plan result in additional belief change events which may in turn give rise to additional intentions when the atomic failure handling plan finishes execution. (Even using the internal actions provided by Jason which directly manipulate the agent's Java state, there is no obvious way to avoid this.) As a result, if an `atomic` plan fails, any intermediate, potentially inconsistent, belief changes are visible to other intentions. Such belief changes may in turn give rise to new intentions, and attempting to restore consistency to the belief base may compound the problem by generating yet more intentions.

Similarly in Coo-AgentSpeak(L) [7], the `acquisitionPolicy`, which determines whether a retrieved plan is added to or replaces any existing plan for the plan's triggering event, is applied immediately on successful completion of the retrieved plan. If such changes to the plan base occur within the scope of an atomic plan, there is no easy way that they can be undone, as there is no record of the previous state of the plan base.

The cause of these problems is a failure to ensure that changes to the agent's belief and plan bases and event list are only visible to the rest of the agent if the plan succeeds.

4 A New Approach to Atomic Intentions

We argue that changes to an agent's state resulting from execution of an atomic plan should have a transactional 'all or nothing' semantics. If an action in an atomic plan fails, the whole atomic plan should fail and the agent's belief and plan bases and event list should be in the state they were before the execution of the atomic plan. However if an action in an atomic plan succeeds, any updates to the agent's belief and plan bases resulting from the action should be visible to subsequent actions in the atomic plan. If all the steps in an atomic plan succeed, all updates to the agent's belief and plan bases made by the atomic plan should be visible to any subsequent actions in the intention containing the atomic plan and to actions in other intentions, and all events generated by the atomic plan should be added to the event list. We believe that automating failure handling in this way can significantly reduce the load on the agent developer when writing atomic plans, particularly as the facilities available in *Jason* for reverting changes to the agent's belief and plan bases and event list are difficult to use and/or incomplete.

In the remainder of the paper we describe the changes we made to *Jason* to support such a transactional model of atomic plans in *Jason*$^+$. The syntax of plans in *Jason*$^+$ is backwards compatible with the current release of *Jason*: in particular, the syntax of the existing `atomic` annotation is unchanged, only its semantics differs. We stress that the automated support provided by *Jason*$^+$ for handling plan failure in atomic plans is restricted to changes to the agent's belief base, plan library and event queue. As noted above, in general, it is difficult or impossible to automatically revert actions performed in the environment or arbitrary changes to the agent's state (e.g., through sequences of 'compensation actions' applied in reverse order). Any effects of external and internal actions that must be undone on plan failure must be anticipated by the programmer and coded as failure handling plans as at present.

4.1 Delta State

To provide a transactional semantics for atomic plans, we must be able to restore the state of the agent that existed at the beginning of the execution of an intention containing an `atomic` plan in case the plan fails, and to commit the changes in the case in which the intention succeeds. This corresponds to the *atomicity* property of the traditional database 'ACID' guarantees (*isolation* is not an issue, as atomic intentions in *Jason* execute with mutual exclusion with respect to all other intentions; *consistency* is the

concern of the belief update function; and *durability* can be achieved using the *Jason* persistent belief base).

A number of approaches can be taken to achieve atomicity. One approach would be to make a copy of the whole state of the agent (belief-base, plan-library and event queue) at the beginning of the execution of an atomic plan. Changes to the agent's state by actions within the atomic plan are made as normal. If the atomic plan succeeds, the copy of the agent's state is discarded. Conversely, if the plan fails, the state of the agent is restored from the copy. An alternative is to instead store a log of the belief updates performed by the atomic intention, and then revert them (and only them) on plan failure.[3] Depending on the size of the belief base and the number of belief updates made by an atomic plan, recording only the belief update actions may require less space than keeping a copy of the belief base. However care is required to ensure correct behaviour. For example, it is important to ensure that any belief change events resulting from the execution of an atomic plan are removed from the event queue on plan failure. A further variation would be to create a copy of the belief base, plan library and event queue at the start of the intention (as in the first approach), and make changes only to the copy. On success, these changes can be merged back into the main belief base, or otherwise discarded. This provides very strong atomicity and isolation guarantees (corresponding to full 'snapshot' isolation), but clearly comes at a high price in terms of memory overhead and complexity.

These difficulties can be overcome by maintaining a *delta state* of the agent to store uncommitted changes, and by querying the combination of the normal and delta state during the execution of an `atomic` plan. The delta state consists of three components: a *delta belief base*, a *delta plan base*, and a *delta event list*. We describe each of these in turn below.

4.2 Delta Belief Base

The delta belief base records changes to the belief base of the agent made by actions in an `atomic` plan. The delta belief base is initially empty. When a new atomic plan starts executing, the agent registers that it should use the delta belief-base instead of the normal belief-base to apply changes to its beliefs resulting from the atomic plan. If the atomic plan completes successfully, the delta belief base is merged with the normal belief-base of the agent and then cleared. If the atomic plan fails, the contents of the delta belief base are discarded, leaving the normal belief-base untouched.

The delta belief base consists of a list of belief additions (the *add list*) and a list of belief deletions (the *delete list*). If a new belief is added as a mental note in an `atomic` plan, it is simply appended to the add list of the delta belief base. Similarly, if an existing belief is deleted it is appended to the delete list. If an existing belief is modified, e.g., through the addition, deletion or modification of an annotation, the existing belief is appended to the delete list and and the modified version is appended to the add list of the delta belief base. The add and delete lists treat updates in the same way as the

[3] Simply recording (but not applying) changes to the agent's state made by an atomic plan can result in unacceptable performance for belief queries, when uncommitted updates are queried during execution of the plan.

normal belief base, i.e., addition and deletion are done according to the normal rules of *Jason*. For example, if `b(a)[p]` is in the add list, then adding the belief `b(a)[q]` simply adds the new annotation `q` to the existing belief `b(a)` resulting in `b(a)[p,q]`. Similarly for deletion: if the add list contains on `b(a)[p,q,r]`, deleting `b(a)[q]` results in `b(a)[p,r]`, but if the add list contains `b(a)[q]`, deleting `b(a)[p,q]` results in the complete removal of `b(a)[q]` from the add list.

To determine whether a literal is a logical consequence of the combined belief and delta belief bases, the *Jason* `logicalConsequence` method is over-ridden to iterate over both the belief base and the delta belief base. Whenever a query is made on the belief base, the beliefs in the delta belief base add list are first checked, and then those beliefs in the belief base which do not occur in the delete list of the delta belief base. The situation is complicated by the fact that, as in Prolog, the order of beliefs in the belief base is significant. For example, newly added beliefs are returned before any existing beliefs with the same functor, and a belief deletion containing a variable, such as `-b(X)`, deletes only the first matching belief: any other beliefs matching `b(X)` remain in the belief base. It is therefore important that the order in which beliefs are returned from the merge of the belief base and delta belief base is the same as if the updates had been applied to the belief base. To ensure this, beliefs which occur in both the add and delete lists of the delta belief base (i.e., modified beliefs) are returned at the position in the sequence occupied by the belief 'deleted' from in the belief base.

If execution of the atomic plan completes successfully, the changes in the delta belief base are applied to the agent's belief base. As above, updates are applied in order to ensure that the order of entries in the belief base is maintained. If the atomic plan fails, the contents of the delta belief base are simply discarded.

Below, we sketch the *Jason*$^+$ algorithms for adding and deleting beliefs and checking whether a belief (pattern) is present in the belief base.[4]

Algorithm 1 Add a belief b with annotation a to the belief base

procedure ADD-BELIEF((b, a))
  if $(b, a') \in$ *belief-base* $\wedge$ $(b, a') \notin$ *delete-list* **then**
    delete-list $\leftarrow$ *delete-list* $\cup \{(b, a')\}$
    add-list $\leftarrow$ *add-list* $\cup \{copy((b, a'))\}$
  add-list $\leftarrow$ *add-list* $+ (b, a)$

Algorithm 1 first checks if the new belief is in the original belief base and has not already been deleted. If so, it marks the original belief as deleted, and adds a copy of the original belief to the add list. It then merges the new belief into the add list (indicated by $+$). This ensures that a belief is either in the add list or among the non-deleted beliefs of the original belief-base, but not both.

Algorithm 2 first checks the add list (as new elements are checked first) for a matching belief, and if a matching belief is found (indicated by $=_m$) applies deletion accord-

[4] Note that, in the interests of brevity, we do not show how the order of beliefs is maintained and also omit some optimisations.

Algorithm 2 Delete the first belief matching the belief pattern p with annotation a from the belief base

```
procedure DELETE-BELIEF((p, a))
    for all (b, a') ∈ add-list do
        if b =_m p then
            add-list ← add-list − (b, a)
            return
    for all (b, a') ∈ belief-base do
        if b =_m p ∧ (b, a') ∉ delete-list then
            delete-list ← delete-list ∪ {(b, a')}
            add-list ← add-list ∪ {copy((b, a'))}
            add-list ← add-list − (b, a)
            return
```

ing to *Jason*'s belief update rules (which either results in the removal of an annotation or the complete removal of the belief). The search has to be sequential as we are looking for a pattern and not an exact belief. If no belief matching the pattern is found in the add list, it looks for a matching belief amongst the non-deleted elements of the original belief base. If a matching belief is found, then, as in Algorithm 1 we mark the belief as deleted, put a copy in the add list, and then apply deletion according to *Jason*'s belief update rules.

Algorithm 3 Return the first belief matching the belief pattern p with annotation a in the belief base

```
function GET-BELIEF((p, a))
    for all (b, a') ∈ add-list do
        if b =_m p ∧ a' =_m a then
            return (b, a')
    for all (b, a') ∈ belief-base do
        if b =_m p ∧ a' =_m a ∧ (b, a') ∉ delete-list then
            return (b, a')
    return not-found
```

Algorithm 3 first checks the add list for a belief matching the pattern. If none is found in the add list, it looks for it amongst the non-deleted elements of the original belief base. Again, the search has to be sequential as we are looking for a pattern and not an exact belief.

4.3 Delta Plan Base

To support retrieval of external plans at run time, e.g., from another agent via plan exchange or from a library of plans, *Jason*$^+$ also incorporates a delta plan base. The delta plan base functions in an analogous way to the delta belief base. When an intention containing an `atomic` plan is initially selected, the delta plan base is initialised and

the agent registers that it should use the delta plan base to apply changes instead of the normal plan base. Plans which are dynamically loaded or modified as a result of actions in the atomic plan are held in the delta plan base. If the atomic plan succeeds, these plan changes are permanently merged with the agent's plan base and the delta plan base cleared. However if the atomic plan fails, the contents of the delta plan base are discarded.

Jason$^+$ implements a version of the module system described in [4]. Changes to the agent's plan base therefore involve the addition of new modules (containing one or more plans) and/or the modification of plans in an existing module. Below, we briefly describe the functionality of the module system necessary to understand the delta plan base.

Jason$^+$ modules

A *Jason*$^+$ *module* is an encapsulated subset of the functionality of an agent consisting of:

1. a local belief base, containing any beliefs that are private to the module;
2. a plan library, implementing the functionality of the module;
3. a local event queue for belief and goal update events that are local to the module;
4. a list of exported belief and goal predicates;
5. a unique identifier (URI) and a 'short' name for the module;

Each module defines an XML-like namespace [8] identified by a URI which acts as a prefix for all beliefs and goals defined in the module. Agents can therefore be composed from existing modules without the risk of name clashes.

Modules control the visibility of predicates they define by *exporting*. Each module contains zero or more `export` directives, which specify the predicate symbols that are visible outside the module. Each export directive is of the form

```
{export predicateNameA/arity, predicateNameB/arity, ...}
```

where `predicateNameA/arity`, `predicateNameB/arity` etc. are the functor names and arities of the predicates exported from the module. In the case of predicates of no arguments, the `/0` can be omitted. Also, for convenience, the wildcard character `?` can be used in place of both the predicate name and arity, and will match any predicate name and arity respectively. For example {`export foo/?`} will export all predicates with the functor `foo` of any arity. Predicates which are not exported from a module are considered implementation details of the module and are not visible outside the module. Each module therefore effectively defines its own local belief base containing its non-exported beliefs, and any belief additions or revisions of non-exported beliefs by plans in a module are applied to its private belief base. For example, a module encapsulating the order processing plan shown in Figure 2 might export the `purchased/3` predicate, but not `stock/2`. The encapsulation provided by modules also applies to events arising from belief and goal addition and deletion (so-called 'internal events'). Each module effectively has a private event queue, and events arising from changes to

non-exported beliefs or goals are added only to that module's event queue. Events arising from changes to the agent's main belief or goal base (i.e., from exported beliefs or goals) are visible to all modules, as in the current *Jason* implementation.

Agents (and modules) access the functionality of a module by *importing* the module. Each *Jason*$^+$ source file may contain zero or more `import` directives. Each import directive specifies the URI of a module and a local or 'short' name that can be used within the source file in place of the full URI.[5] The URI specifies the location of the file containing the source code of the module, and in the current implementation can be a relative or absolute path on the local file system, or a URL. The URI forms the module's fully qualified name, which is prefixed to every term in the module when the code in the module is parsed. The short name expands into the full URI reference for the imported module, and can be used to conveniently refer to the exported beliefs and goals of an imported module in the code of the importing module.[6] For example, the import directive

```
{import order = http://example.com/orderProc.aslm}
```

imports the module with URI `http://example.com/orderProc.aslm` and defines the short name `order`, allowing predicates exported by the module to be referred to using, e.g., `order::purchased(Customer, Item, Num)` (which expands to `http://example.com/orderProc.aslm::purchased(Customer, Item, Num)`). Goals defined in the module can be accessed in a similar fashion, e.g., `!order::invoice(Customer, Item, Num)`. A null short name is taken to refer to predicates defined in the agent itself. For example, the term `::customer(X)` in code for a module refers to the belief `customer(X)` defined in the agent. The standard *Jason* internal actions (e.g., `.send`), reserved terms used in annotations (e.g., `source`), operators, strings, numbers and the literals `true` and `false` are not prefixed with the module URI and are visible in all modules.

Each module may therefore access the beliefs, goals and events defined in the agent. Plans within a module can respond to events that occur either within that module, events exported from an imported module, or evens in the agent event queue. Likewise, the context of a plan can refer to belief predicates that are private to the module in which that plan is defined, beliefs exported from an imported module, or beliefs defined in the agent. As noted above, only those predicates which are exported by a module can be accessed by a module that imports it: attempting to access a predicate which is not exported (using either the short or fully qualified name of the module as a prefix) results in a parse error. Multiple modules can import the same module, but the import directive ensures that the module is only loaded once. Each agent therefore contains a single copy of an imported module. References to a module are shared between all modules that import it. Different modules can therefore communicate with each other through the exported beliefs and goals of a shared module or the beliefs and goals defined in the agent itself. For example, the order processing plan(s) and their associated beliefs may be encapsulated in a `http://example.com/orderProc.aslm`

[5] The URI parameter is optional. If it is omitted, it defaults to a file with the same name as the short name plus the the standard `.aslm` extension in the same directory as the agent.

[6] Short names are also used by the *Jason*$^+$ mind inspector when displaying code during debugging.

module which exports the `order/3` and `purchased/3` predicates and imports a module containing plans which handle dispatch of purchased items which in turn exports the `dispatch/3` predicate (see Figure 3).

```
{export order/3, purchased/3}
{import delivery = http://example.com/dispatchProc.aslm}

/* Initial stock */
stock(widget, 1).

@processOrder[atomic]
+!order(Item, Num)[source(Customer)] : stock(Item, Stock) <-
    Num <= Stock;
    -+stock(Item, Stock - Num);
    +purchased(Customer, Item, Num);
    !delivery::dispatch(Customer, Item, Num).
```

Fig. 3. Example *Jason*$^+$ module.

Dynamically importing modules

Plans (modules) may be dynamically loaded at run time using the `.import` internal action. The `.import` action takes as argument the URI of the module to be loaded, and, optionally, a short name to be used for debugging. The specified file or URL is located, read and parsed. After the module's source file is successfully parsed and prefixed with the module's unique identifier, the contents of the module (beliefs, goals and plans) are dynamically added to the agent's belief and plan bases, and the appropriate events generated. Once the import finishes, execution of the plan continues. If an `.import` action fails, e.g., if the file could not be located or read, the import is aborted and a *Jason*$^+$ action execution failure is generated. This allows failure handling plans to dynamically handle module load failures.

The `.import` action can be used directly in plans or its underlying Java implementation can be used to redefine the `selectOption` function as in Coo-AgentSpeak(L)[7]. Using modules for plan retrieval and exchange eliminates the risk of namespace clashes inherent in the Coo-AgentSpeak(L) model, leading to more robust multiagent systems. Exchanging URIs rather than providing the source of plans in `TellHow` messages, is a departure from the Coo-AgentSpeak(L) model. However we would argue that it is consistent with a more "modular" approach to agent design, which makes greater use of pre-existing library code.

Dynamic importing and the delta plan base

Dynamically loading modules or modifying plans in an `atomic` plan makes use of the delta belief and plan bases and the delta event list:

1. each of the initial beliefs listed in the module are added to the agent's delta belief-base and a belief addition event is added to the delta event list for each belief;
2. a new goal achievement event is generated as new focus for each of the initial goals listed in the module and added to the delta event list; and
3. each plan defined in the module is added to the agent's delta plan base.

As with the delta belief base, the delta plan base consists of a list of plan additions (the *add list*) and a list of plan deletions (the *delete list*). New plans are simply appended to the add list in the same way as plans are added to the normal plan base. The addition of a plan with the same label as that of an existing plan in the plan base is interpreted as a modification of that plan. The plan currently in the agent's plan base is copied to the delete list of the delta plan base and the modified version of the plan (including any modified plan annotations) is appended to the add list. Any subsequent modification of the plan replaces the version of the plan in the add list. If a plan in the agent's plan base is deleted (e.g., using the internal action `.remove_plan`), it is added to the delete list. The deletion of a plan which was added during the execution of an `atomic` plan removes the deleted plan from the add list. Deleting a modified plan causes the modified version to be removed from the add list and the original, unmodified, plan to be removed from the delete list.

Plan retrieval is handled in a similar way to queries of the belief and delta belief bases. First the plan base is checked for applicable plans, ignoring any plans which appear in the delete list of the delta plan base, and then the add list of the delta plan base is checked for applicable plans. As with beliefs, care is required to ensure that the order in which plans are returned is the same as would be the case if the updates had actually been applied to the agent's plan base. Plans which occur in both the add and delete lists of the delta plan base (i.e., modified plans) are returned at the position in the sequence occupied by the plan 'deleted' from in the plan base.

The delta plan base supports the same *Jason*$^+$ transition system events as the delta belief base. If the atomic plan completes successfully, the delta plan base is merged with the normal plan base and then cleared. To merge the contents of the delta plan base with the normal plan base, the delete list is used to achieve the same ordering of plans that would have been produced during the execution of a non-atomic intention. Those plans in the normal plan base which have been modified are replaced with the copy stored in the add list of the delta plan base. At the same time, plans which have been removed are also deleted from the normal plan base. Any newly added plans are then inserted either at the beginning or the end of the plan base, as appropriate. If execution of an atomic plan containing an `.import` (or `.remove_plan`) action fails, either while executing the imported plan or subsequently, the contents of the delta plan base are discarded. Note that this partially pre-empts the role of the `acquisitionPolicy` in [6, 7] as implemented in Coo-AgentSpeak(L). It could be argued that we should not discard a retrieved plan if failure was not attributable to that plan. However the approach adopted here is consistent with our transactional view of atomic intentions. If the atomic plan succeeds, then the `acquisitionPolicy` is applied, which may result in the retrieved plan being permanently added to the agent's plan base, or discarded.

4.4 Delta Event List

The delta event list contains internal belief change events generated by updates to the delta belief base. The delta event list also contains all internal goal change events generated by the atomic plan which could result in the creation of a new intention (denoted by $!!g$ achievement goals in *Jason*). Goal addition events corresponding to subgoals of the atomic plan are added to the event list in the normal way.

If execution of the atomic plan completes successfully, the changes in the delta event list are applied to the agent's event list. If the atomic plan fails, the contents of the delta event list are simply discarded.

5 Conclusion

Most BDI-based agent programming languages support parallel execution of intentions, either as their default execution strategy, or as an option. Several languages also provide some form of 'atomic' construct. For example, in addition to the `atomic` plan annotation in *Jason* [1] considered here, 2APL [9] has a 'non-interleaving operator' which prohibits arbitrary interleaving of plans. Likewise, some form of 'failure handling' facility can be found in most mature BDI-based agent languages and platforms. For example, 2APL [9, 10] provides plan revision rules which can be applied to revise plans whose executions have failed, JACK [11] and SPARK [12] provide failure methods and/or meta-procedures which are triggered when plan execution fails, and in [13, 14] features are proposed for aborting and suspending tasks in the context of the CAN abstract agent programming language. Taken independently, such features can simplify the development of more robust agents. However, to the best of our knowledge, how these features (should) interact has received little attention in the literature.

In this paper we have presented *Jason*$^+$, an extension to the *Jason* agent programming language which adopts a transactional approach to atomic plans. If an action in a *Jason*$^+$ atomic plan fails, the whole atomic plan fails and the agent's belief and plan bases and event list are left in the state they were before the execution of the atomic plan. However if an atomic plan succeeds, all updates to the agent's belief and plan bases made by the atomic plan are visible to any subsequent actions in the intention containing the atomic plan and to actions in other intentions, and all events generated by the atomic plan are added to the event list. *Jason*$^+$, also implements a version of the module system described in [4] which is used to encapsulate retrieved plans. With the exception of minor changes required by the module system, the syntax of *Jason*$^+$ is backwards compatible with the current release of *Jason*.

We have argued that the extensions implemented in *Jason*$^+$ offer a more predictable semantics for atomic plans in the face of plan failure and can reduce the load on the agent developer by automating simple cases of failure handing, leading to the development of more robust agent programs. In many applications, atomic plans tend to be short and generate only a relatively small number of belief and plan updates. In such cases, the impact of the extensions on the performance of the *Jason* interpreter is minimal. For simple atomic plans consisting of fewer than 10 non-recursive actions, the run time memory usage of *Jason*$^+$ is approximately 0.25% greater on average than for an

equivalent implementation in *Jason*, and the CPU overhead is approximately 1.5% on average.[7]

While our approach has been developed in the context of *Jason*, we believe that the transactional approach to atomic intentions adopted by *Jason*$^+$ can be usefully applied to other BDI-based agent programming languages. More generally, our work raises interesting questions about the ways in which universal programming problems such as atomicity, synchronisation and modularity should be addressed in a BDI context. For example, to what extent should the parallel execution of intentions share state? We believe deeper consideration of these issues offers a fruitful avenue for future research, and is necessary for BDI-based agent programming languages to develop into robust software development platforms.

In the current implementation of *Jason*$^+$, nested atomic actions (when an `atomic` plan spawns another `atomic` plan via sub-goal) reuse the existing delta state. This is reasonable in situations where the failure of a sub-plan ultimately results in the failure of the parent/root atomic intention. However in many cases it is possible to recover from the failure of a sub-plan, and in future work we plan to extend our current approach to allow nesting of delta belief bases. Another, related area of future work, is to extend our approach to atomic intentions in *Jason*$^+$ to allow support for *transactional* plans and intentions. For example, this could be implemented by associating a different delta belief and plan base with each intention that contains a plan with a `trans` annotation. In those cases where external actions in plans do not interfere, we believe such an approach would provide the advantages of the cleaner approach to plan failure found in *Jason*$^+$ atomic plans while avoiding the problem of lack of responsiveness inherent in the use of the `atomic` construct.

6 Acknowledgements

We would like to thank Rafael Bordini and Jomi Fred Hübner for helpful discussions on the current implementation of *Jason*.

References

1. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons Ltd (2007)
2. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: MAAMAW '96: Proceedings of the 7th European workshop on modelling autonomous agents in a multi-agent world, Springer-Verlag (1996) 42–55
3. Bordini, R.H., Hübner, J.F., Vieira, R.: ***Jason*** and the Golden Fleece of agent-oriented programming. In Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Springer-Verlag (2005)
4. Madden, N., Logan, B.: Modularity and compositionality in Jason. In Braubach, L., Briot, J.P., Thangarajah, J., eds.: Programming Multi-Agent Systems: 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009. Revised Selected Papers. Number 5919 in LNAI, Budapest, Springer (2009) 237–253

[7] The tests were run on a 2.33 GHz PC with 6 GB of memory.

5. Bordini, R., Bazzan, A.L.C., de O. Jannone, R., Basso, D.M., Vicari, R.M., Lesser, V.R.: AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In: Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (AAMAS'02), New York, NY, USA, ACM Press (2002) 1294–1302
6. Ancona, D., Mascardi, V.: Coo-BDI: Extending the BDI model with cooperativity. In Leite, J., Omicini, A., Sterling, L., Torroni, P., eds.: Declarative Agent Languages and Technologies (DALT 2003). Volume 2990 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 1270–1270 10.1007/978-3-540-25932-9_7.
7. Ancona, D., Mascardi, V., Hubner, J.F., Bordini, R.H.: Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04), Washington, DC, USA, IEEE Computer Society (2004) 696–705
8. Bray, T., Hollander, D., Layman, A., Tobin, R.: Namespaces in XML 1.0, second edition. Technical report, W3C (2006) `http://www.w3.org/TR/2006/REC-xml-names-20060816`.
9. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16** (2008) 214–248
10. Dastani, M., Meyer, J.J.C.: A practical agent programming language. In Dastani, M., Fallah-Seghrouchni, A.E., Ricci, A., Winikoff, M., eds.: Proceedings of the Fifth International Workshop on Programming Multi-agent Systems (ProMAS'07). Volume 4908 of LNCS., Springer (2008) 107–123
11. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK intelligent agents - components for intelligent agents in Java. AgentLink Newsletter (2) (January 1992) 2–5
12. Morley, D., Myers, K.: The SPARK agent framework. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04), Washington, DC, USA, IEEE Computer Society (2004) 714–721
13. Thangarajah, J., Harland, J., Morley, D., Yorke-Smith, N.: Aborting tasks in BDI agents. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'07), Honolulu, HI (May 2007) 8–15
14. Thangarajah, J., Harland, J., Morley, D., Yorke-Smith, N.: Suspending and resuming tasks in BDI agents. In: Proceedings of the Seventh International Conference on Autonomous Agents and Multi Agent Systems (AAMAS'08), Estoril, Portugal (May 2008) 405–412