Jason+

Extension of the Jason agent programming language

G53IDA – Dissertation

Submitted 06 May 2010, in partial fulfilment of the conditions of the award of the degree:

B.Sc. (Hons) Computer Science with Artificial Intelligence

by

Daniel Nicholas Kiss (dnk07u)

Supervised by

Dr. Brian Logan (bsl)

School of Computer Science and Information Technology

University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature: __________________________

Date: 06 May 2010
ABSTRACT

Jason is an open-source platform written in Java which provides support for development of multi-agent systems. It is an implementation and extension of the BDI (Beliefs – Desires – Intentions) architecture based AgentSpeak(L) agent-oriented abstract programming language, and it can be used as a plugin for jEdit or Eclipse; further it provides high level of support and customization for most parts of the development of a complex multi-agent system.

The aim of the project is to find ways and possibilities how the existing Jason agent-oriented programming language could be improved, and to implement these upgrades, in the eventual motivation of providing better support for Agent Development. Thus it means investigating and researching potentials in which way this agent-oriented programming language could be re-engineered or extended.

Hence, this dissertation describes the research has been made so far; including the basics of BDI, AgentSpeak(L) and Jason; and the found improvement possibilities which eventually would support the development of complex multi-agent systems.

Source: wordpress.com
# TABLE OF CONTENTS

Abstract .......................................................................................................................... 2
Table of Contents .............................................................................................................. 3
Introduction and Motivation ............................................................................................. 6
  Introduction .................................................................................................................... 6
  Motivation ...................................................................................................................... 6
  BDI ................................................................................................................................. 7
  Beliefs ............................................................................................................................. 7
  Desires ........................................................................................................................... 7
  Intentions ....................................................................................................................... 8
  Events ............................................................................................................................ 8
  An Example using the BDI model .................................................................................. 8
AgentSpeak(L) ................................................................................................................... 9
Jason ................................................................................................................................. 9
Jason ................................................................................................................................. 12
  MAS Project ................................................................................................................. 12
  Syntax ............................................................................................................................ 12
  Reasoning Cycle .......................................................................................................... 17
Related work .................................................................................................................... 18
  CASO ............................................................................................................................. 18
  AgentSpeak(PL) – Peleus ............................................................................................ 18
  Normative AgentSpeak – Iovis ...................................................................................... 19
  Improving Jason ......................................................................................................... 19
    Jason plugin for Eclipse .............................................................................................. 19
    Jason port to C/C++ ................................................................................................. 19
Project Aims, Objectives and Limitations ....................................................................... 20
  Aims .............................................................................................................................. 20
  Objectives .................................................................................................................... 20
  Limitations ................................................................................................................... 21
    Limitations of the Development Process .................................................................. 21
Description of the work ................................................................................................... 22
  Research ....................................................................................................................... 22
  Implementation Decisions ......................................................................................... 22
  Development Setup .................................................................................................... 22
Specification ........................................................................................................... 23
Module System ......................................................................................................... 23
Truly Atomic Intentions.............................................................................................. 25
Dynamic Module Import ............................................................................................ 25
Design.......................................................................................................................... 27
Module System ......................................................................................................... 27
  System Alternatives: Flat System / Hierarchical System .................................... 27
  Description of the Chosen Alternative .................................................................. 28
  Visibility Control Alternatives: Only Qualified Access / Easy Access ............... 30
Truly Atomic Intentions.............................................................................................. 30
  Alternative of Restoring the Original State .......................................................... 31
  Description of the Chosen Alternative .................................................................. 31
Dynamic Module Import ............................................................................................ 33
  Alternatives of Specifying the Module to be Imported ...................................... 33
  Description of the Chosen Alternative .................................................................. 33
  Extension of Truly Atomic Intentions .................................................................. 34
Implementation .......................................................................................................... 35
Module System ......................................................................................................... 35
  Syntax of the Language ......................................................................................... 35
  Actions at Parse-time ............................................................................................ 36
  Actions at Import-time .......................................................................................... 39
Truly Atomic Intentions.............................................................................................. 41
  Transition System .................................................................................................. 42
  Delta Belief-Base .................................................................................................... 43
Dynamic Module Import ............................................................................................ 45
  Internal Action – ‘import’ ...................................................................................... 46
  Dynamic Import ..................................................................................................... 46
  Delta Plan-Library .................................................................................................. 47
  Delta Goal Addition Events .................................................................................... 48
Other Improvements Made ......................................................................................... 48
  Better Integration of the Module System .............................................................. 49
  Support of URLs in the Include directive ............................................................. 50
  Proper Handling of String Literals ......................................................................... 50
  Syntax Colouring of jEdit ....................................................................................... 50
  More Detailed Error Messages ............................................................................. 50
  Better Error Identification ...................................................................................... 50
  New Branch – Jason+ ............................................................................................. 51
INTRODUCTION AND MOTIVATION

This part of the dissertation will provide a general introduction to the project and describes the motivations and purpose of the project. It also gives the reader a brief introduction to the BDI architecture, to the AgentSpeak(L) agent-oriented abstract programming language which is based on this architecture; and finally to Jason, which is its extension and its Java implementation.

Introduction

A significant amount of development and research has been done in the area of multi-agent systems (MAS), which aims to provide an abstract interface for the developers, and by simplify the tasks of a programmer by providing a more supportive platform. Multi-agent platforms, based on these abstract programming languages, are becoming more and more popular, because they ease development for a wide variety of problems involving multi-agent or ubiquitous systems.

One of the most mature multi-agent platforms is Jason, which is based on the BDI architecture based AgentSpeak(L). Jason has many new features and extensions compared to the original AgentSpeak(L), because of the need of the good support for the development of these systems. Jason is already a very useful tool which provides a high level of support to the developer, however it is lacking in some areas particularly involving organization and structurization issues, which makes the work of developer harder than it should be. The main reason for these issues is that Jason has implemented many new features linearly by extending AgentSpeak(L) again and again, without careful consideration of organization issues.

The aim of this project is to identify the some of the most important of these issues, create a plan how to fix them, and finally extend Jason with these identified upgrades to provide a better development support.

Motivation

The main motivation for the project is to provide an even better support for the development of multi-agent systems by extending an already very useful tool with the same purpose. The extension of this tool (Jason) would significantly improve it in some areas.

It is important because, as the multi-agent systems become more and more popular, the languages used to develop these become richer, which makes it harder for the programmer, who develops these systems, to create well-organized, easily review-able and reusable code. As it has been already described, these multi-agents systems are useful, and it is also proven by their increasing popularity; hence there is a real need for their very good support. And one of the ways that it can be done, by making an already very supportive development platform for these systems even more supportive by identifying where it is lacking, and then extend this tool with fixes for the issues identified.
BDI

This section will introduce the reader to the concepts of the BDI architecture based multi-agent systems.

BDI (Beliefs-Desires-Intentions) is a software model developed for programming rational / intelligent agents. As its name suggests, the BDI model uses the concept of an implementation organized around beliefs, desires and intentions. It has means to create a balanced separation between the selection of a plan and the execution of the selected plan. However the creation of the plans remains the task of the agent programmer, and the agent only decides which one of these predefined plans to choose. Hence this architecture provides the means to develop intelligent agents; however it does not ensures, that the agents are actually going to act rationally; this is still remains the task of the programmer. However it is still one of the best known approaches to the development of rational / intelligent agents.

Beliefs

Beliefs are the knowledge of an agent about the environment in which it acts. This is all the information which the agent has in its current state; hence it means the beliefs about the world including other agents and itself as well. Thus this is the main ‘data’ part of a running agent-program. For example: the glass is on the table; the glass is not on the tray.

Belief-Base

The Belief-Base (BB) of an agent is the actual place or data storage where the current beliefs of the agent are being kept. Sometimes it also referred as the Belief Set of the agent. Beliefs are usually stored in a database, as there may be a large amount of information to be stored and maintained. An adoption of the BDI architecture relieves the programmer of the responsibility of handling or worrying about this data.

Desires

The desires of an agent are the set of objectives which the agent would like to accomplish, thus it is the part of the agent which motivates its actions. It defines the possible purposes of the agent by describing the available aims or sets of aims it should accomplish while it is active. For example: put the glass to the tray.

Goals

A goal is one of the desires (or a set of desires) which the agent is actively pursuing. It is the set of desires that the agent is currently using to choose intentions from. The set of goals is more restrictive than the set of desires by disallowing to have two desires in the set of goals which are in contradiction with each other; such contradiction for example would be: put the glass to the table; put the glass to the tray (assuming that the tray is not on the table).
**Intentions**

The intentions are the goals (hence desires) which the agent has selected to accomplish. Thus, these are the desires to which the agent has dedicated at its current state, and it acts (or tries to act) on its environment to achieve these desires. Intentions are strongly connected to plans, because for an agent to be able to attain its intentions it is highly likely, that the agent is required to act on its environment (including its belief-base and the beliefs of the agent itself).

**Plans**

The plans of an agent are basically the sequences of actions that the agent can perform to accomplish its intentions. Plans are composed of elementary actions performed on the environment, and other plans; further some of their details could be extended during the progress of an agent as well.

**Events**

Events describe the reactions of an agent to different kinds of changes. The source of these changes could be external information – for example sensors of an agent, or communication received from another agent – or could be some internal activity. Hence these are triggers for different kind of events, which describes the reactions of the agent to them. These reactions could be updating the belief-base, executing plans or changing goals (or a set of these).

**An Example using the BDI model**

In the followings an example is going to be described to demonstrate the use of the terms explained above.

In this example system, there are three different agents inside a single-doored room: Agent Claustrophobe who would like to maintain that the door is closed, Agent Paranoid who would like to maintain that the door is opened, and Agent Porter who is actually capable of opening / closing the door, and who could be asked for that.

Each agent has their own preceptor, which provides them the ability to know the state of the door. And hence they have some knowledge about their environment, thus they have beliefs: they could believe that (1) they have no information about the state of the door (e.g.: at the start of the system), (2) they believe that the door is closed, or (3) they believe that the door is opened. They should not believe simultaneously any two of these there, e.g.: the door is opened and closed at the same time.
The *desires* – and because it is a simple system the *goals* – of Agent Claustrophobe is to maintain that the door is closed; the desires of Agent Paranoid is to maintain that the door is opened; and the desires of Agent Portal is both to get the door opened and to get the door closed. Its *goals* however should be maintained runtime depending on the *events* of the environment, either to get the door opened or closed, but not both.

Hence the actions – affecting the environment of the agents – generate the *events*. These events could trigger different kinds of actions: (1) if Agent Claustrophobe detects that the door has just got closed, its *intention* will be to get it opened, hence it executes a *plan*, which asks Agent Porter to open the door. (2) Similarly, if Agent Paranoid detects that the door has just got opened, its *intention* will be changed and the appropriate *plan* – asking Agent Porter to close it – will be selected and executed. (3) And finally, if Agent Porter detects an *event* of receiving a message which asks to open / close the door, its *goals* will be changed; and an *intention* will be selected from these *goals* (the only one available in this simple system), and executes the appropriate *plan*, which is a sequence of actions for opening / closing the door.

**AgentSpeak(L)**

AgentSpeak(L) is based on the BDI architecture explained above. It is an agent-oriented programming language proposal developed in 1996 by Anand S. Rao.

AgentSpeak(L) is an abstract agent-oriented programming language which shows the relation between the BDI architecture and its practical implementation. It was designed to simplify the language used by PRS (Procedural Reasoning System). It is an abstract language with operational and proof-theoretic semantics, which was explicitly designed to formalize BDI agents, and it can be used to prove the properties satisfied by these BDI agents.

**Jason**

Jason is an implementation of the AgentSpeak(L) proposal, which also adds many additional features to the original language. It is a multi-agent system development platform, which provides high level of support to the agent programmer. It is written in Java, and so it platform independent. It is open source and distributed under the GNU LGPL.

In addition to the original AgentSpeak(L) language, Jason supports strong negation, hence it is capable of handling the representation of incomplete information; it supports inter-agent communication, and annotations as well. It is highly customizable, for example additional Environments could be programmed in Java for Jason or the perception / belief-revision / inter-agent communication / action methods could be revised; moreover, user-defined internal actions could be created (again, in Java) too. Further it provides ways to run a multi-agent system, which is however distributed over a network making this platform very broadly usable.
Jason is more than a simple language, it is a platform, a tool for agent programmer to create multi-agent systems, and as it has been explained it provides high level of support. For example Jason is available as integration (a plugin) to an IDE, making it very user-friendly to use. Jason at the moment has plugin for jEdit and Eclipse as well.

The picture below is a screenshot of jEdit and the Jason plugin integrated in it. On the left-hand side of the window, there is the list of plans defined for the agent. On the top there are the currently opened files, including the MAS file which describes all the agents in the multi-agent system. At the bottom, there is an interface of the Jason plugin including the control buttons, a console and the list of agents in the current project. Hence, as it can clearly be seen, Jason provides many possibilities for the support of the agent programmer.

The image on the top in the next page is a screen shot of the Jason plugin integrated in to the widely used Java IDE, Eclipse; which shows similar features as the plugin for jEdit.

Further, Jason also has a Mind Inspector; which is a graphical tool mainly used for debugging and verification purposes. It is rather interesting to be able to see the actual states of the agents at any cycle of the execution of the multi-agent system. It allows the user to observe the current state of the agents and the changes made in them. This state includes many aspects of the agent, including the beliefs, events, intentions and actions of the agent. Moreover, this very useful mind inspector also supports distributed agents in a system which is distributed over the net.
Jason plugin for Eclipse:\textsuperscript{1,13};

Jason’s Mind Inspector used for debugging and verification:\textsuperscript{5};
Because the project is about the extension of Jason, it is important to understand the basics of Jason including its syntax and current functionalities. Hence, this section will briefly introduce the reader to these. If further information is required, then please consult Jason’s documentation.

**MAS Project**

The main purpose of the MAS project source file (.mas2j file) is to describe the agents of the project. It includes listing their names, and optionally stating their source file, verbosity, cardinality and other properties.

It can also describe the infrastructure to be used, the properties of the environment the class-path and source-path in which the different kinds of files are looked; but this dissertation will not go into the details of these, but further details can be found in Jason’s documentation.

Therefore the whole MAS Project consists of a single .mas2j file and an arbitrary number of .asl files for the agents.

**Syntax**

In the followings the syntax of a single Jason agent’s source file (.asl file) is going to be described. The source file is divided into 3 sections: initial beliefs, initial goals and plans; and this way each element must be placed into the section of its own category. Hence, for example no initial goal can be specified after the specification of a plan, because it indicates that the section of plans is started.

**Predicates**

Because almost every element of an agent is built from predicates, first the syntax of a predicate is described. A simplified syntax looks like that:

```
predicate ::= <functor> [ "(" <terms> ")" ] [ "[" <terms> "]" ]
functor ::= <atom>
atom ::= ( <lower_char> | "." <char> ) ( <char> | "." <char> )*
terms ::= <term> ( "," <term> )*
term ::= <predicate> | <log_expr> | <as_string> | <number> | ...
```

The first part is the functor of a predicate is its alphanumeric (including underscore ‘_’) name which may contain multiple parts, where the parts are divided by a dot ‘.’. Also, the predicate can start with a dot. For example: ‘.send’ and ‘my.function’ are allowed, but ‘my..function’ is not.

The optional second part (in parentheses) contains the terms / parameters of the predicate. The arity of the predicate is the number of parameters it have. It can include complex logical expressions as well, for example results operators, but for simplicity these are omitted here.
The optional third part (in brackets) contains the annotations of the predicate. They have the same syntax as the parameters, but they have different function. Predicates have a special predefined annotation ‘source’ which specifies the origin of the predicate – for example: self, percept or a name of an agent.

The indicator of a predicate describes the functor and arity of the predicate; and it is usually marked like the followings: `<functor> "/" <arity>`

An example: ‘agentCalled(bob) [source(self)]’ of which single parameter is ‘bob’, hence its arity is 1, hence its predicate indicator is ‘agentCalled/1’. Similarly, the single annotation of the agent is ‘source’ which also has single parameter ‘self’.

**Variables**

```
variable ::= ( <upper_char> | "_" ) ( <char> )*  
```

Variables start with an uppercase letter or an underscore. They must not have parameters or annotations. Variables starting with an underscore are unnamed and each of them denotes a unique variable. Variables can be in the place of any other predicate or literal (e.g.: strings, numbers, lists).

**Initial Beliefs**

Beliefs are first order literals, which describe the knowledge of the agent about its environment about itself as well (see BDI). As their name suggests, the initial beliefs describe that the agent starts with. Because beliefs have the same syntax as predicates the syntax of initial belief is the following:

```
initialBelief ::= <predicate> "."  
```

The ending dot ‘.’ separates the elements of the agent’s source file from each other.

An example: ‘agentCalled(bob) [source(self)].’

**Initial Achievement Goals**

The initial goals of an agent are those achievement goals which the agent should start to persecute when it is first activated. The notion of achievement goals start with an exclamation mark ‘!’ as the followings show:

```
initialGoals ::= "!" <predicate> "."  
```

And again, the ending dot has a separator function.

An example: ‘!callAgent(bob)’.
Plans

Plans could think of as the core functional part of the agent, as these describe the actual actions performed by it, hence the plans describe the agent’s behaviour. The syntax of a plan follows:

\[
\text{plan} \quad ::= \quad [ \text{@} \; \langle \text{predicate} \rangle \; ] \; \langle \text{trigger} \rangle \; [ \; \text{";"} \; \langle \text{context} \rangle \; ] \\
[\; \text{"<-"} \; \langle \text{plan\_body} \rangle \; ] \; "." \\
\text{trigger} \quad ::= \quad ( \; \text{"+"} \; | \; \text{"-"} \; ) \; [ \; \text{"!"} \; | \; \text{"?"} \; ] \; \langle \text{predicate} \rangle \\
\text{plan\_body} \quad ::= \quad \langle \text{plan\_action} \rangle \; (\; \text{";"} \; \langle \text{plan\_action} \rangle \; )^* \\
\]

The first optional part (with the at sign ‘@’) denotes the unique label of the plan. If it is omitted, a unique one is automatically generated. The label can be used for identification purposes and when the agent’s plans are dynamically modified. Further, the annotations of the plan’s label are the annotations applied the plan itself.

The trigger of the plan describes the event when should the plan be activated. It can be a belief addition / removal event, a new goal achievement / test event, or a handling of the failure of the latter ones. The presence of an exclamation mark (‘!’) denotes an achievement goal, the presence of a question mark (‘?’) denotes a test goal, and the absence of both denotes a belief. A plus and a goal (‘+!’ or ‘+?’) denotes the description of the normal actions of that plan; a minus and a goal (‘−!’ or ‘−?’) denotes the description of a plan corresponding to the appropriate goal which has failed; and a plus (‘+’) or minus (‘−’) and a belief denotes a belief addition / removal event.

A failure event can be caused by many things – for example if a test goal is not specified explicitly, then by default it fails if the agent does not believe in the predicate under test. Failure events can also be generated explicitly by the usage of custom actions.

The context of a plan is an optional additional condition which is required to be met before the plan can be selected for the event. It can test a belief for example or any other logical expression.

The body of the plan contains the arbitrary number of actions which should be performed after each other in sequence. There are several different kinds of action that can be performed as part of a plan, the following lists and described these.

Belief Addition / Removal

\[
\text{beliefModification} \quad ::= \quad ( \; \text{"+"} \; | \; \text{"-"} \; | \; \text{"+-"} \; ) \; \langle \text{predicate} \rangle \\
\]

Plus (‘+’) denotes the addition of a belief, minus (‘−’) denotes the removal of a belief and minus-plus (‘+-’) denotes the removal of the first belief which has the same predicate indicator, and the addition of the new one, therefore in a way it means a belief replacement.

If an existing belief is tried to be added, then only the new beliefs annotations are imported to the old one, hence in a way it is only a belief modification; for example if:

‘\text{+called(bob) [tuesday]’ is executed on the belief-base ‘\text{called(alice) [monday], called(bob) [monday]’}, it would result in ‘\text{called(alice) [monday], called(bob) [monday, tuesday]’.
Similarly if a belief which has multiple is tried to be removed, then only its appropriate annotations are dropped; for example if:

‘-called(bob)[tuesday, source(charlie)]’ is executed on the belief-base ‘called(bob)[monday, tuesday, source(self), source(charlie)]’ it would result in ‘called(bob)[monday, source(self)]’; hence in a way it is only a belief-modification too.

However for example if:

‘+called(bob)[tuesday]’ is executed on the belief-base ‘called(alice)[monday], called(bob)[monday]’ it would result in ‘called(alice)[monday], called(bob)[tuesday]’.

**Goals**

\[
\text{goal ::= } ( "!" \mid "!!" \mid "?" ) <\text{predicate}>
\]

There are two different kinds of goals: achievement (‘!’) and test goals (‘?’); and achievement goals can be started as new focus (‘!!’) as well. As it is described and achievement goal means that the agent should start pursuing those, a test goal simply performs a test and generates a failure if something goes wrong.

If a normal achievement goal (‘!’) is used, it means that this goal is a subgoal of the one which has invoked it; therefore no subsequent actions of the plan’s body can be executed until this sub-goal has been achieved, and the execution of the main plan can only continue after the execution of the sub-plan has finished.

On the other hand, if an achievement goal is used as a new focus (‘!!’), it means that the pursing of this new goal is completely independent from the one which has invoked it. It can be thought of as if the other goal would be executed in a different execution thread (even though Jason is single threaded). Therefore the execution of the new plan can either start immediately (as usually) or it can be delayed until the execution of the original plan has finished; hence new subsequent actions of those can be performed.

**Action**

\[
\text{action ::= } <\text{atom}>
\]

While the previous two has only controlled the reasoning cycle of the agent, these actions include those performed on the environment as well. The name of the action is similar to the functor of a predicate: it is an alphanumeric string (including underscore ‘_’), where possible subparts are divided by a dot (‘.’).

**Environment’s Actions**

Those actions which contain no dots (‘.’) in their name are considered as the actions of the environment. The specific meanings of these are implemented in the Environment’s executeAction method which receives two parameters: the name of the agent which performs the action, and the name of action itself.
Standard Internal Actions

Internal Actions are those which contain at least one dot (‘.’) in their name. However, a standard internal action starts with a dot (‘.’). The implementations of these are part of Jason’s release and they are found in the jason.stdlib package.

The functionality of these is wide-spread (over 65 actions), and they make Jason a rich and powerful tool. For example these can be used to perform complex queries / changes on beliefs. Moreover, by using these standard internal actions it is possible to change the behaviour of the agent at run-time by modifying and communicating plans; and this way the capabilities of an agent can be extended.\(^{21}\)

User-Defined / Custom Internal Actions

Internal actions which name does not start with a dot (‘.’) are interpreted as custom internal actions. The name of these must match the fully-qualified names of the Java class containing their implementation. For example the action ‘uk.ac.nottingham.RunAway’ would cause the agent to invoke the execute(TransitionSystem, Unifier, Term[]) process of the RunAway class implementing the InternalAction interface.

Hence these are rather similar to the standard internal actions, the only difference is that their implementation is not part of Jason’s release, but they are implemented by the agent-developer.

Example

Some examples of plans follow:

\[ +!\text{registerCall}(A) \leftarrow +\text{agentCalled}(A)[\text{cost}(5)]. \]
\[ @\text{memoCall} \]
\[ +\text{needToCall}(\text{bob}) \leftarrow \text{leaveMemo}(\text{alice, ”Please call Bob for me!”}). \]
\[ +!\text{putBookToShelf}(B, \text{Agent}) : \text{not}(\text{bookOnShelf}(B)) \leftarrow +\text{bookOnShelf}(B); \]
\[ .\text{send}(\text{Agent, tell, bookOnShelf}(B)). \]

Rules

Rules help to demonstrate logical consequences; hence – apart from the plans – these are the non-lexical knowledge of the agent. Their syntax follows:

\[ \text{rule ::= <predicate> ":-" <log_expr> "."} \]

Rules can be placed in both the section of beliefs and the section of plans.

An example: ‘car(A) :- numOfWheels(A, 4) & vehicle(A).’, and a possible interpretation could be that if it is believed that the predicate hold by the variable A is a car, then it is also believed that it is a vehicle and it has four wheels.
Directives

Directives are special instructions to the parser and they are not part of the standard description of an agent. In general a directive can have any arbitrary action, for example it can add initial beliefs, goals or plans to the agent based on some predefined rules or its parameters. Their syntax follows:

\[
\text{directive ::= "\{ [ "begin" ] <predicate> "}"
\]

Directives can be placed in any section of the agent. The directive ‘begin’ must end with a directive ‘end’; and the text between two has the same structure that an individual agent would have, hence it has the 3 main sections of beliefs, goals and plans.

The following is an example of a directive.

Include

As its name suggests, the function of the ‘include’ directive is to include the context of another source file to this one. It has one parameter, which specifies the path of the file to be included. It is not a pure string substitution as the included file can have its own 3 sections of beliefs, goals and plans. Once these elements are parsed, they are included as if they would be defined in the agent’s own source file.

For example: ‘{include("dir_commands.asl")}’, and it will locate and load the file ‘dir_commands.asl’. The file is search along the specified source path of the project.

Reasoning Cycle

The execution of Jason is divided into cyclic steps; and at each cycle only a single action is performed. At the begging of each cycle an event is selected by the event selection function. If the selected event has no intention assigned yet, then the relevant plans (those which have a trigger matching the event) are selected. Afterwards, the applicable plans are chosen by filtering out the plans of which context is valuated to false. Then a plan selection function decides which plan to take from the remaining plans, and a new intention is created which will start executing the actions described by the plan.

However if the event already had an intention assigned (a plan has been selected for that event in a previous cycle, but it was not completed yet), then that intention is continued. The intention executes step by step the actions described by the plan, performing a single step at each cycle. Once the plan is finished (either because it is successfully completed or a failure has happened), the event is removed from the agent’s list of events.

If a new event rises (for example a belief was added), then it is included in the agent’s current list of events, and at the next step the event selection function will decide in the same way, but now considering the new event as well. Therefore, even though the actions are performed step by step in a single thread, parallel execution of intentions is possible, as new intentions can be created and selected without the need to finish a previous intention. While one intention is executed, the others remain pending.
RELATED WORK

In this section some of the similar works in the field are described and compared to the project of this dissertation. The following projects are mainly other extensions of Jason or implementations of AgentSpeak(L).

**CASO**

CASO is an agent-oriented programming language extending AgentSpeak(L), and its implementation is a modification of Jason. CASO allows the user to provide real-time a specification of constraints and objectives on possible goal states of reactive BDI agents. CASO also uses a constraint solver to apply its techniques. Hence, CASO adds a new feature to Jason; however it still does not concentrate on the existing issues of Jason; moreover the feature added could lead to even more organization issues; even though that the added new feature could indeed be proven to be very useful.

**AgentSpeak(PL) – Peleus**

As well as CASO, the implementation of Peleus is based on Jason too. AgentSpeak(PL) is a planning-capable extension of AgentSpeak(L). Its agent architecture follows the tradition of PRS (Procedural Reasoning System) agents, they are able to reason about declarative goals. Therefore it is a completely different approach, making the language even more complex – AgentSpeak(L) was explicitly designed to simplify PRS. The approach concentrates on declarative agents, and it still doesn’t address issues with the current BDI based multi-agent systems, even if it does provide new automated features easing some work of the developer.

See the images on the next page demonstrating some of the differences between the two.
Normative AgentSpeak – Iovis

Iovis is an AgentSpeak(L)-based implementation of an interpreter which provides mechanisms to adapt an agent at runtime to norms constraining the behaviour of the agent. Hence it is an extension to BDI agent languages, in which the agent can make modifications run-time on their own behaviour by responding to norms. It is a rather interesting project with similar aims to this one (e.g.: Future Work > Automated Belief Revision), however it still concentrates on a newly implemented feature, and not on the necessary improvement of the existing ones.

Improving Jason

The fact, that there is indeed a need to improve / extend Jason – because it lacks some areas, especially involving complex systems – is proven by that, there is academic research has been made so far which specifically focuses on issues of the language. Such work for example would be the Modularity and compositionality in Jason.

The followings also help to demonstrate the need to extend Jason, but while this project focuses on improving Jason, these projects purely extending it ignoring current issues with it.

Jason plugin for Eclipse

This project mainly involves the development of the plugin which can be used with the popular Java specialized IDE, Eclipse. The project is indeed very useful, and it provides a great support for development of multi-agent systems; even that much that the documentation of Jason has many references to it (see Introduction > Jason). However this plugin uses the original structure and language of Jason, hence it does not extend / modify the language of Jason, and does not solve some of the issues identified (see Future Work), like the project of this dissertation aims to.

Jason port to C/C++

The project – jason-c – aims to port the whole Jason tool to C/C++, which would give a wider possibilities of development, and it might even improve its performance as well; however again it does not address issues with the actual language of Jason.
PROJECT AIMS, OBJECTIVES AND LIMITATIONS

This section of the dissertation describes, what is it that the project tries to achieve, what are the eventual abstract goals that should be attained by the end of the project. Then it describes the specific objectives that should also be implemented by that time, and shows the known limitations of the project including the area of development and the expected limitations of the result.

Aims

The main aim of the project is to extend the agent-oriented programming language Jason (see Introduction > Jason). Hence, the project should create an even more supportive tool for agent development by extending / improving one of the best multi-agent system development platforms at the time of writing this dissertation.

The created extension of this software should still have most of the positive characteristics of the original software; however it should identify and fix some of the found issues of the language or the platform. Thus, for example the completed application should still be platform independent, and it should still support systems / agents broadcasted over a network, to eventually provide a state of the art piece of software.

Further, the created system should be supported by a documentation explaining the differences between this new version of Jason (Jason+) and the original version. For details, see the Appendix.

Objectives

As part of this project, there are 3 large distinguishable extensions of Jason planned to be done. These extensions should fulfil the aims of the project.

First of all, a module system should be created, as such a system would greatly support the agent-developer by providing features for organizing code, and supporting code reusability and team work, which is unavoidable during the development of complex systems; and as it addresses many issues of the development of these.

Secondly, a support of true atomic units of actions should be provided (like transactions in databases), as the application of these provide the agent-developer the possibility to create an agent in a way, which maintains the consistency of the belief-base of created agent by appropriately handling failures.

And finally, as dynamic behaviour modification of agents is possible in Jason, dynamic module imports should be supported too, as it would provide a way of organized behaviour modification and communication between agents. Further, these should be supported even within the atomic units of actions described above to be able to appropriately handle failures, and hence to be able maintain consistency.

If time allows, further improvements should be done, or other extensions should be created.
Limitations

However, the created piece of software still remains a tool only; and hence the real decisions and work remains the agent-developer’s. Jason+ should provide the possibility of efficient code reusability and teamwork; however it will not ensure it. Similarly, it provides the opportunity to create a code which maintains the consistency of the agent’s belief-base; however this consistency is not ensured, it should be verified by the agent developer as part of the development process. Therefore Jason+ supports to develop, but will not develop multi-agent systems.

Limitations of the Development Process

The main limitations of the development process are the time and resource constraints.

Because the field itself is relatively new and because I was mostly unfamiliar with the field, there was a need to spend a significant amount of time with familiarisation of the field, and an even more significant amount of time with research.

In this project, research is not only important, because of familiarisation reason; but because of the nature of the project. It is imperative to make large amount of research, because it is part of the project, to investigate possibilities, in which way Jason can be improved. The understanding of the current system is very important to be able to identify and then eventually fix the issues with the currently existing system. Therefore a large amount of time – which is already limited – has to be spent on research involving exploration of the current system and its design, and further research to find extension possibilities.
DESCRIPTION OF THE WORK

As it was expected – because of the nature of the project – the research took a significant amount of time; however this research may took even more time than it was originally thought. Hence, this section will describe briefly some of the research made, and some of the preliminary setup of the implementation process.

Research

The research has been concentrated on the understanding and familiarization of Jason, the identification of the issues in Jason, and on the related work, which might prove to be useful to improve Jason.

The results of the understanding / familiarization of Jason can be found in the Introduction section on the first couple of pages. The results of the research about similar works in the field mainly involving Jason can be found in the Related Work section on the previous pages, and the identified issues about Jason can be found in the Future Work section on the following pages.

Implementation Decisions

Because it is an extension of an already existing piece of software, most of the implementation decisions were given; because it was inherited from the application to be extended.

Hence the programming language is used for development is Java, the software is going to be platform-independent, and it will be available as an IDE in a form of plugin for jEdit or Eclipse or it could even be used from the command line too. It should also be able to communicate over a network using Saci or JADE.

The IDE used for development is Eclipse, because it is a very useful tool specialised for Java development, and because Jason is available as a plugin for Eclipse, hence the changes made on the project could be tested instantly in the same IDE.

For further testing jEdit is going to be used, to verify the same results gained during the tests which have been done in Eclipse. Further requirements should match the specification of Jason.

Development Setup

An Eclipse project for the development process has been set up using the available source code of Jason. A version control system has been set up as well, because of its advantages; mainly to use as back-up, and to be able to track the progress of the work. The version control system chosen is SVN and the main project is hosted on Google Code; however weekly backups are made using the server(s) of the School of Computer Science. Further, an automated building process has been set up as well; so for example, whenever a change has been made and compiled, it can be instantly tested in jEdit without the need for manual operations which would be an unnecessary waste of time.
This section will list and describe the actual development tasks selected to implement during the project. For each task, it explains what is required to achieve and its application.

**Module System**

During the development of complex systems there is a great need for modularity. Modularity provides: (1) means of organization and structurization of the code, (2) ways and support of code reusability, (3) and the possibility of independent development.\(^4\)

The module system would allow the agents developed in Jason\(^+\) to have multiple modules, which would have completely independent belief-base, goal base and plan-library; and each module would be able to react to each of its events on its own, not interfering with any other module of the agent, or with the agent itself.

Communication and interaction between modules would be allowed through specified predicates which are visible from outside the module as well. Similarly a module would be able to access only those predicates of the base agent which are specified to be visible. The base agent of a module is the agent included in the MAS project which directly / indirectly imports it, hence the agent which is at top of the import hierarchy.

To support reusability the module system should be flat using fully-qualified names for global purposes, and short names for local purposes. A flat system means that modules can import and use other modules as well, but each unique module is truly imported only once, and can be used anywhere where it has been marked to be imported. Thus, many different modules – and the base agent as well – can use the same module (hence the same code) and communicate to each other through the mutual module.
As the previous example shows, the module system can be used to organize code as a separate module can be created for selected elements of the agent which describe similar functionalities (e.g.: Module 2 and 3 are the subunits of Agent 1, and Module 2 has an additional subunit: Module 1). It supports code reusability as modules can be made to encapsulate reoccurring patterns of code (e.g.: Module 4 is used by many two other modules and by all the agents), and it also provides the opportunity of team-work (e.g.: Agent 3’s Module 5 and 6 could be developed by different people).

A module should be able to be imported by using a new language entity: the import directive. It should have two parameters which should provide the local short name for the module which can be used after the directive until the end of the source code (of the agent or module) as a module prefix; and also the actual resource / file where the source code of the module can be found, which will form the fully-qualified name of the module. It should be possible to provide this resource as a relative path, an absolute path or as a whole URL; but the fully-qualified name of a module should be an absolute path. Hence, if it is possible to determine that the source files of two modules are the same, then the two modules should be handled as same as well – even if for example one was specified as a relative path and the other as an absolute path. However if a URL is used, it is likely to be needed to specify the same URL at every occurrence, where it is intended to use the same module.

At each module it should be possible to specify the visible elements of the module, and all the other predicates should handled as hidden from outside of the module, hence access to these elements should be denied in other modules or in the base agent (wherever the modules is imported). For such purpose another language entity should be created: the export directive; which should have two parameters: the name and the arity of the predicate. Hence go(X,Y) and go(Place) should be handled as different predicates, and if both required to be visible then both are needed to be exported.

The approach is rather similar to the one in the Modularity and compositionality in Jason paper, the functionality provided is almost the same. However, while in this solution a single belief-base and plan-library is used for the agent and for all the imported modules, in that solution every module has its own separated belief-base and plan-library with a single transition system per agent. However, the modules of the predicates in that transition system are identified with the same prefixing methodology described here. Further, that approach is more similar to XML as the modules are able to specify their fully qualified name within their source code, while the approach used in this project uses the files absolute path as their fully qualified names. It is done so; because if the modules can specify their own fully-qualified names, it is possible that two different modules specify the same name by accident (and not by purpose) causing unpredictable behaviour as it is impossible to be detected by the parser. However using the absolute path solution, it is ensured that the two modules only have the same fully-qualified name if their source file is the same.

This way the module system can provide a safe and convenient way to achieve its main aims, as it gives a well-defined interface between modules. As it is also less likely to cause confusion, because if no module prefix is applied for a predicate it means that it is a predicate within the same source code, otherwise it is always required to specify the short name of a module – which has been defined beforehand in an import directive; which also supports better code readability.
Truly Atomic Intentions

Jason supports atomic plans only to a limited level: in Jason an atomic plan means that no other plan is allowed to have the focus, hence until the atomic intention finishes no other intention is allowed to be selected. However if the atomic plan fails at some point, nothing is reverted back; everything is left in a half-completed state; which is undesired.

The extension of truly atomic plans makes the atomic plans act like database transactions, hence they should act like real atomic actions: if a sub-action fails, the whole atomic action should fail and everything should be in the state before start of the execution of the atomic plan.

However if none of the sub-action fails, the atomic action succeeds, and everything should behave accordingly, hence the state of the agent should be updated as if every sub-action would have been executed individually.

Further, it is clearly not possible to revert actions done on the environment or to determine whether changes in the environment are truly consequences of the current agent's actions, hence these 'external' actions / events are not going to be handled as atomic, but they are treated normally, and as a consequences it remains the agent-programmer tasks to think of such situations; for example a good technique could be to execute the external actions as the very last sub-actions of an atomic intention, however this is not enforced.

Thus it is not required to change the syntax of the language as an atomic notation already exists in the language, but only its function has to be extended to provide a more sophisticated function as described above; and keeping its property of not selecting any other intention until the current atomic intention has finished.

Dynamic Module Import

In a way this task combines the previous two: the aim of dynamic module import to be able to load a module to agent dynamically during run-time. This way, sources of modules can be asked runtime from other agents for example, and these can be loaded and tested during as an internal action in a plan, hence as part of the reasoning cycle of the agent.

Further, since this module import is done during run-time, it is possible to handle differently if a module fails to load: for example a required exported predicate is missing in the base agent, hence a functionality is not supported by the agent; and in this case the agent can detect and handle the failure, for example by trying to load a different module or selecting a different goal.

Hence, it is clear why a dynamic import of a module can be useful. However, these dynamic module imports should work inside atomic plans as well to provide a consistent functionality: hence if a module is loaded during the execution of an atomic intention, and then something later fails in the same atomic intention, then the module should be ‘unloaded’ the same way as the state of the base agent is reverted.
Similarly to static imports, this dynamic import should support loading modules from relative paths, absolute file paths or from URLs, to support a wide range of applications: this way, for example, an agent can ask another agent for the location of the source code of a module which could be located on a server (which is different from the two agents).

Further, this dynamic import should provide a possibility to name the dynamically imported module, however only for debugging purposes. Using the built-in debugging facilities during runtime, it can be useful for the agent-developer to use the given short name of the module, instead of a randomly generated short name or its fully-qualified name.
DESIGN

This section describes the design of each task. It presents a high-level description of the actual implementation, and compares it to other possibilities. Hence it also explains the considered alternatives, the decisions made during the project, and the reasons behind those decisions.

**Module System**

**System Alternatives: Flat System / Hierarchical System**

There were two main alternatives considered for the module system. One is the flat module system which was described in the previous section, where the same files locate the exact same modules, no matter how many times they were imported. This way, the same module imported in two different modules would denote the same module.

The following figure demonstrates the belief-base system of the chosen alternative: the flat module system:

![Modular Belief Base System](image)

The other alternative considered is a hierarchical system, where modules are considered different if they are imported from different modules, even if the imported file is the same. Hence, while the module ‘data’ imported by the module ‘phone’ (‘phone::data::’) and by the module ‘invoice’ (‘invoice::data::’) would be the exact same module in the flat system (but accessible by both modules) – hence their beliefs and goals are shared –, it would be two completely independent modules in the hierarchical system. Thus in a way, the agent would be a hierarchical tree of modules, where each node denotes a different module. This way, the same code and file could be used to act independently; hence it could provide better code reusability.
However, research has showed that the flat system is more advantageous than the hierarchical system, because it provides better communication between the different modules. For example, consider the following: two modules import the same third module, in this situation both modules can generate the same goal achievement event, which should be handled by the module; and this way it is ensured that the third module is in the exact same state in both cases, no matter which module has actually generated the goal achievement event. However, if a hierarchical system would be used, then this ‘third’ (and fourth) module would be a different sub-module for both modules, hence they could be in different states, which could result in inconsistency.

The import hierarchy of the flat and the hierarchical module system of the same Agent.

Description of the Chosen Alternative

Hence, the chosen alternative is the flat module system. There are two new directives created as part of the module system: the import directive, which imports a module (its initial beliefs and goals, and its plan-library); and the export directive which makes a predicate visible from outside the module, hence controls visibility.

The notation of the import directive is the following:

{import shortName="file/path/URL/location.aslm"}

The ‘shortName’ specifies the name to be used within the source code, to denote the module and to access its members with qualified predicates, along with the ‘scoping operator’ of ‘::’. For example: ‘shortName::makeCall(bob)’. This ‘shortName’ can be used after the import directive until the end of the source code. The second (quoted) part denotes the location of the file, which is then converted to an absolute path during parse-time. This absolute path forms the modules fully-qualified prefix which is applied to every single predicate inside the module, and then imported the base agent. Hence, for example the belief shortName::makeCall(bob) becomes ‘/file/path/URL/location.aslm::makeCall(bob)’ and then added to the base agents belief-base. The same thing is done with the module’s plans and the base agent’s plan-library and with initial goals as well.
Similarly, whenever a module is imported, the module's absolute path as prefix will be applied to every predicate used in the module, the same way as in the base agent, when it is explicitly qualified by the prefix of the short name and 'scoping operator'.

Hence the term 'makeCall(bob)' inside the module '/file/path/URL/location.aslm' becomes '/file/path/URL/location.aslm::makeCall(bob)', and therefore it will be used in the scope of that module. Further, if an empty 'shortName' is specified, hence for example '::makeCall(bob)', it means that even if the predicate is used in a module, it should be handled as if the predicate would be used in the importing base agent. Thus, in this case the term 'makeCall(bob)' would have no module prefix applied at all and therefore it would be used in the scope of the base agent.

The character ':' – colon, which forms the scoping operator – cannot be used inside the name of a predicate; hence the only way to apply a module prefix is to use the scoping operator. Therefore, with this solution, no predicates of different modules can interfere with each other, however the same predicates of the same module will become the same fully-qualified predicate in the base agent, independently from the chosen way to provide the file location (relative/absolute) or in which module they were imported.

The notation of the export directive is the following:
{export predicateName/arity[, predB/arityB[, predC/arityC[, ...]]]}

Hence an export directive can export an arbitrary number of predicates, and it is equivalent to as if an individual export directive would have been created for each, this feature is only provided for convenience.

The predicateName (predB, predC) denotes the name / functor of the predicate and the arity (arityB, arityC) denotes the number of arguments. Predicates without arguments can be exported with an arity of 0 (for convenience the ending '/0' can be omitted, the exported predicate will be treated as if the arity of 0 would have been specified). Also for convenience, the wild char of '?' can be used in place of both the predicate's name and its arity, and it will match any predicate names or any number of arity. For example the directive {export ?/0} will export all the predicates without arguments, while the {export ?/?} will export all the predicates of the module. However it should be used wisely; and the named exports are preferred.

Only those predicates can be accessed from outside the module which are exported within the module, hence whenever a shortName::predicate(argument) form is used on a predicate which is not exported in the module denoted by the shortName, an error is thrown and displayed to the user at parse-time.

This way an efficient modular system can be developed, because there is no need for further belief-bases, plan libraries and event handlers for the modules, and neither there is a reason for communication between these – which could unnecessarily slow down the execution significantly in a large, complex multi-agent system. As this way every elements of the modules are actually handled by the base agent's own belief-base, plan-library and event handlers, and hence by the agent's own transition system and reasoning cycle.
Visibility Control Alternatives: Only Qualified Access / Easy Access

There was an alternative visibility control considered as well.

The original version of visibility control is the system described in the previous section, where each predicate of a module can only be accessed by specifying the short name of the module declared during its import. Further, in this case only those predicates can be accessed, which were specifically exported in the module. This system hence provides a solution which is very unlikely to produce an inconsistent behaviour.

The other alternative considered has this exact same functionality as described above, but in addition it could provide an ‘easy access’ behaviour with a similar directive to the export directive. Hence, if a predicate is declared as ‘easy access’ inside a module, it would act as if the module would import that single predicate of the base agent to module; hence as if that predicate would be declared and used inside the base agent importing the module, instead of inside the module. Further, this would allow accessing this predicate from within the agent without the need to specify any short name. Moreover, two different modules could communicate to each other using ‘easy access’ predicates, as both modules would import the exact same predicate of the base agent. For example, the predicate `phone::makeCall(A) – (makeCall/1)`, would become `makeCall(A)`, and could be accessed like that. Hence this system provides exporting predicates in a module to be used in the base agent; and also importing predicates of the base agent to the module to be used in the module.

However, this second system is potentially unsafe as two modules could interfere with each other unwillingly, if they happen to import the same predicate of the base agent, but without considering that it has been imported in another module as well; as this predicate would the exact same predicate in both modules, even though they might have completely different functionalities in the different modules. Hence the alternative, which only allows qualified access, is safer, therefore that one is selected as described, because it still provides comprehensive features without compromising safety.

Truly Atomic Intentions

As it has been mentioned, the notion of atomic plans is kept, and only the functionality of atomic plans / intentions is changed according to the specification.

To be able to provide the desired behaviour, it is required to be able to restore the original state of the agent at the beginning of the execution of the atomic intention, in case it is discarded, and it is also required to commit the changes in case the atomic intention has succeeded.
Alternative of Restoring the Original State

Probably the most obvious solution to this problem would be to save a whole copy of the original state of the agent at the beginning of the execution of the atomic intention, and apply normally the changes made as a consequence of the sub-actions of the atomic intention. In case something fails, and hence the atomic intention has to be discarded the original state is simply restored by replacing the state of the agent with whole copy made. And in case nothing fails, hence the atomic intention has to be committed, there is nothing that is needed to be done as the sub-actions has been handled normally. It is a feasible solution, however making a fully copy of the whole state of the agent could be slow, and probably not that efficient.

An alternative could be to do not make a copy, but keep track of the changes. Hence whenever something changes in the state of the agent, it should be logged. This way, if the atomic intention is needed to be committed, then similarly there is nothing required to be done, as it the sub-actions are handled normally in this case as well. However, if the atomic intention is needed to be discarded; then using the log made, the sub-actions of the atomic intention should be reverted by applying a reverse action (and in reverse order) on the state of the agent. This solution could work, however it is rather complicated, because it is not always clear what a reverse action is. For example if there is a belief addition event, then it is required to check whether the belief is actually added to the agent’s belief-base as it is possible that it has already contained that belief.

Therefore, another alternative is considered: maintaining a delta state of the agent. It means that at the beginning of the atomic intention, an empty delta state is created and registered, and during the execution of the atomic intention, the changes made by the sub-actions are applied on the delta state and not on the actual state. Further, whenever a query is made on the state as part of the atomic intention the query is made on the combined version of the normal and delta state of the agent. This way, whenever the atomic intention is discarded, only the delta state has to be cleared as the normal state is still unchanged. And whenever the atomic intention has to be committed, the normal state and the delta state have to be merged.

This final alternative is efficient, and is not as complicated to implement as the second alternative (where a log is kept), hence this alternative is chosen to be implemented.

Description of the Chosen Alternative

This section will mainly focus on the implementation of the delta belief-base as mainly the beliefs form the main changes of the agent; and as part of this task the non-atomic plan-library / goal addition events are considered as an acceptable limitation of the software. However the dynamic module import has changed that, and in the final version the atomic intentions are fully supported (hence including the plan-library and goal addition events as well); but these are detailed at the Dynamic Module Import task.

The transition system of the agent has to be modified to notify the agent whenever a new atomic intention is selected, this intention has successfully finished or if a failure happens; in order for the agent to be able to make the appropriate actions on the delta belief-base.
Hence as it has been already described, the agent’s delta belief-base should be in an initial (empty) state when a new atomic intention is started, and also the agent should register that it should use the delta belief-base to apply changes instead of the normal belief-base. When the atomic intention successfully finishes, then the delta belief-base should be merged with the normal belief-base of the agent – after which the delta belief-base should be cleared, and when it has to be discarded then the normal belief-base should be untouched while the delta belief-base should be cleared.

Whenever a change is made on a predicate, then that predicate is put into an ignore list as part of the delta belief-base, if it has been already in the normal belief-base, then it is copied into the delta belief-base (only that single predicate) and changes applied on this delta belief-base. This change can include many things: deletion, addition, modification of annotations, etc. Further, whenever a query is made on the belief-base then first, the non-ignored beliefs in the normal belief-base are checked, and then the beliefs in the delta belief-base. The system ensures that a predicate can either be among the non-ignored normal beliefs or be among the delta beliefs, but not both. Hence, if the predicate is found among the non-ignored normal beliefs, that it is unnecessary to check the delta belief-base.

The actions on the delta belief-base should have the same results as the actions on the normal belief-base to be able to provide accurate queries on the merge of the two belief-bases. To be able to permanently merge the two belief-bases the actual belief additions / deletions are logged separately; and these are applied again on the normal belief-base, when it is required to commit the changes caused by the sub-actions of the atomic intention.

The following shows a simple example:

<table>
<thead>
<tr>
<th>Action</th>
<th>Normal Belief-Base</th>
<th>Delta Belief-Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove b1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove b2</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>add annotation b3</td>
<td>b3[a,b]</td>
<td></td>
</tr>
<tr>
<td>remove annotation</td>
<td>b4[a,b]</td>
<td>b4[a,b,c]</td>
</tr>
<tr>
<td>add annotation b5</td>
<td>b5[a,b]</td>
<td>b5[b]</td>
</tr>
<tr>
<td>add b6</td>
<td>✗</td>
<td>(only ignored) b6</td>
</tr>
<tr>
<td></td>
<td>b7</td>
<td></td>
</tr>
</tbody>
</table>

*crossed* means it is in the ignore list

The newly added *b6* has to be added to the ignore-list, in case later – as part of the same atomic intention – it is still modified, for example an annotation is added / deleted.
Dynamic Module Import

Alternatives of Specifying the Module to be Imported

There were two options considered, how the actual modules to be imported could be specified. One of the alternatives is to specify the location of the file containing the source code. In this case the agent importing the module has to have access to the file and it must be able to read it, however it would only need this file path or URL.

The alternative solution considered is to specify the whole source code of the module. In this case the agent does not need to locate / read files, however it is required to gain this source code in some way.

It is possible that another agent could tell this source code, but if this is the case, then it is also likely that this agent could provide an accessible URL to this source code as well. Moreover, the agent would need to transmit the whole source code instead of a location, which could be on an independent server. Further, in other applications it is also inconvenient, that it is required to provide the whole source code. Hence the alternative of only specifying the location of the source code’s file is chosen, as it is more convenient and more generally useable.

Description of the Chosen Alternative

To dynamically import a module, the newly created internal action .import (or .import_module) can be used. The syntax of this action is the following:

```
.import("<file path / URL>")
```

```
.import(<module_name_for_debugging>, "<file path / URL>")
```

Hence it is possible to specify a short name for the module, however since the module is dynamically imported no reference can be made to it, thus it is only used when the predicates of the module are displayed for debugging purposes.

The file of the path or URL specified is located, read and parsed and then the elements of the module are dynamically added to the agent; hence the appropriate events are generated. The initial goals of the module are added as with new focus (the ‘!!’ operator of Jason). Once the import finishes, the execution of the plan continues normally.

If something goes wrong during the import – for example the file could not be located or read; or a required exported predicate of the base agent (which imports the module) is missing –, then the whole module import is cancelled and an action execution failure is generated. It is useful to dynamically handle module load failures, or using dynamic imports as part of atomic intentions.
Extension of Truly Atomic Intentions

As it is described in the Truly Atomic Intentions section, the main focus of that task was on the belief-base; however with the dynamic module imports, the plan-library and goal addition events are as important as the belief-base.

Hence it is required that atomic intentions support a delta plan-library and a delta goal addition event queue. This goal-addition event queue is somewhat easier as initial goals are considered as new focus goals; hence they are only required to be added at the end of the atomic intention anyway, hence it is possible to simply handle it by another queue storing the delta information.

The delta plan-library however is somewhat more complicated. The solution chosen is similar to the one used during the delta belief-base. There is an ignore list containing the list of plans which are changed or removed from the normal plan-library, and there is a delta plan-library of which actions have the same of effect on the delta-plan-library as the same actions on the normal plan-library; in this way it is possible to make accurate queries on the plan-library. The ignore-list is based on the labels of the plans, as these are guaranteed to be unique for each plan. If there is a change in the plan-library, then if the plan added/removed/modified was already in the normal plan-library, then it is added to the ignore list, and the modified version is stored in the delta plan-library. Whenever a plan is requested, firstly the non-ignored plans are checked and the applicable plans are selected in the normal plan-library, and then in addition the delta plan-library is checked and the applicable plans are selected from that as well. The two lists merged and the resulted list is returned.

Similarly to the delta belief-base, if the delta plan-library must be discarded, then the original plan-library is untouched, and the delta one is cleared; however if the delta-plan-library is required to be committed, then the two plan-libraries are merged based purely on the content of the delta-plan-library.
IMPLEMENTATION

This section outlines how the Jason+ extensions described in the previous chapter were implemented. Further, at the end of the section, other not explicitly specified implementation tasks are detailed, which provide further improvements of the agent programming language and its tools.

Module System

Syntax of the Language

To implement the module system of Jason+, the actual syntax of the language had to be modified, to provide notations for import modules, export predicates (hence visibility control), and to actually be able to qualify the predicates used, thus to be able to indicate their originating module.

The parser of Jason was created by JavaCC, a Java Parser Generator, which is a sophisticated tool designed for developing languages in Java. It has many features including custom token creation by regular expressions and explicit look-ahead specifications (meaning, how far in advance should the parser check for other possible parsing routes).

Therefore, the appropriate .jcc file was modified, which defines the syntax of the AgentSpeak language files (.asl); and there was no need to modify the syntax of the MAS project specification language. The language has been modified the support the 2 new directives, which have special syntax: normally a directive looks like “{<predicate>}”, where <predicate> can be any predicate allowed by the syntax of the language, hence for example simply an atom like '{begin}', or a predicate with some parameters in parenthesis:

'\{register_function("myf.sum",2,"sum")\}' or
'\{include("file.asl")\}'. However as it has been described in the design section, the directives of import and export have different syntaxes, and hence the language has been modified accordingly, to support these.

The import directives are placed in the source file which would like to import modules, and its syntax is the following:

importive ::= "\{ "import" <atom> [ "=" <string> ] "\}"

For example: '{import phone="lib/phoneSystem.aslm"}'

The second parameter is optional, because if it is missing, it is assumed to be a file with the name of the module and with the standard extension in the same directory as the agent.

The export directives are placed in the imported modules, to control the visibility of its own predicates, their syntax follows:

exportive ::= "\{ "export" <pred_ind> ( "," <pred_ind> )* "\}"
pred_ind ::= ( <atom> | "?" ) ["/" ( <number> | "?" )]
For example: `{export speakTo/1, activated, agentCalled/1}`

Hence multiple exports can be defined within the same export directive, and a single export is described by a predicate indicator. The first part of the predicate indicator is the predicate’s functor and the second optional part is its arity. If the arity is omitted, it is assumed to be 0.

Further, in addition to the directives, the support of a module name and a ‘scoping operator’ before a predicate has been implemented as well. Thus, it is possible to qualify the predicates in the source code importing a module, in the way described in the Design section. Hence the syntax of the functor (the name) of the predicate has been modified to the following:

```plaintext
functor ::= [ [ <atom> ] "::" ] <atom>
```

For example: `phone::activated`.

Where the first atom specifies the short name of the module, and it is optional, because the lack of module name indicates, that the predicate is of the base agent, has no module prefix should be applied to it. Similarly the whole prefix is optional as well, because it simply defines the predicate in the normal way. And finally the third atom specifies the actual (non-prefixed) functor of the predicate.

The BNF of the relevant parts have been described in this section, however the actual full BNF for the whole syntax can be found in the doc directory (`AS2JavaParser.html`).

**Actions at Parse-time**

As the syntax parsing is performed, the appropriate actions which should handle the currently parsed token(s) are also performed at the same time.

**Import Directive**

After an import directive is parsed, the Directive Processor is invoked. It identifies which directive has been used (the import) and calls the appropriate handling process. Then the import process will parse the file specified and it will temporarily save the parsed data for the actual import which is done later. This is done recursively, hence if a module imports another module (which is allowed), then that file is going to be parsed as well, and that file’s parsed data is going to be temporarily stored as well, unless that file has been already parsed during a previous import perhaps in another module. Since imports are done recursively as a depth first search, to prevent an infinite loop of imports, if a loop is detected (hence a file is tried to be imported which is currently under parsing), then instead of completely trying to parse that file again (which would result in an infinite loop), that file is re-read, and parsed, but this time only for exports – hence no imports are done during that parse; because that is sufficient information to continue the parsing of the source file which imports that module (see later). See the figure on the next page.
The import process first checks the path/URL parameter of the directive (or generates from the name of the module if it is missing) and locates the file: if it is a URL or an absolute path, then just simply checks whether it exists, accessible and readable. If it is a file, then tries to find it in the directories specified by the MAS project, starting with the base agent’s directory; once an existing, accessible and readable file is found, the search stops. If no such file found or the file specified by URL / absolute file path is not such a file, then a parsing error is thrown which helps the user identifying the location of the error. Similarly the name of the module is checked as it is required to be unique per source file used; and if it is not unique, then a parsing exception is thrown.

After an accessible and readable file is located, its absolute path is generated, as it is going to be used as the unique identifier of the source file. The identifier is used with two purposes: once, it is used as the true / fully-qualified prefixes of the modules, as it is explained in the Design section. Its other purpose is to actually be able to distinguish the modules from each other: 1) to detect loops as it has been already explained and 2) to detect already imported modules. If such a module is detected, it means that the parsed data is already saved (including the initial beliefs, goals and plans and the results of the export directives too), hence it is unnecessary to parse the file again, thus only a new reference is created to this parsed data, with the new (possible different) short name of the module.

However, if a new file located, it is read and parsed for lexical data, hence for the initial beliefs, goals and plans; but it is not yet imported to the base agent, it is only temporarily stored, to be used by the actual import mechanism later. However, the module is already registered within the source file in which it was imported, in order to be able to use it at parse-time in expressions following the import directive: with the short name of the module the results of the export directive are stored as well, because this information is needed whenever a scoping operator is parsed. Thus for each source file, it is required to first import the module before it is actually used (with a scoping operator).
Export Directive

The export directive is rather simple. It simply registers the supplied predicate indicators with the agent / module in which the directive can be found, because this data can be accessed from the possible source files which import this module. There is no need for further checks at this time, as the directive’s requirements are already defined as part of the syntax of the language.

Scoping Operator

There are two main actions performed after a predicate with a scoping operator is parsed: the first one is checking for validity, and the second is actually applying the actual / fully-qualified prefix of the module to the predicate.

Checking validity, involves first checking whether a module has been actually imported with the short name specified for the predicate. If no module is found with that short name, then a parsing error is raised, showing the user the location of unknown module name. However, if the right module is found a second check is performed (still at parse-time): it checks whether the predicate tried to be accessed is actually visible, hence whether it has been exported in the module imported. As it is already explained, a predicate can be exported either by explicitly specifying the actual functor and arity of the predicate, or by using the ‘?’ wild char. Therefore it checks both possibilities, and if no such export term can be found, a parsing error is shown to the user.

Non-modular / Reserved Terms

There are some non-modular / reserved terms in the language which are by default handled as there would not be any module system, hence as if they were used in the base agent even if they were actually referenced in one of the imported modules.

Non-modular terms include the true / false literals, the list literals (not the terms inside the list, but the list itself), the standard / user-defined internal actions (those containing a dot ‘.’, hence not the Environment’s actions), the string literals, the number terms, the arithmetic operators – which also represented as terms, but clearly they have no meaningful functor or they should not be modular.

And the following lists the reserved terms of the language: source (with arity of 1), atomic, breakpoint and all_unifs are special annotations; and the followings are the terms used as parameters of the internal actions which handle the communications between the agents (e.g.: .send): tell, untell, achieve, unachieve, askOne, askAll, askHow, tellHow, untellHow.

So these terms are not meant to be modular, and thus no module prefix is applied to them, unless it is explicitly specified (e.g.: shortName::atomic) for some reason; however in this case the user should be warned that these terms normally have a special meaning and therefore no prefix will be applied to them even if they are inside an imported module. Therefore, after a module-prefixed predicate is parsed, it is checked whether the predicate is one of these special terms, in which case a only a warning is displayed to the user, as it is unlikely, but possible that it is used for a custom purpose – for example: ‘source("text/x-java-source", ~/application.java)’ which has an arity of 2, hence it does not interfere with the predefined annotation source.
Actions at Import-time

The functionality of the export directives and scoping operator is actually done during the parsing – the visibility is checked and the prefix is applied right after the parse of a module-prefixed predicate. Thus, no other actions are required to be done concerning these; hence the only remaining main actions is to actually import the data of all the imported modules (which is now, however already parsed). Thus these modules include the directly imported modules from the base agent, the modules indirectly imported, hence not from the base agent, but from another module.

The import process includes first applying the module prefix to every single modular term of the module, meaning prefixing its functor with the unique identifier of the module, hence by the absolute path / URL of the module’s source file; but only its functor is prefixed, as its parameters and annotations are represented as different terms, hence their functor is going to be prefixed as well (if they are modular). This way, every single term is checked in a recursive way, no matter what actual function it might have: for example belief, goal, plan-label, annotation or just a parameter of a belief.

Further, the import process – similarly to the scoping operator – checks, if a term is non-modular / reserved, in which case it simply leaves its functor without the module’s prefix, in the same way as if the predicate would be used in the base agent. If however, a reserved term’s functor is used, but not at the right place (e.g.: a reserved annotation as a belief or a parameter; or a reserved parameter as a belief or annotation) or not with the right arity (e.g.: source with an arity of 3, \texttt{source(user,defined,annotation)}), then the user is warned about that term might intended to be non-prefixed, but because it is not a real reserved term, a prefix has been applied to it anyway.

The second action of this import process is to actually add the already prefixed initial beliefs, goals and plans to the base agent. It is done relatively simply by iterating through the stored elements and by adding to the agent as if they were defined there with the exception that they have a prefix.

Pretty Module Prefix

During debugging (e.g.: in the Mind Inspector) Jason\(^+\) displays the short names of the modules that they defined in the source code.

To be able to achieve this functionality, whenever a prefix is applied (either because of the scoping operator, or because the predicate has been used inside an imported module), the original functor and the short name of the module are saved. The case is simple when every module is directly imported from the base agent (hence the modules do not import other modules), but there are complications when there are indirectly imported modules as well. Consider the example following on the next page.
Let \( A \) be the base agent, and \( P \), \( Q \) and \( R \) are three different modules.

\( A \) has the following imports:
\[
\begin{align*}
\{ \text{import ap=}"P" \} \\
\{ \text{import aq=}"Q" \}
\end{align*}
\]

And \( P \) has the following imports:
\[
\begin{align*}
\{ \text{import pr=}"R" \} \\
\{ \text{import pq=}"Q" \}
\end{align*}
\]

The other two modules have no imports.

\[
\text{Import Hierarchy}
\]

In this situation, \( R \) is imported under the short name of \( pr \), but only in module \( P \), hence \( pr \) is not a valid short name of a module in the base agent \( A \), and thus the prefix \( \text{`pr::'} \) would not be that useful during debugging. To solve this problem, the debugger should display these predicates with the prefix \( \text{`ap::pr::'} \), and it can be done simply with the depth first search approach: whenever a module is completely parsed (hence its modules are parsed as well) the complete path of short names can be saved as well along with the other parsed temporary data of the module.

However the module \( Q \) can be accessed with two different paths of short paths, namely \( \text{`aq::'} \) and \( \text{`ap::pq::'} \), and the depth first search solution would apply the second (longer) prefix, even though the shorter, direct prefix is more desired. So far, the reference to the temporary data of the module contains the short name of the module and its unique identifier (to be able to use inside the file which imports it), and the whole path of short names (to be able to use in the debugger). However, to solve the problem of multiple identifiers these references are stored in a priority queue along with their levels, in which those ones preferred which have lower levels, and the actual import of modules is done according to the priority queue. Every module is imported only once (the later imports are simply skipped), hence only the first pretty prefix is used, which has to be one of the shortest path of short names. This way the modules are actually imported as if they were parsed in a breadth first fashion, instead of the depth first. If a module has been imported under two different, but same-levelled paths of short names, then a single one of them is arbitrarily chosen.
Rule to Function

Through the register_function directive, Jason provides the functionality of defining custom arithmetic functions. This is possible in two ways: one is to define the function in Java, in which case the directive takes a single argument specifying the name of the Java class, and in this case the name of the function is going to be the name of the class as well. The other way is to provide an implementation in AgentSpeak as rule. In this case the directive takes 3 arguments: the name of the new function to be created, its arity and the name of the defining rule. Then the rule is converted to an arithmetic function assuming that the last parameter of the rule notes the result of the function (hence the rule’s arity must be at least one more than the arity of the arithmetic function).

For example, if the following rule is defined:

\[\text{sum}(X,Y,Z) : Z = X + Y.\]

Then the arithmetic function it represents can be defined with the following directive:

\{register_function("myf.sum",2,"sum")\}

In the case of Java implementation, there are not many complications as the same Java class could be used from all the modules. However, in the case of AgentSpeak implementation, there were some modifications needed to be done, to be able to use it inside modules as well.

First of all, the name of every registered function has to be unique in the agent, hence it is required to apply the module prefix to the name of the functions as well (and thus to every reference of the function as well) in case two different modules define two different functions, but with the same name. The names of these functions are not represented as terms; therefore applying the module prefix is done separately from the terms; however the references (the actual uses of the defined functions) are represented as terms so there is no need to take further actions handling those.

Secondly, when the actual function is evaluated, it needs to use the agent’s belief-base in order to be able to retrieve the rule; which is at that time is already prefixed with the unique identifier of the module. Therefore, during evaluation the literal created as part of the evaluation mechanism of the function has to be prefixed as well.

This way, both the name of the function, and the rule used is prefixed as well, and hence it can be used safely inside modules as well.

Truly Atomic Intentions

As it has been already described, Jason already had a notation which was designed to mark plans as atomic; and hence only the functionality is needed to be changed. The existing notation is an atomic annotation given to the plan’s label. For example:

@label[atomic]
+!start <= +b; ?g.
The previous meaning of this notation was that no other events were allowed to be processed and no other intentions were allowed to be selected (except its subgoals), until the intention described by the plan is finished.

**Transition System**

Therefore, the first implementation task was to locate these atomic plans in the reasoning cycle and transition system of the agent, in order to be able to extend the meaning of this existing feature.

There are 3 main events which are required to be located in order to be able to implement the system described in the Design section: 1) when a new atomic intention is selected, 2) when the execution of an atomic intention has been successfully finished, and 3) when an atomic intention has been dropped, because of some failure event.

**New Atomic Intention is Selected**

When a new atomic intention is selected (until the intention is finished or dropped) the delta belief-base should be activated and used along with the agent’s normal belief-base. There are two cases when this event happens, and both are during the addition of new intended means (a single intention / subgoal of the whole intention-stack).

In Jason, there are two ways to create a goal achievement event: 1) when creating it as a subgoal of the current intention (the ‘!’ operator) – in this case, this subgoal first must be achieved before the next step of the intention can be executed (this way an intention tree of subgoals can be drawn), and 2) when creating it as a new focus (the ‘!!’ operator) – in this case, the goal as new focus have to be achieved after the atomic intention is successfully finished. If the second one is used in a non atomic intention, then either the current intention is continued (just like in the atomic intentions) or the new intention is started (just like if it would be a subgoal) depending on the intention selection function.

Achievement events created as new focus are easily handled as they cannot be started anyway, until the atomic intention has been successfully finished. However the handling of subgoals is a bit more complicated, because even though an atomic intention is under execution, new sub-intentions are selected.

Two cases are checked: 1) when a new intention-stack has to be created to be able to select and add the new intended means (a node in the intention-tree / intention-stack), in which case it is only required to check whether that one is atomic; and 2) the second case is when there is an existing intention-stack and a new node is selected and added to the stack, in which case, it is required to check both that the intention-stack has no atomic nodes before and the added intended-means is atomic. This way, it is possible to check whether the execution of a truly new intention has just been started.
Atomic Intention is Successfully Finished

Similarly to the addition of new atomic intention, it is required to report only events when an atomic intention has been truly finished, and not just a sub-node (intended means) of an atomic intention-stack. An intention-stack is considered to be atomic, if it has at least one intended-means which is atomic.

Therefore, whenever an intended means is finished, two checks are performed: 1) whether the removed intended means was atomic, and 2) whether the remaining intention-stack is now non-atomic. If both checks are successful, then the successful finish of the atomic intention is reported to the agent, in order so that it can merge its delta belief-base to its normal belief-base, and it can start using its normal belief-base only.

Atomic Intention is Dropped (because of a failure)

Because the transition system already had a special mechanism for generating a failure event whenever a goal is required to be deleted, it was relatively simple to report these failure events to the agent’s atomic intention handling system as well, in order so that the agent can discard its delta belief-base and it can start using only its normal belief-base instead. Hence, whenever such a failure event is generated, the transition system reports it to the agent too.

Delta Belief-Base

The already existed implementation contained the actual belief addition / deletions methods inside the Agent; further even though the belief query methods have been placed elsewhere, they had the Agent as an argument. Therefore it was more logical to place the delta belief-base inside the agent separately for easier access (and not inside the agent’s normal belief-base).

Belief Addition / Deletion

The belief addition / deletion function (Agent.brf) first checks whether the execution of an atomic intention is taking place, hence whether the delta belief-base system is activated. If it is not, then everything is handled normally. However if the delta belief-base is activated, then the handling is passed to the delta belief-base; which does two actions: 1) registers the concrete addition / deletion actions performed to be able to easily reproduce them if these should be committed 2) makes the changes in the ignore list and in the delta belief-base.

Belief Addition

If a new belief is added while the delta belief-base is activated, then it first checks whether the belief is already in the normal belief-base. If the belief is found in the normal belief-base, then it means that the action is not a true addition, but a modification action (e.g.: only an addition of a new annotation), therefore the existing belief is added to the ignore list; and copied to the delta belief-base without modifications as it is handled by the second step.
Secondly, then the delta belief-base handles the addition event in the same way as it was handled in the normal belief-base. Therefore, if it is truly a new addition event then the belief is simply added to the delta belief-base; but if it is a modification action, then the belief now must be included in the delta belief-base and be placed on the ignore list (because of the first step), hence the modification is done on the copy held in the delta belief-base.

**Belief Deletion**

The belief deletion is done similarly to the addition: it was already included in the normal belief-base, then the belief is added to the ignore list, and copied to the delta belief-base; and the further actions (true deletion / or just simple modification) is handled in the exact same way as it was handled in the normal belief-base, but it is performed on the delta belief-base instead of the normal one. Hence, if it is a modification action, then the modification is done on the copy held in the delta belief-base; or if it is a true removal action, then it is removed from the delta belief-base, hence from the whole merge of the two belief-bases as well, because the belief is either included in the ignore-list or was never part of the original belief-base.

However, there is one more required step to be performed during deletion as it is possible to delete beliefs by using a pattern defined by variables (e.g.: \( \neg b(x, y) \) or \( \neg b(X, Y) \)) instead of the explicit belief definition, in which case the first matched belief is modified / removed from the belief-base. Therefore, similarly to the normal deletion, first it is checked according to the atomic belief query (see later), whether the belief to be modified / removed is a logical consequence of the merged belief-bases, in which case the unified belief is attempted to be modified / removed. Hence, in other words, a search is made for a belief which matches the given pattern (in the two merged belief-bases), and if such a belief is found, then that one is modified / deleted.

**Belief Query**

In principle, in Jason, every query made to the agent’s belief-base can be thought of as a logical consequence check. It means, that if the agent would like to determine whether it believes a belief ‘b’, then it will check whether this belief ‘b’ is a logical consequence of its belief-base or not.

Therefore to support atomic belief-queries the method (Literal.logicalConsequence) determining whether a literal is a logical consequence of an agent’s belief-base had to be modified. Originally this method is iterated through the literals contained by the normal belief-base, and these literals are matched against the tested belief. To implement atomic belief-queries this iterator is modified in the following way to do not include beliefs that are in the ignore list, but to include beliefs that are found in the delta belief-base.

The iterator basically goes through the joint list of the delta beliefs and non-ignored normal beliefs. First the delta belief-base’s iterator is initiated and each applicable literal is selected and returned from it, until no more literals left. Once this point is reached, then the normal belief-base’s iterator is selected and each element is checked whether it is included in the ignore-list, and the elements which are not included are selected and returned. Once all the elements of the normal belief-bases are checked, a null value is returned, indicating that the iterator has no more elements.
This way, the joint iterator can have the same effect as if the two belief-bases (the normal and the delta) would be merged, at this merged belief-base’s iterator would be returned; however this simulated solution is much more effective, as it is not required to create another belief-base and perform a merge every single time when the delta belief-base is modified and a query is made.

**Keeping the order of beliefs**

The order of beliefs in the belief-base matters – for example because a patterned belief-deletion action removes / modifies only the first applicable belief, even if there are multiple matching beliefs. Hence it is also important that the order in which the simulated merge returns the elements matches the order in which the elements would be returned normally.

To be able to accomplish that, the ignore list, not just simply contains the beliefs which are ignored, but they also contain either a reference to the new belief in the delta belief-base or null value indicating that the element has been already removed an newly added in which case it should appear in the beginning of the list as new elements are inserted at the beginning of the list. Similarly if an element appears in the delta belief-base, but not in the ignore list, it indicates that the literal is newly created, hence it should appear in the beginning of the list as well; which is simply achieved by the joining iterator first processing the delta beliefs, and then the normal beliefs.

Then this additional information stored in the ignore-list is used to create the desired order in the following way: when the delta belief-base is processed those elements are ignored which have actual delta belief-base references (hence not a null value), because it indicates that these values should be returned only during the process of the normal belief-base as their true position is among those. Therefore, when the normal belief-base is processed, when an ignore-check returns that the literal should be ignored, it also checks in the ignore-list whether the literal is included in the delta belief-base – thus it has a reference to this new literal – in which case the normal belief is not simply ignored, but the literal in the delta belief-base indicated by the reference is selected and returned.

This way the joint iterator returns the candidate beliefs in the same order as if they would be returned during the execution of a non-atomic intention.

**Dynamic Module Import**

Many actions required for dynamic import have been already implemented during the previous two tasks; therefore (after some refactoring) these could be reused. For example, the location of the actual import-file, the unique module identifier determination, the lexical parsing of a module and the prefixing mechanism is already implemented; therefore the remaining required implementation tasks are: 1) the process doing the actual imports is required to be modified to support dynamic imports during run-time as part of the reasoning cycle along with some refactoring to prove code reusability, 2) an interface (.import) is required to be provided to be able to access this dynamic import 3) some extensions of the atomic intention handling system is required to be made as described in the Design section.
**Internal Action – ‘.import’**

Jason already provides an easy way to create new internal action by creating a class with the name of the action in its stdlib package. However the word ‘import’ is a reserved keyword of Java, and therefore it is impossible to create class with such name. Hence, a new internal action ‘.import_module’ is created in the stdlib package, and the internal action ‘.import’ is redirected to that one.

The created internal action when invoked first checks its number of arguments. If it has only a solo argument, then it must be the file path / URL specified as a string literal. In this case a unique module name is generated by using the file name part of the given path / URL as basis. If the action has two arguments, it means that the first one must the name of the module specified as an atom and the second one is the file path / URL specified as a string. In this case the unique module name is generated by using the given name as basis. In both cases, the generated module name is only used in the debugging tools of Jason+ (e.g.: Mind Inspector) to display that short and memorable name given by the user, as no external reference can be made to modules imported dynamically. Further, the generated module names start with an underscore (‘_’), in order to show the user that the modules were imported dynamically during run-time by using an internal action and not statically by using the import directive.

Once both the unique short name of the module and its file path / URL is determined, the already existing import process is called with these as parameters. Then this process locates, reads and parses the module’s source file and saves the parsed data temporarily. If an error occurs, the import is cancelled, an error message is logged, the internal action fails, and a failure event is generated.

**Dynamic Import**

After the module’s source file is successfully parsed and prefixed with the module’s unique identifier, the temporarily saved parsed data is dynamically imported: 1) the initial beliefs of the module are added to the agent’s belief-base according to its rules (hence if the import is done during the execution of an atomic intention, then these are added to the delta belief-base) and a new belief addition event is generated for each, 2) a new goal achievement event is generated as new focus (‘!!’) for each of the initial goals of the module, 3) all the plans described in the module are dynamically added to the agent’s plan-library, and 4) all functions registered in the module are re-registered in the agent (but now with a module prefix).
**Delta Plan-Library**

As it is described in the design section, with the dynamic import of modules, there is a real need to support plan additions (and removals) in the atomic intentions, and to be able to do that, a delta plan-library is created and used. The created delta plan-library supports the exact same events of the transition system as the delta belief-base; hence the transition system was not required to be modified this time. Namely the events and the corresponding actions in the delta-plan library are: 1) new atomic intention is selected → the delta plan-library is initialized and activated, 2) atomic intention is successfully finished → the delta plan-library is committed, 3) atomic intention is dropped because of a failure → the delta plan-library is discarded.

Because the plan addition / deletion and the plan retrieval methods are implemented in the plan-library (and not in the Agent), the delta plan-library is placed inside the normal plan-library. Further, similarly to the delta belief-base, whenever the execution of an atomic intention is done (hence the delta plan-library is activated) the plan addition / deletion and retrieval tasks are delegated to the delta plan-library otherwise they are handled as previously by the normal plan-library.

**Plan Addition**

The added plan is always included in the ignore-list as the ignore-list does not simply contain the label of the ignored plans, but also stores the information whether the added plan replaces another plan (hence a modification action), it is inserted at the beginning of the plan-library or it is inserted at end of the plan-library.

However it is still required to check whether the plan added has been already existed in the normal plan-library, because in this case the plan addition action becomes a plan modification action. To handle it, the plan found in the normal plan-library is copied to the delta one, and only this new copy is modified. Otherwise, the new plan is simply added to the delta plan-library in the same way as plans are added to the normal plan-library.

Similarly to the normal plan-library, if a plan with an existing label is tried to be added, then an exception is thrown, hence a new failure event is created during the dynamic import.

**Plan Removal**

Plans can be removed by using the existing internal action `remove_plan`, and it is handled similarly to the plan additions: if the plan to be deleted is found in the normal plan-library, then it is included in the ignore-list; if the action is truly just plan modification, then the plan first is copied to the delta plan-library and it is modified only then, otherwise if it is a compete removal, then nothing is added to the delta plan-library. However if the plan to be removed is found in the delta-plan library, then it is handled (modified / removed) in the same way as in the normal plan-library. If the plan was not found in either plan-library, then nothing is done – as there is nothing to remove.
Plan Retrieval

Plan retrieval is done similarly as the queries are made to the normal / delta belief-bases: joint lists of plans and joint iterators of the normal and delta plan-libraries are used.

During specific plan retrieval, first the normal plan-library is checked for the applicable plans not selecting those which are included in the ignore-list, and then the delta plan-library is checked for applicable plans. This way a joint list of plans is created.

Similarly when an iterator of plans is needed to be created, first those plans of the normal plan-library are selected and returned which are not included in the ignore-list, and then all the plans of the delta plan-library are selected and returned.

Commit

When the delta plan-library must be merged with the normal plan-library the information stored in the ignore-list is used to achieve the same order of plans which would be produced during the execution of a non-atomic intention: first those plans of the normal plan-library are replaced with the copy stored in the delta plan-library which have been only modified; while in the same time the plans which have been truly removed are also deleted from the normal plan-library. After removal and replacing actions are performed, the newly added plans are inserted to the normal plan-library at its beginning or at its end according the information stored in the ignore-list. This way it is possible to achieve the same order that would have been created if the plan would be non-atomic.

Delta Goal Addition Events

The support of goal additions during the execution of atomic intentions has been implemented as well in the following way: if the event was generated as a subgoal, then it is handled normally and hence triggered instantly; however if the event was generated as a new focus, then the event is added to a list of pending events, and is not yet triggered as new events cannot be selected during the execution of atomic intentions anyway.

Whenever the an atomic intention fails, the list of pending events is simply cleared without commit; and whenever an atomic intention succeeds, then the events stored in the list of pending events are processed and hence triggered.

Other Improvements Made

Apart from the main tasks described in the previous sections there are some smaller improvements have been implemented as well. Some of these improvements are fixes of undesired / unintelligent behaviours of either Jason or the plugins interfacing Jason. And some of these are not just simply fixes, but extensions that actually provide additional features of the piece of software developed. The followings list some of these improvements made.
Better Integration of the Module System

To support the agent-developer even more, the new features provided by the module system are now accessible through the GUI provided by the plugins.

Plugin for Eclipse

A new button ( ) was placed on the toolbar for new modules. It opens a new dialog where the details of the modules can be specified, and it eventually creates a new module by using a customizable template. The module is also registered within the agent if specified.

Plugin for jEdit

A new wizard was made to support the creation of new modules. The wizard asks for the details of the module to be created and checks the given parameters. Similarly to the jEdit plugin, the new module is created based on a customizable template, and it is registered within the agent if it was specified.
Support of URLs in the Include directive

Previously Jason did not handle properly the solo argument of the `include` directive, as if a proper URL was tried to be included, it was still assumed to be a file path. It is now fixed by using the same file locating method used in the `import` directive.

Proper Handling of String Literals

String Literals can be created in Jason by enclosing a text with double quote characters ("""). It was possible to escape characters in these (e.g.: ‘"quote: \" backslash: \\ string"’), and this way it was possible to include special characters without causing parsing error, however these were interpreted as is (in the case of the previous example: ‘quote: \" backslash: \\ string’) instead of their desired escaped meaning (in the case of the previous example: ‘quote: " backslash: \ string’).

It has been fixed by actually parsing these string literals to support the escaped meaning; the following escaped characters are supported:

`\n` → new line feed \n
`\t` → (horizontal) tabulator \n
`\b` → backspace character \n
`\r` → carriage return \n
`\f` → form feed \n
`\'` → single quote character (’’) \n
`\"` → double quote character (”")

And it is also possible to specify a character by its octal character code in the ASCII table, for example:

\60 → zero (’0’) \n
\101 → capital A (’A’) \n
\141 → lowercase A (’a’)

Syntax Colouring of jEdit

As the main release of Jason is provided with a plugin for jEdit, it is important that its syntax colouring is accurate. It has been updated and fixed to show the different kind of tokens which have different function in a way which truly respects the actual parser of Jason.

Moreover, this new syntax colouring now also supports the new `import` and `export` directives and the scoping operator (’::’) too.

More Detailed Error Messages

In general, the error messages provided contain more information, which helps to identify the location and nature of the caused error, in order to support the development process of the agent-programmer. For example the name of the source file is now always part of the error messages, and in most cases the actual token location is included as well.

Better Error Identification

To support the debugging process, now the GUI displays actual location of the errors detectable at parse-time. It includes lexical errors, parsing errors, usage of hidden predicates or even if an imported or included file could not be found or read.
Plugin for jEdit

While the plugin for Jason only shows the line which contains the error, the plugin for the new Jason+ also shows the actual location of the token which is likely to be the cause the error.

Plugin for Eclipse

The plugin for eclipse already had that feature; however the error message was not displayed in case the mouse was moved over the error (it was displayed only in a list in a separate window). It now has been fixed, and the error message is displayed both in the list and over the located problem.

New Branch – Jason+

This one is not really an improvement, but because the project was developed as a different branch under a different name it is displayed accordingly (hence Jason+ instead of Jason where it was appropriate), in order to do not cause confusion.

Further, the website's URL and the update mechanism have been amended as well to query the website of Jason+ instead of Jason whenever a check is made for a new version of the software.
EVALUATION

This section contains the evaluation of the created piece of software, hence the evaluation of the result of the project and the evaluation of the project itself, because that one can be found in the Discussion section.

Performance

Because of the nature of the project it is hard to make a proper evaluation. It is partly because it is relatively difficult to distinguish the existing parts – in the running application – and the newly created parts. And it is partly because it would require a whole group of testers to extensively use the application by developing complex multi-agent systems as the purpose of the created piece of software is to support the development of these. And please also bear in mind that even if these sessions of tests would have been performed, then a fail in them could indicate an error in the specification rather than in the implementation, as they test the usability of the tool.

Nevertheless, the following measurements were made to test, but the efficiency (speed and memory) of the implementation of module system and atomic intention handler. However these measurements only test the performance of the system, but not its usability.

Module System

In this test, the ‘modules’ example has been used for input. To test the efficiency of the module system the results of the following two scenarios are compared: 1) in the first scenario each of the agents are parsed normally according to the rules of the newly created module system, and 2) in the second scenario each of the source files (agents and modules) were parsed separately, hence nothing was imported. This way, the difference between the used resources would show the how much resource was used by the module system. For each scenario 5 test cases were made on a single core of computer with CPU power of 2.33 GHz and with 6.0 GB of amount of memory. The average of the measurements were calculated and used in the comparison of the two scenarios.

First Scenario – normal module system

Time taken: 0.1450 s  
Maximum memory used: 22308 kB

Second Scenario – no import, files parsed separately

Time taken: 0.1192 s  
Maximum memory used: 20496.8 kB

Evaluation

The time difference is 0.0258 s, which is about 18% of the time taken for the execution of the whole parsing. And the memory difference is 1811.2 kB which is about 8% of the memory used by the whole system. As a conclusion, it can be said that 18% of parsing time and 8% of parsing memory used by the module system is fair, considering that all the data parsed is required to be stored and organized in a single belief-base / plan-library.
**Atomic Intention Handling**

In this test one of the unit tests were used as an input. Similarly to the module system two scenarios were tested and their difference was analysed. The scenarios are: 1) belief-base / plan-library modification and queries were done using only the normal belief-base / plan-library and 2) the same were done, but with using the delta belief-base and delta plan-library. And again for each scenario 5 test cases were made on the same computer, and the average of the results was used for comparison.

**First Scenario – only normal belief-base / plan library**

<table>
<thead>
<tr>
<th>Time taken:</th>
<th>0.0994 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum memory used:</td>
<td>20444 kB</td>
</tr>
</tbody>
</table>

**Second Scenario –using the delta belief-base / delta plan-library**

<table>
<thead>
<tr>
<th>Time:</th>
<th>0.1008 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum memory used:</td>
<td>20497.6 kB</td>
</tr>
</tbody>
</table>

**Evaluation**

The time difference is 0.0014 s which is about 1.5% of the time taken to completely perform all the actions; and the memory difference is 53.6 kB which is about 0.25% of the all the memory used by the whole system. Hence, it can clearly be seen that hardly any difference at all, therefore the atomic intentions are handled efficiently. These results were expected; since duplicates are rarely made, only the structurization of data was changed.

**Unit Tests**

To ensure that the implementation was indeed done according to the specification, an automated unit test was created for each of the subunit of the newly added features. As a conclusion unit tests were made to test the followings: import of modules, delta belief-base, delta plan-library, delta goal addition events. These unit tests include ‘control tests’ as well to be able to filter out false positives.

The unit tests helped to identify when an action did not have the desired results or it had undesired effects; and these detected errors were fixed. As these are automated tests, they also helped to detect undesired side-effects of refactoring.

**Quality Assurance**

To ensure maximum accuracy, whenever a task was considered finished, further manual, but comprehensive tests were made; which did not simply test a single unit of the system, but they tested the whole system, all the units together with their full complexity.
LIMIATATIONS AND FUTURE WORK

This section briefly describes the known limitations of the implemented tasks; and it also lists further extension ideas which could be future steps.

Limitations, Further Improvements

Custom Sleep Action (‘idle’)

In Jason it is possible to define a custom action that the agent should do when there are no events to process and no actions to execute. It is done by creating a plan which triggers the goal achievement event of the goal ‘idle’. Since there is no possibility of concurrent execution of modules yet (see later), this plan has to be defined in the scope of the base agent.

Shorter Unique Module Identifier

Because many operations done on the beliefs in the belief-base and on the plans in the plan-library use some kind of string comparison based on the functor of the predicate, the created long prefixed functors can slow down these operations. Therefore it would be better to pair (by the aid of a HashMap for example) the unique module identifiers created from the long absolute paths / URLs to short identifiers which however remain unique (e.g.: increasing integers) and then use these; as only their uniqueness is their required property.

Other Extensions

This section briefly introduces the reader to other possible extensions of Jason, which could make significant improvements and give great advantage to the agent-programmer.

Concurrent Execution of Modules

The created module system does not yet support the concurrent execution of modules or the multiple reactions to the same event. There is a simple workaround to this problem however:

A solo plan (for example in the scope of the base agent) is created which responds to the event. Then this solo plan creates a new goal achievement event as new focus for each of the modules. For example:

```goals
+!goal <- !!moduleA::goal; !!moduleB::goal; !!moduleC::goal.
```

However, this solution is inconvenient and it requires knowing in advance all the modules which needs to respond to the event. And therefore it would better if these would be supported at the level of the language.
Belief-Base with Datalog

Currently the Belief-Base language of Jason is Prolog, in which language the order of addition of new beliefs has an explicit effect on execution. This is undesired, because it could cause issues if this order is changed – which could be because it is unavoidable or just simply because of structurization reasons.

Further, Prolog might have issues handling large number of beliefs, which is likely to be the case during the development / execution of complex systems.

Moreover, in Prolog it is possible to describe a non-terminating algorithm, which is absolutely undesirable in the case of a belief-base language, because if such a non-terminating context check would happen, it could cause the halt of the agent or the entire system.

Even though it has already been started during the project, it has found to be better rescheduled, so it could be connected to the task of Automated Belief Revision (described next).

Automated Belief Revision

During the usage and handling of complex multi-agent systems, there is a definite need for belief revision within an agent, and it is clear if this process could be automated, it would provide a large amount of support for the programmer. For example it should automatically handle the inconsistency when both \( \neg p \) and \( p \) become mutually present in the belief base.

The created automated belief revision process should be practical by doing an efficient belief revision, and it should also be rational in the meaning of satisfying some predefined key objectives of this automated belief revision.
DISCUSSION

This section provides a brief evolution of the project and development process, it lists the lessons learnt from the positive outcomes and it also analyses the situations when something went wrong.

Lessons Learnt

Even though it was not the first large individual project I have ever worked on, I still have learnt a lot from it. This learnt knowledge includes project management and scheduling skills, communication skill and inevitable technical skills.

Project Management and Scheduling

Because it was a large project it clearly required consistent management and planning. During the project I was able to apply the techniques learnt from my courses, and also the experience that I have gained during my previous projects. Applying this knowledge has given me valuable experience, because I was able to test them myself, and see the actual results of these techniques used. This way I was able to learn which methodologies work the best in a project like this one, and which ones are those which does not work that well. I also learnt how very important scheduling is.

Communication

I have also learnt communication skills from the project, as I was consulting my supervisor, Dr. Brian Logan, and Dr. Neil Madden on a weekly basis in form of a meeting, and I had conversion with them about different topics via e-mail as well. Our discussions included my progress report and the next step to be performed, arguments about specification decisions, and the assessment of some design ideas.

Because I did not use Jason before the project, I had to rely on the multi-agent development experiences of Dr. Brian Logan and Dr. Neil Madden. Hence to be able to specify useful requirements of Jason+, it was absolutely required discuss these with them; for example to be able to determine what effects of an action are desired and what or not. However, in most cases, once I have gained the specific requirements I was able to make the implementation decisions on my own.

Technical Knowledge

Because I did not use an agent programming language or I have never developed a multi agent system before the project, I had to learn using these, before I was actually able to understand what is actually required to be done. I have learnt the concepts of agent-programming tools, and specifically I have learnt the syntax and features of Jason in details.

Further, I have gained invaluable programming experiences in Java, I have learnt how to handle large codebases like Jason’s, how to use Apache’s Ant to handle builds, and how to use most efficiently the features provided by the Eclipse IDE.
What could have been done differently

Schedule

In the first period, I did not manage the project appropriately. I did not keep myself to the schedule as I tried to use an agile technique. However, once I have realized that it caused delays, I have forced myself to keep the tight schedule created by me by prioritizing tasks. This way I was able to keep the agile technique I used without introducing further delays. Even though I was able to finish the project in time, if I was able to restart the project, I would first spend some time to be able to create a close estimated schedule leaving some gaps for modification, and I would ensure that the schedule created is met.

Specification

During the project, it happened that I did not understand clearly the specification (even though I thought I do) and I have implemented something else. Later when I have realized that what I implemented does not match the specification, I had to spend additional time re-implement that part. In the future – as I did from that point – I would make sure that the specification I understood is truly the right interpretation.

Acknowledgements

Finally, I would like to thank my supervisor Dr. Brian Logan for his supportive guidance, comprehensive advices and great understanding; and I would also like to thank Dr. Neil Madden for his contribution given during the weekly meetings.

SUMMARY

Multi-agent systems are becoming more and more popular – because of their great usability in some areas, and hence, agent-oriented programming languages are becoming more and more richer as well; making the task of the programmer harder to organize the produced code, to make it well structured, or even to generate a consistent system, especially in the development of complex multi-agent systems.

The project addresses this issue, by improving / extending a rather popular and useful agent-oriented platform called Jason, which is based on the BDI-based language proposal, AgentSpeak(L); by for example providing methods for modularization or for proper execution of atomic units.


Tree of Development Directories

The following is a list of the important directories and their description.

```plaintext
+----project
|   +----.externalToolBuilders
|   +----.settings
|   +----bin
|   |   +----jason
|   |   |   \----test
|   |   +----doc
|   |   |   +----AS2JavaParser.html
|   |   |   +----faq.html
|   |   +----MAS2JavaParser.html
|   |   \----api
|   +----examples
|   |   +----auction
|   |   +----directory
|   |   |   \----modules
|   |   |   \----src
|   |   |   |   +----asl
|   |   |   |   +----images
|   |   |   |   +----jason
|   |   |   |   +----jeditPlugin
|   |   |   |   +----templates
|   |   |   |   \----test
|   |   |   \----xml
|   +----_eclipseplugin
|   |   +----.settings
|   |   +----bin
|   |   |   \----net
|   |   \----src
|   |   |   +----icons
|   |   |   +----lib
|   |   |   +----modes
|   |   |   \----net
|   |   +----_libs
|   |   |   \----jedit
|   |   \----_web
+----_releases
   \----Jason+-1.3.2-dir

\----_software
   +----bin
   |   +----jason.bat
   |   +----jason.sh
   |   \----jedit
   |   +----doc
```
### Abbreviations Used in the Dissertation and Source Code

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annot</td>
<td>Annotation</td>
</tr>
<tr>
<td>App</td>
<td>Applicable</td>
</tr>
<tr>
<td>AS</td>
<td>AgentSpeak</td>
</tr>
<tr>
<td>ASL</td>
<td>AgentSpeak Language</td>
</tr>
<tr>
<td>BB</td>
<td>Belief-Base</td>
</tr>
<tr>
<td>BDI</td>
<td>Beliefs, Desires, Intentions</td>
</tr>
<tr>
<td>Bel</td>
<td>Belief</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus–Naur Form</td>
</tr>
<tr>
<td>BRF</td>
<td>Belief Revision Function</td>
</tr>
<tr>
<td>DBB</td>
<td>Delta Belief-Base</td>
</tr>
<tr>
<td>Dir</td>
<td>Directory</td>
</tr>
<tr>
<td>Doc</td>
<td>Documentation</td>
</tr>
<tr>
<td>DPL</td>
<td>Delta Plan-Library</td>
</tr>
<tr>
<td>FAQ</td>
<td>Frequently Asked Questions</td>
</tr>
<tr>
<td>IM</td>
<td>Intended Means</td>
</tr>
<tr>
<td>Int</td>
<td>Intention</td>
</tr>
<tr>
<td>Lib</td>
<td>Library</td>
</tr>
<tr>
<td>Lit</td>
<td>Literal</td>
</tr>
<tr>
<td>LogExpr</td>
<td>Logical Expression</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi Agent System</td>
</tr>
<tr>
<td>NF</td>
<td>New Focus</td>
</tr>
<tr>
<td>Pen</td>
<td>Pending</td>
</tr>
<tr>
<td>PL</td>
<td>Plan-Library</td>
</tr>
<tr>
<td>Pred</td>
<td>Predicate</td>
</tr>
<tr>
<td>PredInd</td>
<td>Predicate Indicator</td>
</tr>
<tr>
<td>RC</td>
<td>Reasoning Cycle</td>
</tr>
<tr>
<td>RegExp</td>
<td>Regular Expression</td>
</tr>
<tr>
<td>Sel</td>
<td>Selected</td>
</tr>
<tr>
<td>Src</td>
<td>Source</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>Var</td>
<td>Variable</td>
</tr>
</tbody>
</table>

### User Manual – FAQ

The followings are the relevant sections of the User Manual / FAQ of Jason⁺.

#### How can I import a module?

You can import a module by using the import directive along with an arbitrary local name and with the path of the imported module.

For example:

`{import phone="phoneSystem.aslm"}`

After you have imported a module you can use access it by using the arbitrary local name given along with the scoping operator (`::`). For example: `phone::makeCall(bob)`.

#### Why was I given a “module has no such exported predicate” error?

It means that you have tried to access a hidden predicate of a module.
Before you are accessing predicates of a module, first you must export them by using the export directive.

For example: `{export speakTo/1, activated}` exports the one-parametered `speakTo` predicate and the zero-parametered `activated` predicate of a module.

**Can I import modules dynamically during runtime?**

Yes you can, by using the `.import` internal action.

You can either specify only the imported modules path, for example:

`.import("phoneSystem.aslm")`

Or you can specify the optional local name of the module displayed in the Mind Inspector, for example:

`.import(phone, "phoneSystem.aslm")`

This internal action is handled just like any other internal action, so you can use it as part of any plan.

**How does Jason+ support atomic plans?**

A plan is considered atomic, if its label has the `atomic` annotation.

For example:

```prolog
@bookToShelf[atomic]
+!putBookToShelf(B, Agent) : not(bookOnShelf(B)) <- +bookOnShelf(B);
   .send(Agent, tell, bookOnShelf(B)).
```

Jason+ fully supports atomic plans, meaning that a comprehensive consistency is ensured. Hence, no other plans are selected to execute until the execution of an atomic plan is finished.

Further, it is also ensured that after attempting to execute an atomic plan, the state of the agent is either going to be in the exact same state as before (in case of a failure) or in the state resulted by the successful execution of all the actions of the plan.

Thus in the previous example, the belief `bookOnShelf(B)` is only added to the belief base permanently if the `.send` was successfully executed.

**Examples**

For an example of the static module system please, see the project in the following directory:

`project/examples/modules`

For an example of the dynamic module imports and atomic intention application, see the project in the following directory:

`project/examples/directory`