Modularity and compositionality in Jason

Neil Madden and Brian Logan

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

Abstract. In this paper, we present our experiences using the *Jason* agent-oriented programming language to develop a complex multi-agent application. We highlight a number of shortcomings in the current design of the language when building complex agents, and propose revisions to the language to allow the development of modular programs that facilitate code reuse and independent development. In particular, we propose a mechanism for modular construction of agents from functionally encapsulated components, and discuss alterations to the belief base language to enable more robust software engineering.

1 Introduction

Agent-oriented programming languages, such as *Jason* [1] and 2APL [2], have been the subject of a great deal of research and development in recent years. Building on the foundations of logic programming and theories of agency—in particular, the BDI (belief-desire-intention) model—they aim to raise the level of abstraction used in constructing complex modern software applications. However, while they have been successfully applied in a number of interesting problem domains, the literature contains relatively few reports of attempts to apply such languages to large-scale software development efforts.

In this paper we present our experiences of applying the agent-oriented programming language *Jason* to the development of a large-scale multi-agent system consisting of (relatively) complex *witness narrator agents* which report on events occurring in online persistent game environments [3]. Witness-narrator agents are embodied in a virtual environment and observe and narrate activities occuring within that environment. The deployed system consisted of a team of 100 agents which reported on events in a medium-scale persistent virtual environment over a period of several weeks. The agents had to handle a number of complex tasks during this period, including activity recognition, multi-agent coordination, generation of prose stories describing activities, and interaction with human participants. The system makes use of a range of technologies, including ontological reasoning, plan and activity recognition, and multi-agent coordination and teamwork. The architecture of the agents is organised as a collection of functionally encapsulated 'capability' modules to handle distinct tasks, such as low-level activity recognition, editing of reports from multiple agents, and generation of prose text for a particular output medium.

Our experiences with *Jason* indicate that it is a useful and flexible language which provides a clean, high-level approach to defining complex agent logic. However, we

also found the language to be lacking in some respects, particularly in relation to the development of more complex agents. In this paper we describe the problems that we encountered, and propose revisions to the language to allow the development of modular programs that facilitate code reuse and independent development. In particular, we propose a mechanism for modular construction of agents from functionally encapsulated components, and discuss alterations to the both the belief base language and plan execution to enable more robust software engineering.

The remainder of the paper is organised as follows. In section 2 we first give a brief introduction to the *Jason* programming language. We then discuss two related areas where we experienced some problems with the current design of *Jason*, and argue that these problems can be addressed by adding a module mechanism to *Jason*. In section 4 we look at the choice of Prolog for the default belief base language in *Jason*, discuss some problems that this presents for modular agent construction, and describe an alternative based on Datalog. In section 5 we then look at the requirements for general modular construction of agents, and propose a simple module system that addresses these requirements. Lastly, we conclude with a look at related work (section 6) and some general conclusions in section 7.

2 Jason

Jason [1] is a Java-based interpreter for an extended version of AgentSpeak(L). AgentSpeak(L) is a high-level agent-oriented programming language [4] which incorporates ideas from the BDI (belief-desire-intention) model of agency. The language is loosely based on the logic programming paradigm, exemplified by Prolog, but with an operational semantics based on plan execution in response to events and beliefs rather than SLD resolution as in Prolog. *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). The syntax of *Jason* plans essentially consists of a single triggering event (such as a goal or belief addition or deletion, or a percept), a belief context pattern, and then a sequence of actions to perform if the plan is selected.[1] During execution, *Jason* first processes any events and updates the 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 allowed to perform an action, before the cycle repeats. The complete cycle is shown in Fig. 1, adapted from [1], Chap. 4.

The process of constructing software using *Jason* proceeds at two levels. At a high-level, the problem domain is broken down in terms of a society of autonomous and cooperating (or competing) agents. Agents in *Jason*, as in other agent-oriented programming languages, are autonomous encapsulated processes that communicate with each other by sending messages (speech acts). In the BDI paradigm, programming in

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

1. Perceive the environment;
2. Update the belief base;
3. Receive communication from other agents;
4. Select socially acceptable messages;
5. Select an event;
6. Retrieve relevant plans;
7. Determine applicable plans;
8. Select one applicable plan;
9. Select an intention for execution;
10. Execute an action.

Fig. 1. *Jason* interpreter cycle.

the large thus involves decomposition of the system into entities (agents) to which belief and other propositional attitudes can most naturally be ascribed. At a lower level, individual agents are authored in terms of their beliefs and goals, and plans which specify how to achieve the agent's goals and how to react to events. The primary mechanism for structuring a *Jason* program in the small is therefore the plan.[2] Issues arise when a natural decomposition of the system into (intentional) agents results in entities with large numbers of plans. In such cases, a modular approach to agent development is often desirable.

3 Problems of Modular Agent Programming

An agent-oriented approach to constructing large software has several advantages, such as encouraging separation of concerns, so-called loose coupling between components, and extending relatively naturally to a distributed environment in which messages are sent over a network to other remote agents. *Jason* provides good support for constructing multi-agent systems at this level, providing natural and easy to use speech-act based communications, and abstracting away from many of the details of the underlying infrastructure. At the level of constructing an individual agent, the BDI model of *Jason* and the sophisticated plan and belief base facilities it provides allow the developer to express complex logic in a concise and clear fashion. However, our experience with using *Jason* to develop a large and complex application indicates that there is a gap between these two levels that becomes more apparent as individual agents grow in complexity. In particular, *Jason* lacks any mechanism for decomposing an individual agent into constituent components or modules. Figure 2 shows the overall architecture of one of the agents in the system we developed, known as a *witness-narrator agent*. The witness-narrator agents have a quite complex internal structure, consisting of a number of different competences ('capabilities') that are conceptually independent of one another to a large degree and which communicate only through clearly defined interfaces. Each capability module consists of a set of *Jason* plans, along with some beliefs and

[2] Although traditional Prolog-style rules in the belief base can also form a significant part of the codebase.

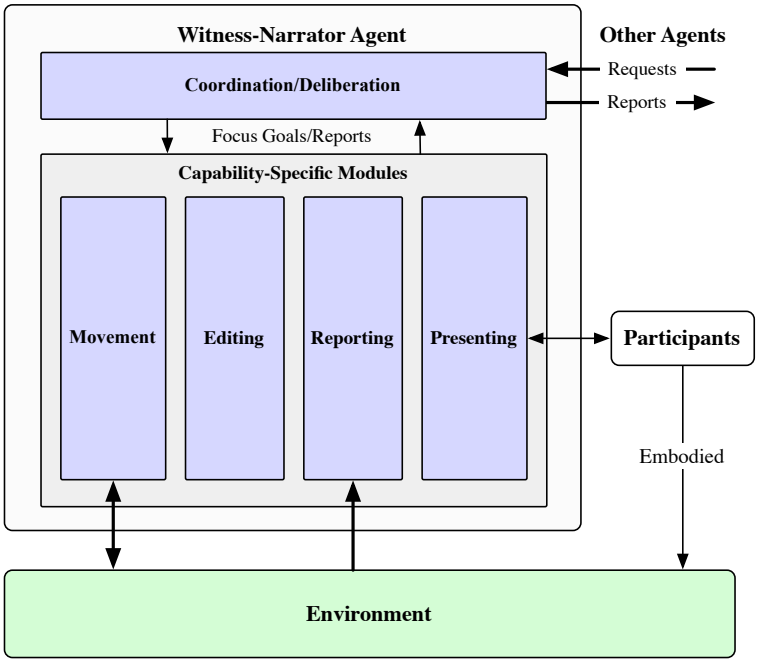

Fig. 2. Architecture of witness-narrator agent.

rules, that are used to implement that particular competence. For instance, the reporting module contains plans for detecting and recording activity occurring within an environment, while the presenting module has plans and rules for formatting a report for a particular output medium (HTML, Atom [5]). A separate coordination module handles communication and interaction with other agents, such as team formation. The current implementation of witness-narrator agents in *Jason* implements such modules simply as a set of files which are included one at a time into a main agent program. There are a number of drawbacks with this approach:

- there is the possibility of name clashes between belief and goal relations defined in separate modules;
- it may be desirable for plans in different modules to react to the same triggering event;
- the order in which modules are included can have surprising effects on the execution of the agent.

The first problem is one of namespace management, and can be addressed to a certain degree by adopting coding guidelines to ensure that beliefs and goals in separate modules have different names, for instance using a simple unique prefix for each module. However, this issue becomes more important when we consider third-party modules that are developed independently by different organisations. While guidelines can be adopted to try to ensure that unique prefixes are chosen (such as incorporating the insti-

tution name into the prefix), such approaches can be cumbersome to use. The possibility that agents may dynamically locate and acquire new modules at runtime (e.g., OWL ontologies) also suggests that good namespace management should be built in to the language itself. This could be achieved in a straightforward way by using a mechanism similar to that adopted for XML [6], in which components of an agent are associated with an unique Uniform Resource Identifier (URI), and in which short string prefixes can be assigned to these URIs within a source file. Such a scheme presents a good combination of flexibility, safety, and ease of use.

The second and third drawbacks present more serious problems. The semantics of plan execution in *Jason* require a single plan to be selected to respond to any particular triggering event. This prevents multiple modules from responding to the same event in different ways, and introduces a form of implicit coupling between modules that must be explicitly resolved by the agent developer. For instance, in the development of the witness-narrator agents, certain events were of interest to both the reporting and general movement modules. If an agent is attacked, for example, this is both a cause for the movement module to take action (to evade the attack), but also presents a potentially interesting event that the reporting module may want to record. In the current implementation, this is achieved by having one module handle the event, and then to generate a secondary event purely for the other module to be notified. This requires both explicit cooperation between the modules (and corresponding effort from the developer), and also introduces extra code whose purpose can be obscure to a reader of the program.

The third problem stems partly from the second, in that if two modules attempt to react to the same triggering event (and the developer hasn't noticed this), then which plan gets to run depends on the plan selection function, which by default is based on the order in which plans are defined. This can lead to some surprising and difficult to understand behaviour where one plan appears not to be triggering correctly as a consequence of another plan in an entirely unrelated module. The problem also occurs if one module included earlier in the sequence includes a plan that effectively matches many or all triggering events, for instance if an unbound variable is used for the trigger. This then takes precedence over any subsequent plans introduced by later modules.

These problems can be seen as stemming from the overall problem of how to construct an agent by composing independent functional components. Ideally, an agent could be constructed by glueing together such independent reuseable modules. The most important property that such a module system should address is that the behaviour of an agent composed of separate modules should not depend on the order in which those modules are composed. In other words, we would like composition of modules to be both associative and commutative, so that the order and combination of modules does not effect the resulting behaviour. This property is particularly important in an event-driven architecture such as *Jason*, where plan execution in response to events can sometimes be difficult to predict. Minimising effects caused by code refactoring helps to reduce the opportunities for confusion when constructing sophisticated software.

The requirements for associative and commutative composition of modules have a number of implications for the design of *Jason*. Firstly, we must ensure that the composition of beliefs from all included modules has the same meaning, and produces the same runtime inference behaviour, no matter in what order those beliefs are added to

the belief base of the agent. This has an impact on the choice of belief representation language and reasoning strategy employed in the belief base. Similar considerations must be taken into account for goals. Secondly, in order to minimise conflicts between modules, it is important to support encapsulation of beliefs, goals and plans that are internal implementation details of a module, while exposing those that form part of its interface. This includes preventing simple name clashes, but also more important issues of belief and goal scope and visibility.

4 Belief Base Compositionality

One of the enhancements of *Jason* over AgentSpeak(L) is the support for a substantial subset of Prolog in the belief base language, including backward-chaining rules. This considerably increases the power of the language, and allows for succinct descriptions of some problems, while also allowing the *Jason* developer to take advantage of the large amount of existing Prolog code. However, despite these advantages, it is not clear whether Prolog is in fact the most appropriate choice for a belief language.

- As already noted, the order in which clauses are added to the belief base has an explicit effect on execution in Prolog (and hence *Jason*), with clauses defined earlier in a program having precedence over later clauses. In some cases, reversing the order of two clauses can lead to incorrect behaviour or even nontermination of a previously correct definition. This clearly violates the requirements for compositionality of beliefs.
- The backtracking execution strategy of Prolog may not be the most efficient way of handling large numbers of beliefs, particular when these may be stored in a relational database system, or other persistent store. For example, the agents used in our software used a MySQL database for persistent beliefs (archives of generated reports and previous activity), and over the several weeks the system was running, acquired many thousands of ground facts.
- Prolog is a computationally complete language, capable of expressing nonterminating algorithms. This is an undesireable property for a belief language, the purpose of which is largely to perform limited inferences to determine if a plan is currently applicable. The possibility that such a belief context check may in fact not terminate has consequences for the rest of the plan execution semantics, and indeed could entirely halt the agent or even the entire MAS depending on the implementation and infrastructure in use.

The sensitivity to belief addition order is particularly important for agents in dynamic environments, where the agents are continually acquiring (and possibly revising) their beliefs over time. This is also a concern with regard to the modularity issues discussed in the previous section: if two modules add facts or rules to the same belief relation, we would like the order in which they are added to not affect the resulting behaviour. For example, the witness-narrator agents combined default rules for classifying observations together with dynamically acquired knowledge regarding individuals. For instance, an agent may have a default rule for determining whether an animal can fly, perhaps by reasoning about its anatomy or species, but then can also record instances

where it has directly observed individuals flying. The default rule is likely to be defined when the agent is created, whereas the specific instances are added to the belief base as they are observed. In *Jason*, depending on the belief-base implementation, this can result in the observations being ordered after the default rule.[3] Due to the execution strategy of Prolog, this will lead to the inefficient and surprising behaviour that the agent will spend time trying to infer if a particular individual is capable of flight when it already has an explicit fact recording this information in its belief base. Another area where the use of Prolog caused some difficulties was in the translation of ontological rules from an OWL ontology into equivalent belief base rules. While a number of such translations have been described in the literature for Prolog-like rule languages (in particular, Datalog), the details of the translation for Prolog itself are complicated by the sensitivity to ordering of rules, and naïve translations can easily result in rules that do not terminate on certain inputs.

In the implementation of the witness-narrator agents, we worked around these problems by providing a custom persistent belief base implementation which implemented slightly different semantics to that of Prolog, by always preferring ground facts to rules, regardless of the order in which they were added to the belief base. This was sufficient to ensure correct operation of the software in a wide range of cases, but the possibility for encountering a non-terminating query could not be entirely ruled out.

4.1 Datalog as a Belief Base Language

A better solution to the problem would be to replace Prolog with a more appropriate knowledge representation language. There are a number of candidates, including restricted versions of Prolog designed for interfacing with relational databases, such as Datalog [7], or a description logic based language [8], such as OWL. Current description logic languages, however, are unable to express many interesting rules that are (easily) expressible in Datalog. In addition, restricting the belief base language to unary and binary predicates (concepts and roles, respectively) makes some problems rather clumsy to express, and can lead to difficulties keeping track of all the information associated with an individual, e.g., when performing belief revision.

Datalog offers many of the advantages of Prolog (the entire belief base used in our witness-narrator agent framework could have been expressed in Datalog with very few changes) while affording more efficient implementation. In particular, the properties of Datalog that make it suitable as a belief language include:

- the order of clauses in a Datalog program does not effect the semantics of query answering;
- the limitations on the language allow all queries to be answerable in polynomial time;[4]

[3] The description of belief updates in the *Jason* documentation and formal semantics [1] state only that a belief is 'added' to a belief base, and that subsequent to this addition the belief base entails the new belief; the ordering of this belief wrt. the previous beliefs is not specified. In practice, the default in-memory belief base adds dynamically acquired beliefs before existing beliefs, whereas the supplied JDBC (database) belief base we were using did not implement this behaviour.

[4] Although this isn't true for various extensions to support negation or disjunctive heads in rules.

– more efficient query answering, particularly for larger sets of beliefs.

As Datalog was designed to be a database query language, it should also provide better support for large belief bases, such as those backed by a relational database management system. The non-recursive subset of Datalog has a natural translation into the relational algebra, allowing for efficient and direct compilation of Datalog belief rules into equivalent SQL queries or views. As a further benefit, there also exist translations from various description logics into Datalog. The use of Datalog as a belief base language would therefore go some way towards supporting efficient ontological reasoning in *Jason*, in particular supporting efficient ABox queries over large ontologies.

5 Encapsulating Beliefs, Goals and Plans

The problems described in section 3 indicate that some mechanism for modular decomposition of individual agents is needed in *Jason*. Such a mechanism could take a number of forms, ranging from a relatively simple module or namespace mechanism, up to a complex object-oriented solution, complete with component instantiation, inheritance, and encapsulation. It could be argued that the agent abstraction could also be used at this level: an individual agent in a society being itself composed of a society of simpler agents. Such an approach is intuitively appealing, but it is not clear whether the advantages of agent-oriented programming in the large also hold for development of agents themselves. In addition, the capabilities within our system do not naturally fit with the usual notion of an 'agent', and it seems unnatural to try and shoe-horn the architecture into layered societies of agents. In this section we examine the requirements and desireable features of a modular approach to building agents, and tie these to the specific agent-oriented programming language, *Jason*.

For *Jason*, the requirements for a module system are relatively modest, as the agent level already provides many of the more sophisticated features required for agent development, such as dynamic instantiation of agents and communication interfaces. From our experience, the main requirements for a module system in *Jason* would be as follows:

1. to simplify the reuse of software components by allowing functional units of code, including beliefs, goals, and plans, to be encapsulated into independent modules;
2. to provide a mechanism for avoiding name clashes between goals and beliefs from separate modules;
3. to allow each module to respond independently to events;
4. to provide a mechanism for controlling which beliefs and events are visible to other modules, and which are private to a particular module;
5. to ensure, as far as possible, that the order in which modules are included in an agent has no effect on the resulting behaviour of the agent.

These requirements are sufficient to address the immediate problems that were experienced during construction of the witness-narrator agent system. We do not consider more advanced functionality, such as parameterisation of modules, instantiation of multiple instances of a particular component, or customisation of plan selection or intention execution functions.

5.1 A Module System for *Jason*

Based on the requirements outlined above, we sketch a proposal for a module system for *Jason*. A *Jason module* consists of the following elements:

1. a local belief base, containing any beliefs that are private to the module;
2. a local goal base;
3. a plan library, implementing the functionality of the module;
4. a local event queue, for belief and goal update events that are local to the module;
5. a list of exported belief and goal predicates;
6. a unique identifier (URI) that acts as a prefix for all belief and goal symbols in the module, based on XML namespaces;
7. a mapping from simple string prefixes to imported module URIs.

A module is therefore a subset of the functionality of an agent, encapsulated into a functional unit, along with an URI for identification. An agent is then defined as a composition of modules, together with an interpreter based on the original *Jason* BDI interpreter. Composition of modules within an agent is a flat one-level hierarchy, with the agent as the root. Nested sub-modules are not permitted in this scheme. This approach greatly simplifies the treatment of beliefs and events, while still addressing all of the requirements that we have described. This implies that there is only a single copy of each module within each agent, and references to that module are shared between all other modules that import it.

We adopt the XML approach to namespaces [6], where each module is considered as a separate namespace, identified by a URI. To ease the use of such a system, we also adopt the XML mechanism of allowing each module to declare a set of simple string prefixes that expand into the full URI reference for another module. Thus the belief predicate symbol `foo:fatherOf` would expand by looking up the prefix 'foo' in the current module's prefix mapping and substituting it for the corresponding URI, e.g., into `http://example.com/foo#fatherOf`. This mechanism could be layered on top of *Jason*'s existing annotation mechanism, so that the belief actually expands into a literal like `fatherOf(...)[module("http://example.com/foo")]`. Goal symbols are scoped in a similar fashion, such that a goal like `!nwn:travel(Destination)` is expanded into `!travel(Destination)[module("http://example.com/nwn")]`. The URI used for a module, as well as the prefix mappings used by that module, could be defined in a *Jason* source file using simple directives, e.g.,:

```
{ module("family", "http://example.com/family.asl");
  import("friend", "http://example.com/friend.asl");
  export(["fatherOf/2","brotherOf/2","!birthday/1"]); }
```

This example syntax declares that the source file implements a module that is identified by the URI `http://example.com/family.asl`, and which imports a 'friend' module with a similar URI. Both modules are given simple string names that can be used within the source file in place of the full URIs (here, the strings 'family' and 'friend' are used respectively). Note that the string prefixes are just a convenience within this source file, and only the URIs are significant outside of the module. The `import`

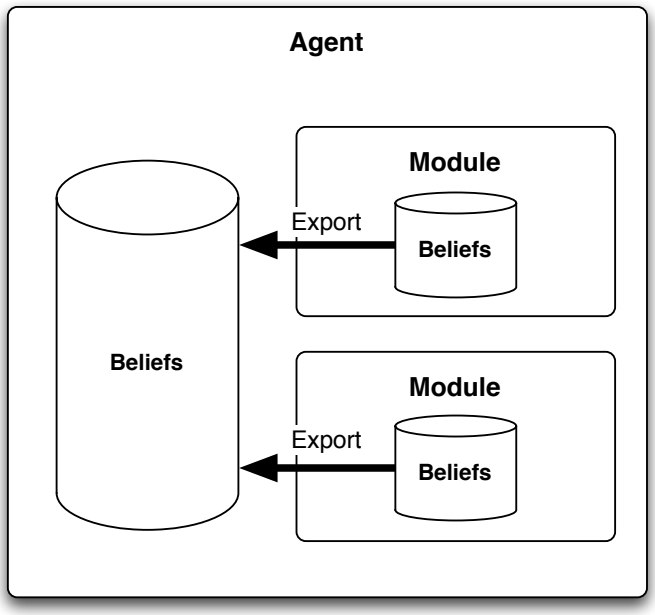

Fig. 3. Scoping of beliefs within modules with export.

directive would also ensure that the specified module is loaded (once) into the agent, if it has not already been loaded[5]. Imported modules are not nested, but conform to the flat module hierarchy. Finally, an `export` directive specifies which belief and goal predicates should be exported from this module, as discussed below.

Each module is considered to contain its own independent belief base. By default, any belief additions or revisions performed by a module are applied to its own local belief base. A mechanism is provided to override this behaviour, by allowing some beliefs (and goals) to be declared as *exported*. Exported beliefs are not added to the module's own belief base, but are instead delegated to the belief base of the agent itself, as shown in Fig. 3. The same approach is used for goals, in that each module has a private goal base, while exported goal symbols are added to the main agent's goals. This partitioning of beliefs and goals into module-private areas also has implications for the generation and propagation of events arising from belief and goal addition and deletion (so-called 'internal events'). A belief or goal predicate that is considered to be private to a module can be seen as an implementation detail of that module, and so events relating to beliefs or goals matching that predicate should not be visible outside of that module. Otherwise, other modules might become reliant on these events, and therefore subject to errors if those implementation details are changed in future updates to a module. However, it is clearly desireable for a module to be able to respond to events arising from changes in its own internal state. We therefore also include a mechanism

[5] We do not specify how module URIs are used to locate implementations.

for scoping of events on a per-module basis. Each module has a private event queue. Events arising from changes to beliefs or goals in a per-module belief or goal base 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 added to the agent's main event queue, as in the current *Jason* implementation.

Each module inherits the beliefs, goals and events of the main agent interpreter. This means that plans within a module can respond to events that occur either within that module, or in the main 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, or those which are added to the main agent belief base. Importing one module into another does not expose the private beliefs or goals of the imported module, but merely ensures that the module is loaded into the agent and sets up a convenient short name for it within the context of the importing module.

5.2 Interpreter Cycle Changes

The changes we have described to incorporate modules into *Jason* imply a number of changes to the *Jason* BDI interpreter cycle, shown previously in Fig. 1. We here describe the changes to the interpreter cycle that are required for our proposal. Items without comments are assumed to be identical to the current *Jason* implementation.

1. *Perceive the environment.*
2. *Update the belief base.* Belief updates must now take into account the scoping of beliefs. This is achieved by examining a belief update for any URI prefix and using this to determine which module's belief base should be updated (taking into account export lists). Any unqualified belief updates are performed using the agent's main belief base.
3. *Receive communication from other agents.*
4. *Select socially acceptable messages.*
5. *Select an event.* The set of events to choose from is taken as the union of all of the pending event queues for each module and the agent's main event queue.
6. *Retrieve relevant plans.* The set of potentially relevant plans depends on the scope of the triggering event. For events that are local to a module, then only that module's plan library should be considered. For events scoped at the level of the overall agent, the set of potentially relevant plans is the union of the plan libraries of all modules. Determining whether a plan matches a triggering event is done by first expanding any URI references in both into *Jason* annotations, as described previously, and then performing matching as in the current implementation.
7. *Determine applicable plans.* As for relevant plans, determining applicable plans (i.e., those whose belief context is satisfied), must also respect scoping of those beliefs and expanding URI references.
8. *Select one applicable plan.* As described in Sec. 3, one of the motivating justifications for this work, and a key compositionality requirements, is that each module should be able to respond independently to events. It therefore seems sensible to change this step in the cycle so that up to one applicable plan is selected *per module*. However, this approach is more complicated than it immediately appears. The

problem is that belief and goal events resulting from an existing intention cause any responding plans to become part of that same intention structure. Clearly this presents a problem if multiple plans are selected, as only one can become part of the intention (without much more drastic changes to the structure of intentions). We discuss how to tackle this problem below.

9. *Select an intention for execution.*
10. *Execute an action.*

As described, the problem of allowing multiple modules to respond to the same event is more complex than it initially appears, due to interactions with the BDI model of *Jason*. In determining a solution to this problem, it is worth considering the types of events that can occur, and what an appropriate behaviour should be in each case. *Jason* currently distinguishes between so-called 'internal' events, arising from changes to the agent's internal state (beliefs and goals) caused by executing intentions, and 'external' events, caused by the arrival of new percepts and messages from other agents. External events always cause a new intention structure to be created, whereas internal events reuse the intention structure that generated the event. One solution, then, would be to only allow multiple modules to react to external events (creating a new intention for each module), and treat internal events as before, in which case only one module would get to respond. However, this seems overly restrictive, as it is reasonable for multiple modules to respond to *belief* updates caused by an executing intention, whereas a *goal* update should only result in one course of action being taken, to avoid conflicting or incoherent behaviour. Belief change events are largely *incidental* rather than intentional, and therefore should not put such strong constraints on resulting behaviour. For instance, an agent may want to perform various house-keeping tasks in response to a belief change: inferring further conclusions (forward-chaining), informing other agents of the change, and so on, but this is much less likely for goals. It therefore seems more sensible to distinguish events not by an internal/external distinction, but rather by a belief/goal distinction. We therefore propose reverting to the approach taken in the original Procedural Reasoning System (PRS) [9] (Sec. 8.2), in which plans responding to belief update events always create new intention structures, allowing multiple modules to respond to the same event, whereas goal update events follow the current behaviour in *Jason*: only a single plan can respond to the event and this plan becomes part of the same intention that caused the event (if one exists). This addresses the compositionality requirements for beliefs, but still requires some coordination between module authors wishing to respond to the same goal event. Conflicts between multiple plans reacting to the same *goal* are properly the concern of the agent's usual practical reasoning and plan selection functions. In practice, these changes address all of the issues that we encountered in the witness-narrator agent framework.

6 Related Work

Most popular current programming languages support some form of module system allowing for decomposition of large programs into functionally encapsulated components. Approaches range from simple namespace mechanisms, to sophisticated module

systems supporting parameterisation, instantiation, and complex nested module structures. Within the realm of agent-oriented programming languages and frameworks, a number of proposals have been presented in the literature.

A form of modularity based on the notion of 'roles' in multi-agent systems was introduced in [10]. A *role* encapsulates the beliefs, goals, plans and social norms/obligations that are required for an agent to fulfill a particular role in a society. Roles can be 'enacted' at runtime, in which case the beliefs of the role are added to the agent's beliefs and the goals and plans are also adopted. Only a single role can be enacted at any one time, in contrast to most other modularity proposals. This approach would not have sufficed for the witness-narrator agent system, in which agents can be assigned multiple roles simultaneously, either within a single team or in multiple teams.

The notion of modules as 'capabilities' was developed within the context of the JACK^{TM}agent framework [11] and has been extended in the JADEX framework [12]. A *capability* in this proposal is a collection of plans, beliefs, and events together with some scoping rules to determine which of these elements are visible outside of the module, and which are encapsulated. Like the proposal described in the current paper, capabilities represent a middle layer between that of an agent and the level of individual plans and beliefs. Capabilities address the concerns of avoiding name clashes and hiding of implementation details, while also supporting multiple instances of the same module to be created. The later work with JADEX extends the concept to include a more flexible notion of scoping for beliefs and events, as well as allowing capabilities to be parameterised and dynamically instantiated. However, capabilities do not address the problems of plan selection and execution that we have described, i.e., to allow plans from different modules to each have a chance to react to an event.

A proposal for incorporating a similar notion of modules has been described for an extended version of the 2APL programming language [13]. As with capabilities, extended 2APL modules can be instantiated multiple times, or can be declared as *singleton* modules, for which a single instance of the module is shared within an agent. 2APL modules can contain any elements that an agent can contain, including plans, beliefs, goals, and action specifications, and are similar to agents in many respects. Each module can be executed by the module (or agent) that instantiated it, and can receive and generate belief updates, goal revisions, and other events. Another approach to modularity has been described for the related 3APL language [14], in which a module encapsulates just the plans related to a particular goal, or set of goals. A module must be explicitly 'called' with a goal to pursue, after which that module (and only that module) will use its plans to try and achieve the goal. This avoids issues with conflicting plans (assuming each module is internally coherent), but at the cost of requiring more explicit programming from the agent author—deciding not only what goals to adopt, but which modules should be used to achieve them. The notion of modules as 'policy-based intentions' in GOAL [15] relaxes this restriction by allowing the module to specify a context condition on beliefs and goals that determines when it is activated. However, to the best of our knowledge, the implementations of these languages do not address the issues of belief compositionality we have described. For *Jason*, a simple module system has been developed as part of the work on integrating ontological reasoning in the JASDL system [16]. However, this work concentrates on allowing per-module customisation

of the various plan and intention selection functions in *Jason*, and does not address the modularity concerns described in the current paper.

The changes described in Sec. 5.2 to the processing of events are based on the original scheme used in the Procedural Reasoning System (PRS) [9] (Sec. 8.2), in which belief change events cause new intentions to be created, whereas goal changes reuse the current intention structure. Given AgentSpeak's (and therefore *Jason*'s) historical basis in the PRS architecture, it seems natural to revert to this scheme.

While many agent programming languages use Prolog as a belief base language, e.g., *Jason*, 2APL, GOAL[17], there has been some work on alternative belief representation languages. The use of alternative knowledge representation languages for the belief base of a *Jason* agent has been considered in the context of adding support for ontological reasoning and OWL [18, 16]. However, this work has concentrated on extending the Prolog facilities of *Jason* with support for ontological reasoning, rather than replacing the existing belief base language. In [19] an approach to abstracting an agent language from any particular knowledge representation format is presented. While the authors note that Datalog and SQL meet their requirements for a *Knowledge Representation Technology*, the focus of the paper is on translations between knowledge representation technologies rather than the practicalities of any specific technology. Most agent programming frameworks (including the implementation of *Jason*) allow customising or replacing the default belief base to a certain degree. While such customisation can facilitate the integration of 'legacy belief bases', there remains a need for a practical knowledge representation formalism, even if only as a default, and we feel that that Datalog is a more appropriate choice than Prolog for a default belief-base language.

7 Summary

In this paper we identified a number of problems with the *Jason* agent-oriented programming language motivated by our experience of building a moderately large and complex piece of software using *Jason*. While *Jason* provides a clean and elegant framework for building sophisticated multi-agent systems, it provides less support for developing complex agents with diverse, interacting capabilties. We identified two key problems: a lack of support for modular software development, and an order-dependence in the semantics of the belief representation language which makes it hard to compose modules and to author plans within a module. To address these problems we proposed revisions to the language to simplify the construction of agents from functionally encapsulated components, and changes to the belief base language and plan execution to support more robust software engineering.

In future work, we plan to investigate further revisions to *Jason* suggested by our experiences developing the witness narrator agents system, including changes to the triggering conditions of plans and the semantic characterisation of percepts.

References

1. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons Ltd (2007)

2. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16** (2008) 214–248
3. Madden, N., Logan, B.: Collaborative narrative generation in persistent virtual environments. In: Proceedings of the AAAI Fall Symposium on Intelligent Narrative Technologies, Arlington, Virginia, USA (November 2007)
4. 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
5. Gregorio, J., de hOra, B.: The Atom Publishing Protocol. Technical Report RFC 5023, Internet Engineering Task Force (October 2007)
6. 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`.
7. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering **1**(1) (1989) 146–166
8. Baader, F., Calvanese, D., McGuiness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook. Cambridge University Press (2003)
9. Myers, K.L.: Procedural reasoning system user's guide. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA (1997)
10. Dastani, M., van Riemsdijk, M.B., Hulstijn, J., Dignum, F., Meyer, J.J.C.: Enacting and deacting roles in agent programming. In: Proceedings of Agent Oriented Software Engineering (AOSE 2004). Volume 3382 of LNCS., Springer (2004) 189–204
11. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI agents in functional clusters. In Jennings, N.R., Lespérance, Y., eds.: Intelligent Agents VI, LNAI 1757, Springer (2000) 277–289
12. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible BDI agent modularization. In: Proceedings of ProMAS 2005, LNAI 3862, Springer (2005) 139–155
13. Dastani, M., Mol, C.P., Steunebrink, B.R.: Modularity in agent programming languages: An illustration in extended 2APL. In: Proceedings of the 11th Pacific Rim International Conference on Multi-Agent Systems (PRIMA 2008), LNCS 5357, Springer (2008) 139–152
14. van Riemsdijk, M.B., Dastani, M., Meyer, J.J.C., de Boer, F.S.: Goal-oriented modularity in agent programming. In Nakashima, H., Wellman, M.P., Weiss, G., Stone, P., eds.: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006, ACM (2006) 1271–1278
15. Hindriks, K.V.: Modules as policy-based intentions: Modular agent programming in GOAL. In: Proceedings of ProMAS 2007. Volume 4908 of LNCS., Springer (2007) 156–171
16. Klapiscak, T., Bordini, R.H.: JASDL: a practical programming approach combining agent and semantic web technologies. In: Proceedings of DALT 2008. (2008) 91–110
17. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.C.: Agent programming with declarative goals. In: Intelligent Agents VII Agent Theories Architectures and Languages. Lecture Notes in Computer Science. Springer, Berlin (2001) 248–257
18. Moreira, A.F., Vieira, R., Bordini, R.H., Hübner, J.: Agent-oriented programming with underlying ontological reasoning. In: Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies (DALT), Utrecht, Netherlands (July 2005) 132–147
19. Dastani, M., Hindriks, K.V., Novák, P., Tinnemeier, N.A.M.: Combining multiple knowledge representation technologies into agent programming languages. In: Proceedings of DALT 2008. (2008) 60–74