INF/SCR-09-67

Belief/Goal Sharing BDI Modules

Michal Cap

May 17, 2010

Abstract

This thesis proposes a modularisation framework for BDI-based agent programming languages. In this framework, BDI modules are seen as encapsulations of cognitive components that can be instantiated and manipulated in a similar fashion as objects in object orientation. An agent's mental state is formed dynamically from the BDI modules by means of the module activation. The agent's active modules deliberate concurrently and interact by sharing their beliefs and goals. Each module can specify an interface that determines which beliefs and goals will be shared with the other modules of the agent.

The framework is formally specified as an extension of the 2APL programming language. In particular, we extend its syntax, operational semantics and deliberation cycle. Likewise, the applicability of the newly introduced concepts is demonstrated using 2APL example programs.

Acknowledgements

I would like to thank Mehdi Dastani and Maaike Harbers for their guidance, support and especially long passionate discussions, which provided me with a lot of useful feedback and motivation.

Contents

1 Introduction **3**
- 1.1 Background . . . 3
- 1.2 Problem Statement . . . 3
- 1.3 Methodology . . . 4
- 1.4 Structure of Thesis . . . 5

2 Background **6**
- 2.1 Agent-Oriented Programming . . . 6
- 2.2 BDI Languages . . . 6
- 2.3 Modular Programming in BDI Languages . . . 6
- 2.4 2APL: A Practical Agent Programming Language . . . 8

3 Belief/Goal Sharing Modules **9**
- 3.1 General Description . . . 9
- 3.2 Newly Introduced Features . . . 10
  - 3.2.1 Concurrency . . . 10
  - 3.2.2 Module Instantiation . . . 10
  - 3.2.3 Module Activation . . . 11
  - 3.2.4 Belief/Goal Sharing . . . 13
  - 3.2.5 Include Section . . . 14

4 Syntax **15**
- 4.1 2APL Multi-Agent System . . . 15
- 4.2 2APL Agent . . . 16
- 4.3 2APL Module . . . 16
- 4.4 Module Instantiation . . . 16
- 4.5 Module Activation . . . 18
- 4.6 Belief/Goal Sharing . . . 19
- 4.7 Include Section . . . 20

5 Semantics **22**
- 5.1 Multi-Agent System . . . 22
- 5.2 2APL Agent . . . 22
- 5.3 2APL Module . . . 22
- 5.4 Module Instantiation . . . 24
- 5.5 Module Activation . . . 25
- 5.6 Belief/Goal Sharing . . . 27

5.7 Communication and External Environments . . . 32
5.8 Planning Goal Rules . . . 33
5.9 Message and Event Handling . . . 34

6 Execution Strategy **36**
6.1 Deliberation Cycle . . . 36

7 Discussion **40**
7.1 Extending Object-Oriented Inheritance . . . 40
7.2 Execution Strategies . . . 42
7.2.1 Serial Module Execution . . . 42
7.2.2 Parallel Module Execution . . . 42
7.2.3 Interleaved Module Execution . . . 43
7.3 Updating Beliefs and Golals . . . 44
7.3.1 Updating the Belief Base . . . 44
7.3.2 Dropping Goals . . . 47
7.4 Planning goal rule . . . 47
7.5 Communication and Interaction with External Environments . . . 48

8 Demonstration **50**
8.1 Bomb Disposal . . . 50

9 Conclusion **56**
9.1 Future Work . . . 56

Bibliography **58**

A Example Programs **60**

Chapter 1

Introduction

1.1 Background

Agent oriented programming is a programming paradigm that promotes a societal view of computation, where solutions are achieved by cooperation of autonomous entities – agents [17]. Several specialised languages emerged to facilitate the development of intelligent agents, based on different psychological and philosophical theories. This thesis focuses on a family of languages based on the Belief-Desire-Intention (BDI) theory [2] [15], the 2APL language in particular [6]. These languages use mentalistic notions such as beliefs, goals and plans to implement an agent's behaviour.

As with other software paradigms, the ability to decompose agent programs to separate, to some extent independent modules, is crucial for the development of any complex software system. It is widely believed that design principles such as loose coupling, information hiding, encapsulation or code reuse contribute highly to the software maintainability and the effectivity of development. This is typically achieved by dividing the program logic into separate modules, each of them with a well defined public interface or by employing one of the object oriented programming languages, which usually already contain constructs for such design patterns.

However, modularisation is something one cannot count on when using agent programming languages. Some of the platforms (e.g. JACK [5], Jadex [3]) have introduced notions that facilitate program decomposition and code re-use and several theoretical papers have been published on this topic, but a widely accepted concept of modularisation for BDI system is still missing. This thesis aims to contribute to this discussion.

1.2 Problem Statement

Although agent programming languages inherently support program decomposition by means of distributing different tasks to different agents, this form of decomposition is not always sufficient. Intensive inter-agent communication may introduce significant overhead and tasks that require global reasoning might become difficult to tackle. Having complex agents is in some situations unavoidable.

To facilitate the development of complex agents, a programmer must be able to use standard software engineering techniques such us separation of concerns, encapsulation or code reuse.

Virtually any traditional imperative language allows programmer to structure programs into smaller manageable units. The concept of module as used in imperative languages is well understood and almost universally used.

Unfortunately, there is no obvious way to transform the notion of module (or object in case of object oriented languages) to agent programming languages. Current mainstream languages are based on variables (fields) to store data and functions (methods) to process data. BDI-based agent programming languages are based on beliefs, goals, rules and plans. Imperative languages are deterministic and well suited for programs that run in a serial manner. Agent programming languages are in contrast generally non-deterministic and by their nature interpreted in a parallel manner. All these differences account for a concept of a module that is different from what we use now in the mainstream software development.

The BDI-module should however exhibit similar properties as classical modules, in particular:

Separation of concerns A module should allow programmer to focus on one concern of a program a time.

Encapsulation A module should be able to offer a well-defined interface through which other modules can interact with it. The actual implementation of the module logics should be invisible to the outside world. The programmer should be even able to later change it as long as the interface stays stays intact and produces identical behaviour.

Reusability A module should be able to encapsulate some general functionality, which can be re-used in different parts of program or in a completely different systems.

We also believe that we should aim for some additional properties of the BDI-module system:

Agent-orientation A module should respect the specifics of agent-oriented paradigm and facilitate use of the agent-oriented concepts.

Familiarity A concept of BDI-module should be easy to grasp for programmers familiar with the imperative modules and object orientation.

1.3 Methodology

We are going to investigate the relevant research that has been done on the topic of BDI-oriented modules. Then we will propose a modularisation system suitable for BDI-based languages. The concept will be rooted in the modularisation as used in traditional imperative languages and in object orientation.

We will investigate options and propose a concept of a BDI-module that will aim to satisfy the properties stated above. We have chosen the 2APL programming language to be able to formalise the concept of the BDI-module as an extension of an existing BDI language. We will introduce extensions to the 2APL syntax, semantics and deliberation cycle. In addition, we will offer examples of multi-agent programs that may take advantage of the new features of the language.

The formal model will be incorporated into the current 2APL interpreter in order to investigate the technical feasibility of the proposed solution.

1.4 Structure of Thesis

This thesis is structured as follows: The following chapter discusses the background of our work. The third chapter introduces the concept of belief/goal sharing modules. The fourth chapter presents extensions to the 2APL syntax. The fifth chapter extends the operational semantics of 2APL language. The sixth chapter introduces changes to the 2APL deliberation cycle. The seventh chapter discusses decisions made during the design of the framework. The eighth chapter demonstrates the applicability of the proposed framework and finally, the ninth chapter presents conclusions and suggest directions of future work.

Chapter 2

Background

2.1 Agent-Oriented Programming

Agent-oriented programming is a software engineering paradigm that is claimed to offer high level programming constructs for the development of complex software systems that may be infeasible to realize using traditional techniques [17].

Agent-oriented programs are typically build up as a collection of autonomous self-interested agents who through their interaction arrive to a problem solution. While such systems can be implemented in virtually any programming language, several specialised agent programming languages believed to facilitate the process have been proposed. One representative of such approach is the family of the BDI-based agent programming languages.

2.2 BDI Languages

BDI agent programming languages are stemming from Michael Bratman's theory of human practical reasoning [2], commonly referred to as belief-desire-intention theory. Bratman is in his theory relying on concepts from folk psychology, i.e. everyday language concepts such as beliefs, desires or commitments, which humans seem to use when talking and thinking about how they reason about the mental states of other people.

BDI programming languages employ those notions as constructs allowing specification of the agent's behaviour. A typical BDI language consists of the following mental components: 1) a belief base storing the agent's beliefs about the current state of the world, 2) a goal base representing the motivational stance of the agent, 3) an intention base containing the agent's future plans and 4) a rule base representing the library of pre-cooked plans an agent can use to pursue his goals and react to upcoming events.

To this day a number of BDI-based languages have been proposed. Some of them adhere closely to the theoretical models and apply logic programming techniques, e.g. Jason [1], 2APL [6] or GOAL [10]; others, e.g. JACK [4] or Jadex [14], add BDI-related constructs to some established imperative language (usually Java).

2.3 Modular Programming in BDI Languages

Modularisation is an essential design principle in software engineering. It encourages the programmer to break down the program into a number of independent components, each

focusing on one sub-problem. Such decomposition makes the development of complex software systems feasible as it allows programmer to concentrate on one concern at a time. Modularised software systems can be even developed by a team of people since each of them can work on his independent part without worrying about the interferences with the work of the others. Moreover, as modules are often designed to solve some general problem, a module that has been once implemented can be reused to solve the same problem in other software projects.

It appears that virtually all mainstream programming languages support either implicitly or explicitly the idea of program modularisation by offering constructs to divide the program code into smaller independent units. The majority of them also allows a programmer to define which elements of each independent part will be accessible from outside the module. In C programming language, for example, a program's source code can be divided in a number of source files that can be compiled individually into so-called object code and finally put together using a linker tool. Each object is typically associated with a header file, which lists declarations of available subroutines and variables to be used by other parts of the program [11].

Object oriented programming, which has become the predominant programming paradigm in the course of the last two decades, can be seen as a successor of modular programming. It further stresses the importance of loose coupling, encapsulation and data abstraction, and provides a complete object-centred framework that facilitates the design of the software that adheres to those principles [16].

It can be argued that agent oriented programming goes beyond this and promotes the ideas related to modularity even more intensively. Indeed, the agent-oriented programs consist of a number of agents, which are by their nature almost perfectly encapsulated (it is not permitted to examine or alter their internal state from other agents) and loosely coupled (agents communicate using an agent communication language) entities. However, this holds only for the multi-agent perspective.

In some domains (e.g. when an agent implements a human-like virtual character), individual agents are becoming increasingly complex and themselves candidates for the decomposition. Moreover, it is often the case that different agents share a substantial part of their specification. Therefore, it is becoming clearer that modularisation is a principle that can be fruitfully used also on a single-agent level. There are several proposals supporting modularisation in BDI-based languages:

For example, in the work of Hindriks [9], modules for the GOAL programming language are viewed as "policy-based intentions". In this framework a programmer can specify the context condition determining when the module becomes active.

Madden and Logan [**?**] build on their practical experience with Jason and introduce revisions to the language that are claimed to facilitate the development of modular programs. They propose a mechanism for the construction of agents from functionally encapsulated components and extend the belief language with an XML-like prefixing mechanism in order to solve the identifier collision problem.

A rather different approach is employed by the modular BDI architecture [13], which proposes a framework that allows to abstract from the concrete knowledge representation technique used to implement each of the BDI components and offers the possibility to use the best suited technology for each of them. E.g. one can use a Bayesian network module as a belief base and a Prolog module as a goal base of an agent.

In JACK, the agent can be specified as a composition of so called capabilities [5] that are seen as functional clusters of BDI components. The capability concept has been further

extended in [3] and applied as the modularisation mechanism within the Jadex platform.

Finally, [18] proposes the concept of goal-oriented modularity for the 3APL language, which is based on the idea that modules encapsulate the information on how to achieve a goal, or a set of (related) goals.

You may notice that in contrast to imperative languages, there is still no widely accepted notion of the modularisation for BDI languages. Some of the platforms provide constructs that allow for decomposition of agent specification, but the approaches they take differ. Other platforms support modularisation in a very limited way or not at all.

2.4 2APL: A Practical Agent Programming Language

Although the concept of belief/goal sharing modules presented in this thesis is general and in principle applicable to any BDI-based language (given that the language supports declarative goals), we decided to extend an existing language to provide grounds for our experimentation. We chose for the 2APL agent programming language [6]. This language combines declarative programming style to implement an agent's beliefs, goals and belief updates, with imperative programming style to specify an agent's plans. Besides that, the 2APL agent contains a set of practical reasoning rules, which can be of three different kinds: 1) PG-rules specifying plans to achieve agent's goals, 2) PC-rules specifying plans for abstract actions and plans for handling external events, 3) PR-rules specifying repair plans handling plan failures.

In [7], the 2APL language has been extended with a notion of a module, which is seen as an encapsulation of 2APL cognitive components. Thus, the module consists of beliefs, goals, belief update specifications, rules and plans. The modules are assumed to be dynamic, in the sense that they can be instantiated from a module specification and their execution can be controlled by other modules during the run-time. The proposal introduces several actions that can be used to manipulate module instances. Firstly, the `create` and `release` actions are used to instantiate a module specification and to discard an existing module instance. Secondly, beliefs of a module instance can be updated using the `updateBB` action and goals of a module instance can be adopted or dropped using the respective goal action. Thirdly, beliefs and goals of a module instance can be tested using the test action. Finally, the execution control can be handed over to a module instance by means of the `execute` action.

Although the above mentioned module proposal offers constructs for modular programming in 2APL, we believe it needs to be further extended in order to provide program decomposition support comparable with that of mainstream imperative languages. We design our framework as an extension of this existing proposal. The following chapter will elaborate on the relation between the existing modular framework as specified by Dastani et al. [7] and the extended modular framework, which will be introduced in this thesis.

Chapter 3

Belief/Goal Sharing Modules

This chapter presents the concept of belief/goal sharing modules suited for BDI-based agent programming languages that support the notion of declarative goals. We base our proposal on earlier work discussing modularity in the 2APL [7] [8] and we extend it in several directions. Section 3.2 elaborates on this.

3.1 General Description

This section explains the general concepts behind the modular system presented in this thesis.

In our framework, we define an agent as an execution of the deliberation cycle on the agent's mental state, which is modelled as a composition of module instances. The mental state of any agent contains at least one module instance – the agent's *main module instance*.

We consider a module instance as an encapsulation of cognitive components such as beliefs, goals, belief updates, rules and plans. If a module instance forms a part of an agent's mental state, the agent's deliberation cycle operates on it and the module instance is considered *active*. Conversely, a module instance can exist outside the mental state of any of the agents, in which case the module instance is considered *inactive*. As we will see later, a module instance can be activated or deactivated at any time during the execution of the multi-agent system.

The `create(`s`,`m`)` action can be used to create a new inactive module instance m from a module specification s. The resulting module instance represents a static collection of data, which can be inspected and altered by any agent that knows the module instance identifier m. In particular, the beliefs and goals of an inactive module instance m can be queried by means of the belief test m`.B(`φ`)` and the goal test m`.G(`κ`)`. Further, its belief base can be updated by means of the m`.updateBB(`φ`)` action, and the goal base can be manipulated by means of the adopt goal actions m`.adopt[a/z](`γ`)` and the drop goal actions m`.drop[sub/super]goal(`κ`)`.[1] Finally, an inactive module instance m can be discarded using the `release(`m`)` action.

Should any of the agents decide to activate an inactive module instance m, it can do so by means of the `include(`m`)` and `seclude(`m`)` actions, which incorporate the module instance into the agent's mental state. Consequently, the agent's deliberation cycle starts operating on the internals of the module instance, resulting in the rules of the module instance being applied and the plans getting executed. Since the beliefs and goals of an active module instance change

[1] We refer to the former module proposal [7] for a more detailed description of these actions.

dynamically as a consequence of the deliberation process, it is unsafe to access the internals of a module instance from other module instances directly. Therefore, the actions like m.B(φ), m.G(κ), m.adopt[a/z](γ) and m.drop[sub/super]goal(κ) cannot be executed on an active module instance. Instead, the interaction between the active module instances of an agent is realized by sharing their beliefs and goals. Each module instance specifies an interface that determines which beliefs and goals are to be shared with the other module instances. As in some situations it is undesirable to let all the agent's module instances share beliefs and goals with each other, we provide methods to divide module instances into separate clusters, each forming an independent belief/goal sharing scope.

An active module instance m can be deactivated, i.e. removed from an agent's mental state, either explicitly by means of the exclude(m) action or implicitly when a predefined stopping condition has been satisfied. The life cycle of a module instance is depicted in the Figure 3.1. For more explanation on the Include: transition please see the Section 3.2.5.

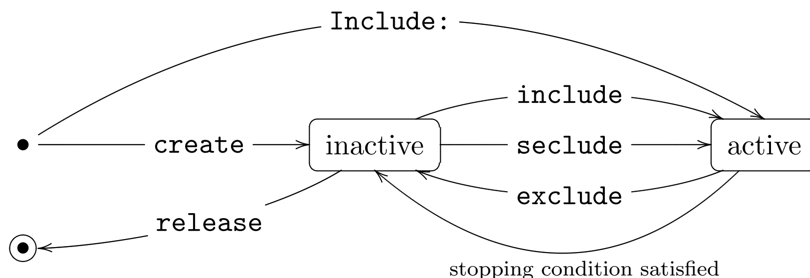


Figure 3.1: Lifecycle of a Module Instance

3.2 Newly Introduced Features

3.2.1 Concurrency

The original module proposal [8] allows only one active module instance per agent at a time. We believe that this is overly restrictive since it is not unusual for an agent to deliberate on several different issues in parallel. Consider for example a bomb disposing agent that consists of two module instances: 1) a module instance sensing the bombs in the environment and 2) a module instance handling the disposal of a bomb to the bomb trap. It seems reasonable to require both module instances to be active in parallel, so that the agent can sense the bombs also on his way towards the bomb trap. Therefore, in our proposal an agent can consist of an arbitrary number of active module instances, all of them deliberating concurrently.

3.2.2 Module Instantiation

Previously, a programmer had to determine in design time which module specifications will be instantiated and assign a unique name to each instance. This can be regarded restrictive, as in some situations it is practical to postpone the decision on how many of which module specifications to instantiate to run-time, e.g. based on the current beliefs and goals of the agent. Consider for example an auction bidder agent whose belief base contains a set of items

he plans to bid on. Further, suppose that the functionality needed to bid on one item is available in a separate `bidding` module specification. In such setting a programmer needs to create a number of instances of the `bidding` module specification for each item the agent wants to bid on. The natural way to do this would be:

```
PG-rules:
  <- item(Item) | {create(bidder, Bidder); ... }
```

However, since in the original module proposal variables cannot be used as parameters of the `create` action, such an approach would be infeasible. Therefore, we allow and encourage programmers to use a variable at any place where a module instance identifier is expected. This approach offers also some additional flexibility, e.g. the reference to a module instance may be passed to an abstract action as illustrated in the following example:

```
PG-rules:
  <- item(Item) | {
    create(bidder, Bidder);
    Bidder.updateBB(item(Item));
    startBidding(Bidder)
  }
PC-rules:
  startBidding(Module) <- true | {Module.B(item(Item)); ... }
```

Moreover, the former proposal regarded a module instance as being accessible only from its parent i.e. from the module instance that has instantiated it. We employ a more general approach and let any agent manipulate any inactive module instance residing in the multi-agent system. However, in practice the rule of general accessibility is limited by the fact that an agent needs to first know the module instance identifier in order to be able to refer to the module instance. The creator of the module instance is initially the only one who is aware of the concrete identifier of the newly instantiated module. Therefore, the creator is capable of delegating the access right to the new module instance by disclosing the module instance identifier e.g. through communication.

Consequently, we have decided to change the way the module instances are assigned their names. Formerly, the module instance identifier was composed of a basic module name and the identifier of the parent of the module instance. In our proposal, we do not assume any structure in a module instance name. In the majority of cases a module instance is assigned a randomly generated system name, which a programmer obtains after the module instantiation.

You may notice that this closely resembles the process of object instantiation as used in the majority of object oriented programming languages. Take for example the following Java statement: `Color color = new Color(0,0,0);`. In fact, the Java operator `new` instantiates the class `Color` using given parameters on the heap and returns a system-determined pointer to the object, which will be stored in the variable `color`. The correspondence between the object instantiation and module instantiation as used in our system is intentional as it helps to satisfy the *familiarity* property and thus makes the modular system more comprehensive for programmers acquainted with the object orientation.

3.2.3 Module Activation

Originally, the only way to activate a module instance was by means of the `execute` action. In our proposal we provide two distinct methods one can use to activate a module instance:

1. *module inclusion* and 2. *module seclusion.* Both methods incorporate a given module instance into the agent's mental state as the consequence of which, the internals of the module instance will be processed during the agent's deliberation cycle. The two methods differ only with respect to belief/goal sharing, on which we will further elaborate in the following section.

An agent's mental state is modelled as a tree of module instances. The root node of an agent's module instance tree is always the agent's main module instance. When an inactive module instance is activated, it becomes a child node of the module instance that activated it. If an inactive module instance is activated by means of the module inclusion (e.g. by means of an `include` action), it will become a child node connected to its parent by an inclusion link. If an inactive module instance is activated by means of the module seclusion (i.e. by means of a `seclude` action), it will become a child node connected to its parent by a seclusion link. Consider the example depicted in the Figure 3.2 as an illustration of the process of module activation.

The example shows a worker agent who activates an instance of `searching` module, which encapsulates a functionality needed to search the agent's environment, and an instance of `colleague` module, which is used as a profile of one of his fellow worker agents. In the first step, the `worker` module instance instantiates the modules by means of the create action. We can see that after this the modules reside inactively in the multi-agent system. In the second step, the `worker` module activates the module instances: The `searching` module instance is activated by means of the module inclusion, whereas the `colleague` module instance is activated by means of the module seclusion. We can see that the two module instances became children of the main module instance of the worker agent and therefore part of the agent's mental state.

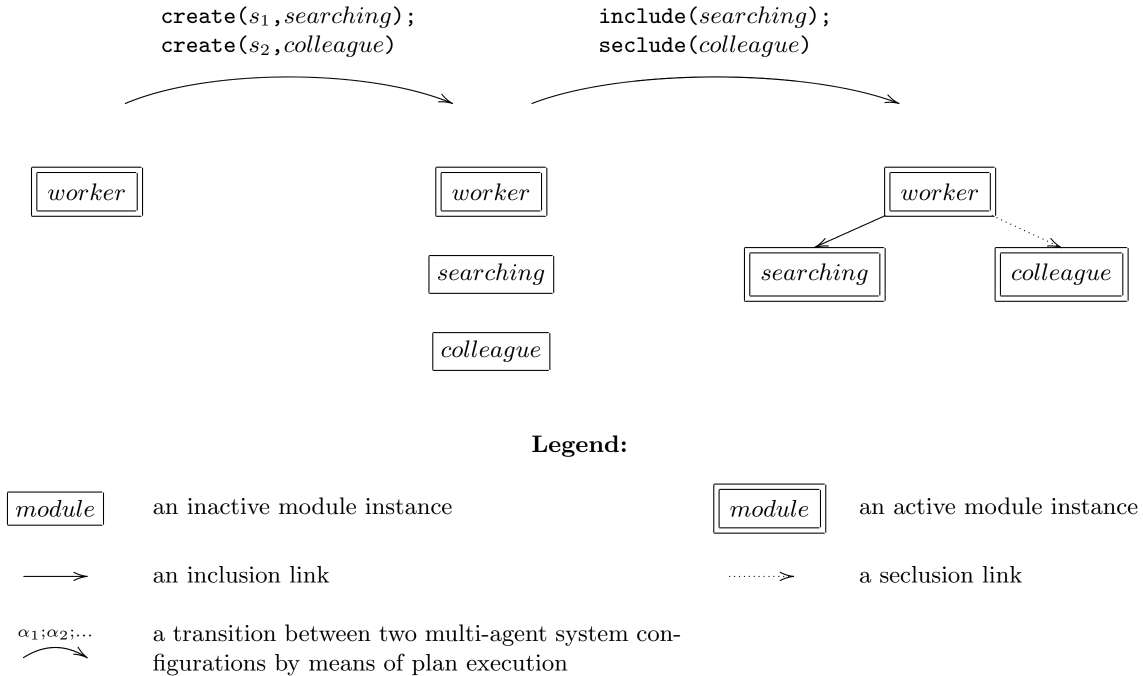


Figure 3.2: Module Activation Example

3.2.4 Belief/Goal Sharing

Since in our framework all of the agent's module instances deliberate concurrently, we recognise that a robust mechanism allowing them to interact and exchange data should be provided. For this purpose we introduce a notion of belief/goal sharing based on the idea of clustering module instances into so-called *sharing scopes*, within which the module instances share beliefs and goals constrained by their predefined *interfaces*.

It should be noted that the sharing scope is an implicit concept. The clustering of module instances into sharing scopes is derived from the structure of the agent's module instance tree. Each agent contains initially one sharing scope, which consist only of the agent's main module instance. Module instances are added to an existing sharing scope when they are activated by means of module inclusion. Analogically, new sharing scopes emerge as a consequence of activating a module instance by means of module seclusion.

In order to establish the mechanism as *agent-oriented*, we attributed some of the notions of agency to the concept of the sharing scope. Firstly, the sharing scope is assumed to have a global belief base that combines interfaced beliefs from all module instances in the sharing scope. Analogously, the sharing scope is assumed to have a global goal base that combines interfaced goals from all module instances in the sharing scope. Finally, the interrelations between the global beliefs and global goals are governed by the rationality principle i.e. it is not possible to desire a goal if it is already believed to be achieved. You may notice that if an agent's mental state is composed of just one sharing scope, the concept of the agent's mental state and the concept of the sharing scope become identical.

In fact, the main reason why we discriminate between an agent's mental state and a sharing scope is to allow several sharing scopes having mutually inconsistent beliefs and goals within one agent. There is a number of applications that could take advantage of this feature. One example would be user profiling: Imagine an agent who keeps the profile of another agent. Since the profile's beliefs and goals might be contradictory to the agent's general beliefs and goals, it would be practical to implement the profile in an isolated sharing scope to avoid unwanted interactions. Another example would be an agent who reasons about possible future world states. Such agent might use an isolated sharing scope to keep each alternative world state it can foresee.

To satisfy the *encapsulation* property of the proposed modular system, we introduce a notion of module interface. Each module instance can specify a module interface which is defined as a set of module interface entries, each of them being an atomic formula. The module interface serves several functions:

Ontological Function A module interface specifies a *module interface language* that is to be used to interact with the module. A module interface language allows to build formulas composed only from the atomic formulas that are unifiable with one of the interface entries, and logical connectives `and`, `or` and `not`. All beliefs and goals interfaced by the module instance will be expressed in the module interface language.

Export Declaration Function The module interface is used to define which of the basic beliefs and goals of the module instance will be interfaced and therefore assumed to be global beliefs and goals of its sharing scope. In fact, the module instance will interface only those beliefs that can be expressed in the module interface language. Similarly, only the goals entailed by the goal base of the module instance that can be expressed in terms of the module interface language will be interfaced.

Import Declaration Function The module interface can be used to define which of the global beliefs and goals will be accessible from the module instance. All the global beliefs and global goals of the sharing scope of the module instance that can be expressed in terms of the module interface language can be queried from inside the module instance.

Visibility Restriction Function The module interface may be used to limit the visibility of the internals of a module instance. Any belief or goal that cannot be expressed in terms of the module interface language stays private and cannot be accessed from outside the module instance.

The composition of module instances that form an agent's mental state can be specified either at design-time using the `Include:` construct or dynamically at run-time using the `include`, `seclude` and `exclude` actions. See the following section for an explanation on the former option. The latter approach may be used e.g. for the implementation of roles, which can be enacted and de-acted by an agent dynamically, or to implement self-optimizing agents able to evolve their internal structure. We provide three actions that dynamically manipulate the module structure of the agent's mental state. Firstly, a module instance can be added to an existing sharing scope by means of the `include(`m`)` action. The `include` action activates the given inactive module instance m and incorporates it into the sharing scope of the module instance that executed the action. Secondly, a new sharing scope can be created using the `seclude(`m`)` action. The `seclude` action creates a new secluded sharing scope, activates the given inactive module instance m and adds the module instance into the new sharing scope. Finally, a module instance can be removed from a sharing scope by means of the `exclude(`m`)` action, which deactivates the given active module instance m and removes it from the sharing scope it currently belongs to. In case the sharing scope contains no more module instances it ceases to exist.

3.2.5 Include Section

The original 2APL language [6] provides the `Include:`*filename* construct that can be used to include source code from an external file. Since the inclusion is realised merely as a textual merger of the given files, its use for program logic encapsulation is limited. We redefine the existing construct in line with the proposed modular framework. The `Include:` section is an optional part of a module specification, which is consulted each time the module specification gets instantiated. All the module specifications listed in the `Include:` section are instantiated and included into the sharing scope of the newly instantiated module. The construct can be therefore used to specify what module instances will form an agent's mental state statically at the design-time. Note that the statement `Include:` s is functionally equal to the following initial plan: `Plans: create(`s`,`m`);include(`m`)`.

Chapter 4

Syntax

This chapter presents the modular extensions to the 2APL syntax. The complete EBNF syntax of the language for the specification of a multi-agent system and an individual module is presented in Figure 4.1 and Figure 4.2 respectively. The changes we introduce are highlighted in black.

4.1 2APL Multi-Agent System

The initial state of a multi-agent systems can be specified in terms of agents, their main module specifications and a list of environments accessible to the agent. A concrete multi-agent system is described in the multi-agent system specification file whose syntax is presented in Figure 4.1.

Example 1
The following specification defines a multi-agent system that contains four agents. The first line prescribes that the initial mental state (i.e. main module) of agent `m` is instantiation of module specification `manager`. The second line prescribes that there are three agents named `w1`, `w2` and `w3` whose main module is an instantiation of module specification `worker`. Further, the worker agents can access the `blockworld` environment.

```
m : manager
w : worker 3 @blockworld
```

```
<MAS_Prog>         =  (<agentname> ":" <mainmodulespec> [<int>] [<environments>])+;
<agentname>        =  <ident>;
<mainmodulespec>   =  <ident>;
<environments>     =  "@" <ident> {"," <ident>};
```

Figure 4.1: The EBNF syntax of 2APL multi-agent system

4.2 2APL Agent

The notion of the 2APL agent corresponds to a deliberation cycle operating on a tree of module instances that represent the mental state of the agent.

4.3 2APL Module

A module instance encapsulates a part of an agent's mental state. It consist of cognitive components such as beliefs, goals, belief updates, rules and plans. Further, it also contains links to its children in the agent's module instance tree: Firstly, it keeps a list of included module instances i.e. module instances included in the sharing scope of the module instance. Secondly, it keeps a list of secluded module instances, i.e. module instances that reside in their own isolated sharing scope. The initial state of a module instance is specified in form of a *module specification*, whose syntax is presented in Figure 4.2.

Example 2
The following module specification defines a module instance that initially believes it is located at position `[0,0]` and desires to be at position `[5,5]`. It also contains one rule prescribing how can be such goal achieved.

```
Beliefs:
  at([0,0])
Goals:
  at([5,5])
PG-rules:
  at([X,Y]) <- true | @blockworld(goto(X,Y),_)
```

4.4 Module Instantiation

Our framework provides two methods one can use to create a new module instance: Firstly, a module can be instantiated from a module specification stored in a 2APL file by means of the `create` action. Secondly, all the module specifications listed in the `Include:` section will be automatically instantiated. Conversely, a module instance can be removed from the multi-agent system by means of the `release` action.

When a module specification is instantiated using the `create` action, the identifier of the new module instance can be either specified explicitly by a programmer or it can be automatically generated by the system. Usually, it is more practical to let the system generate a unique (typically random) identifier for the module instance. A variable can be provided in place of the module identifier parameter, which will get bound to the system-generated identifier. Then, a programmer will have obtained the module identifier, which can be used to refer to the module instance during a subsequent execution of the agent's plans.

After the instantiation, the module instance resides in a multi-agent system, not yet associated with any particular agent. In fact, any agent that knows the module instance

```
<APAPL>              =  { "Include:" <includes> | "Interface:" <ientries>
                     | "BeliefUpdates:" <beliefupdates> |  "Beliefs:" <beliefs>
                     | "Goals:" <goals> | "Plans:" <plans> | "PG-rules:" <pgrules>
                     | "PC-rules:" <pcrules> | "PR-rules:" <prrules> };
<includes>           =  <include>+;
<include>            =  <ident>;
<ientries>           =  <ientry> {"," <ientry>};
<ientry>             =  <atom>;
<beliefupdates>      =  <beliefupdate>+;
<beliefupdate>       =  "{" [<belquery>] "}" <beliefupdatename> "{" <literals> "}";
<beliefupdatename>   =  <upperatom>;
<beliefs>            =  <belief>+;
<belief>             =  <ground_atom>"."
                     |  <atom> ":-" <literals> ".";
<goals>              =  <goal> {"," <goal>};
<goal>               =  <ground_atom> {"and" <ground_atom>};
<baction>            =  "skip" |  <beliefupdatename> |  <sendaction> |  <externalaction> | <abstractaction>
                     |  <test> |  <adoptgoal> |  <dropgoal> | <createaction> | <releaseaction>
                     |  <includeaction> | <secludeaction> | <excludeaction> | <moduleaction>;
<plans>              =  <plan> {"," <plan>};
<plan>               =  <baction> | <sequenceplan> | <ifplan> | <whileplan> | <atomicplan> | <scopeplan>;
<createaction>       =  "create(" <iv> ["," <iv> "])";
<releaseaction>      =  "release(" <iv> ")";
<includeaction>      =  "include(" <iv> ["," <test> "])";
<secludeaction>      =  "seclude(" <iv> ["," <test> "])";
<excludeaction>      =  "exclude(" <iv> ")";
<moduleaction>       =  <iv> "." <maction>;
<maction>            =  <adoptgoal> | <dropgoal> | <updBB>;
<updBB>              =  "updateBB(" <literals> ")";
<sendaction>         =  "send(" <iv> "," <iv> "," <atom> ")"
                     |  "send(" <iv> "," <iv> "," <iv> "," <iv> "," <atom> ")";
<externalaction>     =  "@" <ident> "(" <atom> "," <var> ")";
<abstractaction>     =  <atom>;
<test>               =  ( [<iv> "."] "B(" <belquery> ")" ) | ( [<iv> "."] "G(" <goalquery> ")" )
                     |  <test> "&" <test>
<adoptgoal>          =  "adopta(" <goalvar> ")" | "adoptz(" <goalvar> ")";
<dropgoal>           =  "dropgoal(" <goalvar> ")"  |  "dropsubgoals(" <goalvar> ")"
                     |  "dropsupergoals(" <goalvar> ")";
<ifplan>             =  "if" <test> "then" <scopeplan> ["else" <scopeplan>];
<whileplan>          =  "while" <test> "do" <scopeplan>;
<atomicplan>         =  "[" <plan> "]";
<scopeplan>          =  "{" <plan> "}";
<pgrules>            =  <pgrule>+;
<pgrule>             =  [<goalquery>] "<-" <belquery> "|" <plan>;
<pcrules>            =  <pcrule>+;
<pcrule>             =  <atom> "<-" <belquery> "|" <plan>;
<prrules>            =  <prrule>+;
<prrule>             =  <planvar> "<-" <belquery> "|" <planvar>;
<goalvar>            =  <atom> {"and" <atom>};
<planvar>            =  <plan>  | <var> |  "if" <test> "then" <scopeplanvar> ["else" <scopelanvar>]
                     |  "while" <test> "do" <scopeplanvar> |  <planvar> ";" <planvar>;
<scopeplanvar>       =  "{" <planvar> "}";
<literals>           =  <literal> {"," <literal>};
<literal>            =  <atom> |  <infixatom>  |  "not" <atom> | "not" <infixatom>;
<belquery>           =  "true" |  <belquery> "and" <belquery> | <belquery> "or" <belquery>
                     | "(" <belquery ")" | <literal> ;
<goalquery>          =  "true" |  <goalquery> "and" <goalquery> | <goalquery> "or" <goalquery>
                     |  "(" <goalquery> ")" | <atom>;
<iv>                 =  <ident> | <var>;
```

Figure 4.2: The EBNF syntax of 2APL module specification

identifier may test and update its internals[1].

Example 3
Consider the following fragment of a plan. The first action of the plan creates a new module instance from the module specification `carrier`. The module instance will be assigned an identifier, for example `carrier#1c57d`, which will be unified with variable `MyCarrier`. This allows a programmer to update the belief base of the module instance, test it and finally discard it using the `release` action.

```
create(carrier, MyCarrier);
MyCarrier.updateBB(bomb([1,6]));
...
MyCarrier.B(done);
release(MyCarrier);
```

Module Create Action

The module create action of the form `"create(" [<moduleSpec> ","] <moduleId> ")"` creates a new module instance in the multi-agent system. The first optional parameter `<moduleSpec>` determines the module specification to be instantiated. The interpreter assumes that the module specification is stored in the file named identically as the module specification succeeded by the `.2apl` suffix. In the case the `<moduleSpec>` parameter is not provided, the system creates an empty module instance. The second `<moduleId>` parameter is either a variable or a constant. If `<moduleId>` is a constant or a bound variable, its value will be used as the identifier of the module instance. In the case that `<moduleId>` is an unbound variable, the interpreter will generate a unique module identifier and unifies it with the provided variable.

Module Release Action

The module release action of the form `"release(" <moduleId> ")"` removes the module instance identified by the `<moduleId>` from the multi-agent system. The `release` action can be performed only on an inactive module instance.

4.5 Module Activation

We provide two actions a programmer can use to activate a module instance. Firstly, the `include` action may be used to include the given module instance into the sharing scope of the calling module instance. Secondly, the `seclude` action may be used to seclude the given module instance by creating a new isolated sharing scope and adding the module instance into it.

[1] Note that only the creator of the module instance knows its identifier by default. Other agents may become aware of it only if it has been disclosed to them.

Include Module Action

The include module action of the form "include(" <moduleId> ["," <stoppingCond>]")" activates the module instance <moduleId> by adding it to a list of included modules of the module instance that executed the action. Consequently, the module instance is incorporated into the agent's mental state, forming a part of the sharing scope of the calling module instance. The module instance will be automatically deactivated as soon as its mental state satisfies the stopping condition, which can be set using the <stoppingCond> parameter. If the stopping condition is not provided, it is assumed to be falsum. Therefore we consider include(m) as a syntactic abbreviation for the action include(m, B(not true)). The include action fails if the module instance is already active or if the module specified by the <moduleId> does not exist.

Seclude Module Action

The seclude module action of the form "seclude(" <moduleId> " ["," <stoppingCond>]")" activates the module instance <moduleId> by adding it to a list of secluded modules of the module instance that executed the action. Consequently, the module instance is incorporated into the agent's mental state, forming a new isolated sharing scope for itself. The module instance will be automatically deactivated as soon as its mental state satisfies the stopping condition, which can be set using the <stoppingCond> parameter. If the stopping condition is not provided, it is assumed to be falsum. Therefore we consider seclude(m) as a syntactic abbreviation for the action seclude(m, B(not true)). The seclude action fails if the module instance is already active or if the module instance specified by the <moduleId> does not exist.

Exclude Module Action

The exclude module action of the form "exclude(" <moduleId> ")" deactivates the module instance <moduleId> by removing it from a list of included modules and a list of secluded modules[2] of the module instance that executed the action. Consequently, the module instance is removed from the agent's mental state and stays inactive in the multi-agent system. The action fails if the provided module instance does not exists or if it is inactive.

4.6 Belief/Goal Sharing

Interface Section

Each module specification may contain an interface section that declares which beliefs and goals will be shared with the other module instances in the sharing scope. The interface section is defined as a set of interface entries preceded by the Interface: label. An interface entry is formally a Prolog-like atomic formula that performs the function of a template matching concrete beliefs and goals. Therefore, it often contains variables. Examples of interface entries are expressions like at([X,Y]) or book(_).

[2] A module instance can be a member of only one of the two lists.

Example 4
Consider the following excerpt from a module specification file:

```
Interface:
  book(_),
  sold(_)

Beliefs:
  book(hamlet),
  book(1984)

Goals:
  sold(hamlet) and read(1984)
```

The interface section specifies that all beliefs derivable from the basic belief base of the module instance that are unifiable with the atomic formula `book(_)` will be interfaced. In our example, the module instance will interface beliefs `book(hamlet)` and `book(1984)`. If there will be any belief unifiable with `book(_)` interfaced by some other module instance in the sharing scope, the belief will be available for querying from within the module instance.

We can also see that the interface contains entry `sold(_)`. This entry will interface all subgoals derivable from the basic goal base of the module instance that are unifiable with `sold(_)`. In particular, the module will interface its goal `sold(hamlet)` because `sold(hamlet) and read(1984)` $\models$ `sold(hamlet)` and `sold(hamlet)` is unifiable with `sold(_)`.

4.7 Include Section

A module specification may contain an `Include:` section, which is defined as a list of module specifications preceded by the `Include:` label. Each time a module specification s is instantiated forming the module instance m, the system automatically instantiates all module specifications listed in the `Include:` section of the module specification s and includes them into the sharing scope of the module instance m. It should be noted that since a programmer does not obtain the identifiers of the module instances, it is not possible to manipulate them directly by means of the module actions. In particular, a programmer cannot perform the `exclude` action on any module instance that has been created by instantiating a module specification from the `Include:` section.

Example 5
Assume that the module specification `harry` is as follows:

```
Include:
  blockworld
```

```
Interface:
  at([X,Y])

Plans:
  adopta(at([5,4]))
```

Each time the module specification `harry` is instantiated, the multi-agent system also creates an instance of the module specification `blockworld`, which will be assigned a system-generated name (e.g. `blockworld#35df`). The two module instances belong to the same sharing scope and therefore the adoption of goal `at([5,4])` inside the instance of the `harry` specification may trigger PG-rules also inside the instance of the `blockworld` specification.

Chapter 5

Semantics

In this chapter we will introduce modular extensions to the 2APL semantics. Only transition rules and definitions relevant to our framework are presented, for the remaining parts of the language we refer to the earlier module proposal [7] and to the non-modular version of 2APL [6].

5.1 Multi-Agent System

The state of a multi-agent system can be expressed in terms of module instances, agents and the state of their shared environments[1].

Definition 1 (Multi-Agent System Configuration)
The configuration of the multi-agent system is a triple $\langle \mathcal{M}, \mathcal{A}, \chi \rangle$, where $\mathcal{M}$ is a set of module configurations, $\mathcal{A}$ is a set of identifiers that represent the main module instances of each agent and χ is the state of shared environments.

5.2 2APL Agent

The notion of a 2APL agent corresponds to a deliberation cycle that operates on the agent's mental state. An agent's mental state is modelled as a tree-like structure consisting of module instances, where the agent's main module instance serves as the root node.

5.3 2APL Module

The state of an individual module instance is captured in the form of a module configuration. It encapsulates the cognitive state of the module instance, but it also contains the components necessary for its function of a node in the module instance tree. Firstly, the stopping condition can be used to specify when the module instance should be automatically deactivated.

[1]The notion of shared environments has been taken from the non-modular 2APL [6]. It represents a set of internal states of all environments that are accessible to the agents.

Secondly, the module configuration contains the sets ϵ and υ that point to the children of the module instance in the module instance tree.

Definition 2 (Module Configuration)
A module configuration is a tuple $(M_\iota, \omega, \epsilon, \upsilon)$. The cognitive state of a module instance ι is denoted by $M_\iota = \langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle$, where ι is a string representing the identifier of the module instance, σ is a set of belief expressions `<belief>` representing the basic belief base of the module instance, γ is a list of goal expressions `<goal>` representing the basic goal base of the module instance, Π is a set of plan entries (`<plan>`, `<pgrules>`, `<ident>`) representing the plan base of the module instance, θ is a ground substitution that binds domain variables to ground terms, and $\xi = \langle E, I, M \rangle$ is the event base of the module instance. The module configuration further contains three execution related components: ω is a `<test>` expression representing the stopping condition of the module instance, ϵ is a set of module identifiers representing module instances the module instance ι secludes, and υ is a set of module identifiers representing module instances the module instance ι includes.

Module Linearisations

The module instances residing in the multi-agent system form a set of module instance trees. Such a structure can be linearised in a number of ways. In the following definitions we will assume a given multi-agent system $S = \langle \mathcal{M}, \mathcal{A}, \chi \rangle$.

Definition 3 (Module Linearisations)
Let υ_m be a set of identifiers of module instances the module instance m includes and ϵ_m be a set of identifiers of module instances the module instance m secludes. The set $\mathbb{M}$ contains identifiers of all module instances residing in the given multi-agent system.

$$\mathbb{M} = \{m : (M_m, \omega, \epsilon, \upsilon) \in \mathcal{M}\}$$

The set $\mathbb{D}_m$ contains identifiers of all descendants of module instance m. Note the recursive definition of $\mathbb{D}_m$ that contains also the children of the children.

$$\mathbb{D}_m = \{m\} \cup \bigcup_{m' \in \epsilon_m \cup \upsilon_m} \mathbb{D}_{m'}$$

The set $\mathbb{Q}_m$ contains identifiers of all queryable descendants of module instance m. Only the descendants accessible through the υ-links are queryable.

$$\mathbb{Q}_m = \{m\} \cup \bigcup_{m' \in \upsilon_m} \mathbb{Q}_{m'}$$

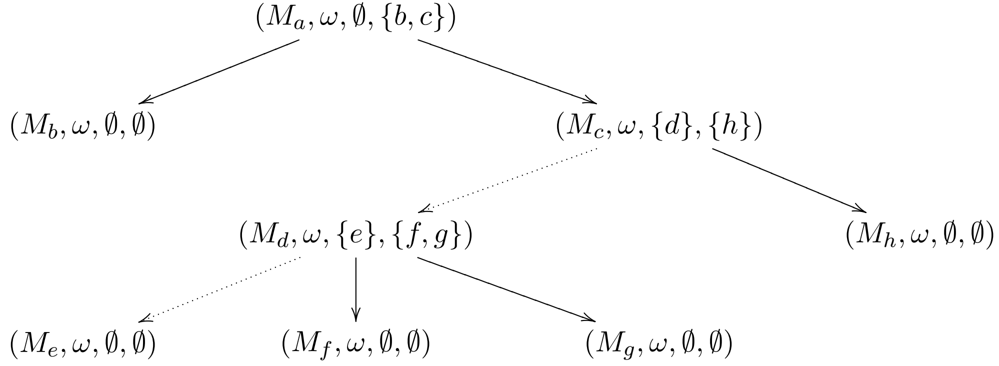
The set $\mathbb{A}$ contains identifiers of all active module instances in the given multi-agent system. The descendants of the main module instance of each agent are active.

$$\mathbb{A} = \bigcup_{m \in \mathcal{A}} \mathbb{D}_m$$

The set $\mathbb{I}$ contains identifiers of all inactive module instances in the given multi-agent system.

$$\mathbb{I} = \mathbb{M} \setminus \mathbb{A}$$

Example 6
Consider the following module instance tree consisting of eight module instances of the form $(M_\iota, \omega, \epsilon, \upsilon)$. A solid arrow $\longrightarrow$ denotes an included module instance (an υ-link), a dotted arrow denotes a secluded module instance (an ϵ-link).



We can obtain several linearisations of this structure. Firstly, the set of all descendants of module instance a: $\mathbb{D}_a = \{a, b, c, d, e, f, g, h\}$. Further, we define a set of queryable descendants, which is used to determine a sharing scope in which belief and goal sharing takes place. For example $\mathbb{Q}_a = \{a, b, c, h\}$ or $\mathbb{Q}_d = \{d, f, g\}$. Notice that a sharing scope is extended only by υ-links. In contrast, an ϵ-link creates a new belief/goal sharing scope.

5.4 Module Instantiation

A module instance can be created by means of two variants of the create action. The first version of the `create(`s`,`m`)` action creates a new module instance from a module specification s and assigns it the user defined identifier m. The second version creates a new module instance from module specification s and assigns a system-generated name to it. Such a name is guaranteed to be unique within the multi-agent system. Then, the system-generated identifier of the module instance gets unified with the variable m. We use _ to denote an anonymous variable, i.e. a variable that is irrelevant for purposes of a definition.

Definition 4 (Module Create Action with Assigned Name)

$$\frac{(M_\iota, \omega, \epsilon, \upsilon) \in \mathcal{M} \quad \& \quad M_\iota \xrightarrow{create(s,m)!} M'_\iota \quad \& \quad ground(m) \quad \& \quad m \notin \mathbb{M}}{\langle \mathcal{M}, \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A}, \chi \rangle}$$

where $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega, \epsilon, \upsilon)\}) \cup \{(M'_\iota, \omega, \epsilon, \upsilon), (M_m, \perp, \emptyset, \upsilon^*)\} \cup \mathcal{M}^*$. M_m is an initial mental state of the module instance m as specified in the module specification s. Further, $\mathcal{M}^*$ is a set that contains instantiated module specifications from the `Include:` section of the module specification s, and $\upsilon^* = \{m : (M^*_m, _, _, _) \in \mathcal{M}^*\}$.

Definition 5 (Module Create Action with Generated Name)

$$\frac{(M_\iota, \omega, \epsilon, \upsilon) \in \mathcal{M} \quad \& \quad M_\iota \xrightarrow{create(s,m\theta)!} M'_\iota\theta \quad \& \quad var(m) \quad \& \quad \theta = [m/genid(\mathcal{M})]}{\langle \mathcal{M}, \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A}, \chi \rangle}$$

where $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega, \epsilon, \upsilon)\}) \cup \{(M'_\iota\theta, \omega, \epsilon, \upsilon), (M_m, \perp, \emptyset, \upsilon^*)\} \cup \mathcal{M}^*$. M_m is an initial mental state of the module instance m as specified in the module specification s, the function $t = genid(\mathcal{M})$ generates a unique identifier t such that $t \notin \{m : (M_m, _, _, _) \in \mathcal{M}\}$. We assume $M'_\iota\theta$ to be the same as M_ι except that the `create` action has processed and the substitution $\theta = [m/t]$ is applied. Further, $\mathcal{M}^*$ is a set that contains instantiated module specifications from the `Include:` section of the module specification s, and $\upsilon^* = \{m : (M^*_m, _, _, _) \in \mathcal{M}^*\}$.

The `release(`m`)` action removes a module instance m from the multi-agent system. The release action succeeds only if the module instance m is inactive and results in the mental state of the module instance m (i.e. its basic belief base, basic goal base, plan base etc.) being discarded.

Definition 6 (Release Module Action)

$$\frac{(M_\iota, \omega, \epsilon, \upsilon) \in \mathcal{M} \quad \& \quad M_\iota \xrightarrow{release(m)!} M'_\iota \quad \& \quad (M_m, \omega', \epsilon', \upsilon') \in \mathcal{M} \quad \& \quad m \in \mathbb{I}}{\langle \mathcal{M}, \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A}, \chi \rangle}$$

where $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega, \epsilon, \upsilon), (M_m, \omega', \epsilon', \upsilon')\}) \cup \{(M'_\iota, \omega, \epsilon, \upsilon)\}$.

5.5 Module Activation

The `include(`m`,`φ`)` action adds the identifier of the module instance m to a set of included module instances of module instance ι and assigns the stopping condition φ to it. The action succeeds if the module instance m exists in the multi-agent system and if the module instance m and all its descendants are in an inactive state.

Definition 7 (Include Module Action)

$$\frac{(M_\iota, \omega, \epsilon, \upsilon) \in \mathcal{M} \quad \& \quad M_\iota \xrightarrow{include(m,\varphi)!} M'_\iota \quad \& \quad (M_m, \omega', \epsilon', \upsilon') \in \mathcal{M} \quad \& \quad \mathbb{D}_m \cap \mathbb{A} = \emptyset}{\langle \mathcal{M}, \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A}, \chi \rangle}$$

where $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega, \epsilon, \upsilon), (M_m, \omega', \epsilon', \upsilon')\}) \cup \{(M'_\iota, \omega, \epsilon, \upsilon \cup \{m\}), (M_m, \varphi, \epsilon', \upsilon')\}$.

The `seclude(`m`,`φ`)` action adds the identifier of the module instance$m$ to a set of secluded module instances of module instance ι and assigns the stopping condition φ to it. The action succeeds if the module instance m exists in the multi-agent system and if the module instance m and all its descendants are in an inactive state.

Definition 8 (Seclude Module Action)

$$\frac{(M_\iota, \omega, \epsilon, \upsilon) \in \mathcal{M} \quad \& \quad M_\iota \xrightarrow{seclude(m,\varphi)!} M'_\iota \quad \& \quad (M_m, \omega', \epsilon', \upsilon') \in \mathcal{M} \quad \& \quad \mathbb{D}_m \cap \mathbb{A} = \emptyset}{\langle \mathcal{M}, \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A}, \chi \rangle}$$

where $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega, \epsilon, \upsilon), (M_m, \omega', \epsilon', \upsilon')\}) \cup \{(M'_\iota, \omega, \epsilon \cup \{m\}, \upsilon), (M_m, \varphi, \epsilon', \upsilon')\}$.

A module instance m will be deactivated when 1) the `exclude(`m`)` action has been executed on the module instance m or 2) when the stopping condition ω of the module instance m has been satisfied. Then, the identifier of the module instance m is removed from the set of included module instances υ and the set of secluded module instances ϵ of the parent of the module instance m, i.e. the module instance ι. Note that since a module instance cannot be both included and secluded in the same time, the identifier of the module instance m can be a member of only one of the sets. The module instance m and all its descendants remain inactive in the multi-agent system.

Definition 9 (Exclude Module Action)

$$\frac{(M_\iota, \omega, \epsilon, \upsilon) \in \mathcal{M} \quad \& \quad M_\iota \xrightarrow{exclude(m)!} M'_\iota \quad \& \quad m \in \upsilon \cup \epsilon}{\langle \mathcal{M}, \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A}, \chi \rangle}$$

where $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega, \epsilon, \upsilon)\}) \cup \{(M'_\iota, \omega, \epsilon \setminus \{m\}, \upsilon \setminus \{m\})\}$.

Definition 10 (Check Stopping Condition)

$$\frac{(M_m, \omega, \epsilon, \upsilon) \in \mathcal{M} \quad \& \quad (\sigma^*_m, \gamma^*_m) \models_t \omega \quad \& \quad (M_\iota, \omega', \epsilon', \upsilon') \in \mathcal{M} \quad \& \quad m \in \epsilon' \cup \upsilon'}{\langle \mathcal{M}, \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A}, \chi \rangle}$$

where $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega', \epsilon', \upsilon')\}) \cup \{(M_\iota, \omega', \epsilon' \setminus \{m\}, \upsilon' \setminus \{m\})\}$, σ^*_m stands for the extended belief base of the module instance m, γ^*_m stands the extended goal base of the module instance m. These two concepts are further explained in the Section 5.6. The test entailment relation $\models_t$, which evaluates test expressions with respect to the belief and goal bases (σ, γ) is defined in [6] as follows:

- $(\sigma, \gamma) \models_t \texttt{B(}\phi\texttt{)}\tau \quad \Leftrightarrow \quad \sigma \models \phi\tau$
- $(\sigma, \gamma) \models_t \texttt{G(}\psi\texttt{)}\tau \quad \Leftrightarrow \quad \gamma \models_g \psi\tau$
- $(\sigma, \gamma) \models_t (\varphi \texttt{ \& } \varphi')\tau_1\tau_2 \quad \Leftrightarrow \quad (\sigma, \gamma) \models_t \varphi\tau_1$ and $(\sigma, \gamma) \models_t \varphi'\tau_1\tau_2$

5.6 Belief/Goal Sharing

Module Interface

Each module instance can specify a set of interface entries to control which parts of the basic belief base and the basic goal base of the module instance will be shared with the other module instances in the sharing scope. The interface entry takes the form of a Prolog-like atomic formula. When used in formal semantics, we interpret the interface entries as first-order logic atomic formulas. Then, for example, the interface entry `at(_,_)` would be interpreted as $at(\tau_1, \tau_2)$. A set of all interface entries specified in the module instance m is denoted by $\mathcal{I}_m$. The functions provided below are used to formalise which beliefs and goals will pass through the module interface.

Definition 11 (Interfaced Formula)
The function $I_m(\varphi)$ determines if an atomic formula φ is interfaced by a module instance m:

$$I_m(\varphi) = \begin{cases} \emptyset & \text{if } \neg\exists i \in \mathcal{I}_m : Unify(i, \varphi) \neq \bot \\ \{\varphi\} & \text{if } \exists i \in \mathcal{I}_m : Unify(i, \varphi) \neq \bot \end{cases}$$

Definition 12 (Interfaced Subset of Atomic Formulas)
The function $I_m(S)$ determines a subset of atomic formulas interfaced by a module instance m from a set of atomic formulas S:

$$I_m(S) = \bigcup_{\varphi \in S} I_m(\varphi)$$

The interface of a module instance can be used to control which beliefs from the given belief base will be interfaced.

Definition 13 (Interfaced Beliefs)
The function $I_m^B(\sigma)$ determines a set of all derivable atomic facts from belief base σ that are interfaced by module instance m:

$$I_m^B(\sigma) = \bigcup_{\sigma \models \varphi \ \& \ atomic(\varphi) \ \& \ ground(\varphi)} I_m(\varphi)$$

Likewise, the interface of a module instance controls which goals (or subgoals of individual goals) will be interfaced. First, we will define how the individual goals are interfaced. Recall that in 2APL an individual goal γ_i is represented as a conjunction of atomic facts.

Definition 14 (Interfaced Subgoal)
The function $I_m^g(\gamma_i)$ returns a subgoal of goal γ_i interfaced by a module instance m:

$$I_m^g(\gamma_i) = t'\left(I_m\left(t(\gamma_i)\right)\right)$$

where $t(\varphi)$ is a function that converts the formula φ of the form $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ to a set of atomic formulas $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ and function $t'(s)$ is a function that performs the inverse conversion, i.e. the conversion from a set of atomic formulas $s = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ to a conjunction of atoms $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$. Further we define $t'(\emptyset) = \bot$.

We can now define a function $I_m^G(\gamma)$ which operates on an entire goal base. Recall that a 2APL goal base γ is modelled as a list of individual goals $\gamma = [\gamma_0, ..., \gamma_i, ..., \gamma_n]$.

Definition 15 (Interfaced Goal Base)
The function $I_m^G(\gamma)$ returns a list that contains all subgoals from goal base γ interfaced by m.

$$I_m^G(\gamma) = \bigcup_{\gamma_i \in \gamma} \begin{cases} \emptyset & \text{if } I_m^g(\gamma_i) = \bot \\ \{I_m^g(\gamma_i)\} & \text{if } I_m^g(\gamma_i) \neq \bot \end{cases}$$

Example 7
Consider the following module specification:

```
Interface:
  b, c
```

```
Goals:
  a and b and c,
  b
```

The module instance will interface two goals: “b” and “b and c”. The reason is that:

1. $I^g_m(\texttt{a and b and c}) = \texttt{b and c}$
2. $I^g_m(\texttt{b}) = \texttt{b}$

Sharing Scope

The sharing of beliefs and goals takes place only between the module instances within one sharing scope. Each agent's main module instance creates a new sharing scope. Each included module instance becomes part of the sharing scope of its parent. In contrast, each secluded module instance establishes a new sharing scope. A module instance that established a new sharing scope (i.e. a main module instance or a secluded module instance) is called a sharing scope root and defined as follows:

Definition 16 (Sharing Scope Root)
The function $\textit{ss-root}(m)$ returns the predecessor of a module instance m that established the sharing scope to which the module instance m belongs.

$$\textit{ss-root}(m) = m' \in \mathbb{M} \texttt{ s.t. } \quad \big((\exists_{m^* \in \mathbb{M}} : m' \in \epsilon_{m^*}) \vee m' \in \mathcal{A}\big) \quad \& \quad m \in \mathbb{Q}_{m'}$$

Then $\mathbb{S}_m$ is a set that contains identifiers of all module instances that belong to the same sharing scope as module instance m.

Definition 17 (Module Sharing Scope)

$$\mathbb{S}_m = \mathbb{Q}_{\textit{ss-root}(m)}$$

The set $\mathbb{R}_a$ contains identifiers of all sharing scope roots of agent a.

Definition 18 (Sharing Scopes of Agent)

$$\mathbb{R}_a = \{s : s = \textit{ss-root}(m) \ \& \ m \in \mathbb{D}_a\}$$

Global Beliefs

The notion of global beliefs of a sharing scope represents an aggregation of interfaced basic beliefs from all module instances in the sharing scope. Since the global beliefs can be accessed only from within one of the module instances in the sharing scope, we define them relative to a module that performs a query on them.

Definition 19 (Global Beliefs)
A set of the global beliefs of the sharing scope of a module instance m contains interfaced basic beliefs of other module instances within the sharing scope. The basic belief base of module instance m' is denoted by $\sigma_{m'}$.

$$\bar{\sigma}_m = \bigcup_{m' \in \mathbb{S}_m \setminus \{m\}} I^B_{m'}(\sigma_{m'})$$

Global Goals

Likewise, the notion of global goals of a sharing scope represents an aggregation of interfaced basic goals from all module instances in the sharing scope. For the reasons explained above, we define them relative to a module instance that performs a query on them.

Definition 20 (Global Goals)
A list of global goals in the sharing scope of module instance m contains interfaced basic goals of other module instances within the same sharing scope. The basic goal base of module instance m' is denoted by $\gamma_{m'}$.

$$\bar{\gamma}_m = \bigcup_{m' \in \mathbb{S}_m \setminus \{m\}} I^G_{m'}(\gamma_{m'})$$

Extended Belief Base

In our modular framework, a belief query consults not only the basic belief base of the module instance in which was the belief query performed, but also a subset of global beliefs that are interfaced (imported) by the module instance. The combination of the basic beliefs of the module instance and the beliefs that the module instance imports through its interface constitutes the extended belief base of the module instance.

The basic belief base of a module instance σ_m is a set of Prolog belief expressions, which can be ground facts or derivation rules. The set of interfaced global beliefs $I^B_m(\bar{\sigma}_m)$ consists only of atomic ground facts. We define the extended belief base of the module instance σ^*_m as the basic belief base of the module instance augmented with the facts the module instance can obtain from the global belief base through its interface.

A module instance may test its beliefs for a number of purposes: Firstly, the beliefs of a module instance are queried to determine if the guard of a rule is satisfiable. Secondly, a

programmer may perform a test action on the belief base of a module instance and finally, the stopping condition of a module instance may query the belief base of the module instance. In all those cases and in contrast to non-modular 2APL, a module instance m will perform the belief queries on its extended belief base σ_m^*.

Definition 21 (Extended Belief Base)

$$\sigma_m^* = \sigma_m \cup I_m^B(\bar{\sigma}_m)$$

where σ_m is the basic belief base of the module instance m and $\sigma \cup S$ denotes an operation that extends a belief base σ with the atomic facts from a set S.

Extended Goal Base

Analogically to a belief query, a goal query consults not only the basic goal base of the module instance in which was the query performed, but also a subset of global goals that are interfaced (imported) by the module instance. The combination of the basic goals of the module instance and the goals that the module instance imports through its interface constitutes the extended goal base of the module instance.

The basic goal base of a module instance γ_m is a list of goals, each of them being a conjunction of atomic ground facts. The interfaced global goals $I_m^G(\bar{\sigma}_m)$ are also represented as a list of conjunctions of atomic ground facts. We define the extended goal base of a module instance γ_m^* as the basic goal base of the module instance concatenated with the list of goals the module instance can obtain from the global goal base through its interface.

A module instance may test its goals for a number of purposes: Firstly, the goal base is queried when a module instance tries to apply a PG-rule in order to determine whether the head of the rule is entitled by any of the goals of the module instance. Secondly, a programmer may perform a test action on the goal base of a module instance. Thirdly, the stopping condition of a module instance may test its goal base and finally, before any plan is executed, the head of the PG-rule is re-checked against the goal base to ensure that the plan is still desirable. In all those cases and in contrast to non-modular 2APL, the module instance m will perform the goal queries on its extended belief base γ_m^*.

Definition 22 (Extended Goal Base)

$$\gamma_m^* = \gamma_m \cup I_m^G(\bar{\gamma}_m)$$

where γ_m is the basic belief base of the module instance m and $\gamma \cup L$ denotes an operation that adds the goals from a list L to the end of a goal base γ.

Updating the Belief Base

In our modular framework, the adding and removing of beliefs has the identical semantics as in the non-modular version of 2APL and affects only the basic belief base of the module instance that executed the belief update.

Adopting Goals

A module instance can adopt new goals by means of the `adopta` and `adoptz` actions. The semantics of the goal adopting actions is identical to the non-modular version of 2APL. The specified goal expression is added to the beginning (in the case of `adopta`) or end (in the case of `adoptz`) of the basic goal base of the module instance. If the newly added goal is fully or partly interfaced, the interfaced part of the goal becomes global and the other module instances of the agent will be able to query it.

Dropping Goals

A module instance can drop its goals by means of the `dropgoal`, `dropsubgoals` and `dropsupergoals` actions. The semantics of the goal dropping actions stays the same as in the non-modular 2APL and therefore it only affects the basic goal base of the module instance.

5.7 Communication and External Environments

An agent sends a message by means of the $\texttt{send(}\iota, j, p, l, o, \phi\texttt{)}$ action. Contrary to the send action semantics as presented in the non-modular 2APL, the sender indicated in the message is not necessarily the module instance that executed the send action. Instead, we assume that the message sender is the module instance representing the root of the sharing scope to which this module instance belongs.

Definition 23 (Send Action)
Given a multi-agent system $S = \langle \mathcal{M}, \mathcal{A}, \chi \rangle$, let $(\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \in \mathcal{M}$ and $\iota \in \mathbb{A}$.

$$\frac{\varphi = \langle s, j, p, l, o, \phi \rangle \ \& \ s = \mathit{ss\text{-}root}(\iota) \ \& \ \gamma^*_\iota \models \mathcal{G}(r)}{(\langle \iota, \sigma, \gamma, \{(\texttt{send(}(j, p, l, o, \phi)\texttt{)}, r, id)\}, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \xrightarrow{\varphi\theta!} (\langle \iota, \sigma, \gamma, \{\}, \theta, \xi \rangle, \omega, \epsilon, \upsilon)}$$

where $\varphi\theta = \langle \iota, j\theta, p\theta, l\theta, o\theta, \phi\theta \rangle$. The broadcasting of the message is indicated by the exclamation mark in $\varphi\theta!$ which means that the transition proceeds by broadcasting the message event. The condition $\gamma^*_\iota \models \mathcal{G}(r)$, which is also the condition of other plan execution transition rules on the single module level, ensures that a plan is not executed if its purpose (i.e.,the goal for which the plan was generated) is not desirable anymore. The send action always succeeds. Note that this rule redefines the rule 11 from non-modular 2APL [6].

We introduce the analogical change also to the semantics of the external action execution. An agent performs an external action α in environment *env* by means of the

$\texttt{@}env\texttt{(}\alpha(t_1,\ldots,t_n),V\texttt{)}$ action. Contrary to the external action semantics as presented in the non-modular 2APL, the indicated source of the external action may not necessarily be the module instance that executed the action. Instead, we assume that the source is the module instance representing the root of the sharing scope to which this module instance belongs.

Definition 24 (External Action Execution)
Given a multi-agent system $S = \langle \mathcal{M}, \mathcal{A}, \chi \rangle$, let $(\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \in \mathcal{M}$ and $\iota \in \mathbb{A}$. The execution of external action broadcasts the event $env(s, \alpha(t_1\theta, \ldots, t_n\theta), t)$ to the multi-agent level indicating that one of the module instances in the sharing scope of the module instance s performs action $\alpha(t_1\theta, \ldots, t_n\theta)$, and waits for the return value t from the environment env in one transition step. A successful execution of external action will receive a return value t which is not falsum.

$$\frac{t \neq \bot \ \& \ s = ss\text{-}root(\iota) \ \& \ \gamma_\iota^* \models \mathcal{G}(r)}{\begin{array}{c}(\langle \iota, \sigma, \gamma, \{(\texttt{@}env\texttt{(}\alpha(t_1,\ldots,t_n),V\texttt{)}, r, id)\}, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \\ \xrightarrow{env(s,\alpha(t_1\theta,\ldots,t_n\theta),t)} \\ (\langle \iota, \sigma, \gamma, \{\}, \theta \cup \{V/t\}, \xi \rangle, \omega, \epsilon, \upsilon)\end{array}}$$

If the return value t is falsum, then the execution of the external action is considered as failed. The action is not removed from the plan base and its identifier is added to the event base of the module instance.

$$\frac{t = \bot \ \& \ s = ss\text{-}root(\iota) \ \& \ \gamma_\iota^* \models \mathcal{G}(r)}{\begin{array}{c}(\langle \iota, \sigma, \gamma, \{(\texttt{@}env\texttt{(}\alpha(t_1,\ldots,t_n),V\texttt{)}, r, id)\}, \theta, \langle E, I, M \rangle \rangle, \omega, \epsilon, \upsilon) \\ \xrightarrow{env(s,\alpha(t_1\theta,\ldots,t_n\theta),t)} \\ (\langle \iota, \sigma, \gamma, \{(\texttt{@}env\texttt{(}\alpha(t_1,\ldots,t_n),V\texttt{)}, r, id)\}, \theta, \langle E, I \cup id, M \rangle \rangle, \omega, \epsilon, \upsilon)\end{array}}$$

Note that the two aforementioned rules redefine the corresponding rules 13 and 14 from non-modular 2APL [6].

5.8 Planning Goal Rules

Plans are generated by applying PG-rules of the form κ `<-` β`|`π. The rule can be applied if the head of the rule κ is entailed by the extended goal base of the module instance, guard of the rule β is entailed by the extended belief base and if the plan base does not already contain a plan generated by this rule for the same goal[2]. We restrict this condition even more: The PG-rule can be applied only if there is no module instance in the same sharing scope pursuing a possibly overlapping goal (with respect to the goal sharing).

In the following definitions we will use $atoms(\varphi)$ to denote a function that returns a set of all atomic formulas the expression φ is composed of.

[2]More precisely, by a rule that has been applied using the same substitution to satisfy κ

Definition 25 (Pursued Subgoals)
The set Γ_m contains all atomic subgoals currently pursued by the module instance m.

$$\Gamma_m = \bigcup_{\kappa \in \{\kappa' : (\pi, \kappa' \texttt{ <- } \beta | \pi', id) \in \Pi_m\}} atoms(\kappa)$$

Definition 26 (Planning Goal Rule)
Given a multi-agent system $S = \langle \mathcal{M}, \mathcal{A}, \chi \rangle$, let $(\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \in \mathcal{M}$ and $\iota \in \mathbb{A}$. One of the PG-rules $r = \kappa \texttt{ <- } \beta | \pi$ specified in a module instance ι can be applied if: 1) κ is entailed by one of the goals from the extended goal base of the module instance, 2) guard β is entailed by the extended belief base of the module instance, 3) there is no plan in the plan base of the module instance that has been generated by applying the same PG-rule to achieve the same goal, 4) there is no module instance in the same sharing scope that is pursuing a possibly overlapping goal. Let P be a set of all possible plans, id be a new unique plan identifier, and $r' = \kappa' \texttt{ <- } \beta' | \pi'$ be a variant of r. Applying the PG-rule r will add an instantiation of the plan π' to the agent's plan base.

$$\frac{\begin{array}{r} \gamma_\iota^* \models \kappa'\tau_1 \quad \& \\ \sigma_\iota^* \models \beta'\tau_1\tau_2 \quad \& \\ \neg\exists \pi^\star \in P : (\pi^\star, (\kappa'\tau_1 \texttt{ <- } \beta | \pi), id') \in \Pi \quad \& \\ I_\iota(atoms(\kappa\tau)) \cap \bigcup_{m' \in \mathbb{S}_\iota \setminus \{\iota\}} I_{m'}(\Gamma_{m'}) = \emptyset \end{array}}{(\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \longrightarrow (\langle \iota, \sigma, \gamma, \Pi \cup \{(\pi'\tau_1\tau_2, (\kappa'\tau_1 \texttt{ <- } \beta | \pi), id)\}, \theta, \xi \rangle, \omega, \epsilon, \upsilon)}$$

Note that this transition rule replaces the rule 22 from non-modular 2APL [6].

5.9 Message and Event Handling

The agents in the multi-agent system communicate by sending messages to each other. Similarly, an agent can interact with an environment by executing an external action and the environment can communicate information back by generating external events that are received by the agent. Messages and external events are always addressed to a particular module instance in a multi-agent system. When a message or an external event is sent to a module instance, the message or an external event is added to the event base of the module instance.

However, in line with the idea of belief/goal sharing, we allow module instances within one sharing scope to process messages and external events from each other's event bases. That way, one module instance can apply a PC-rule for a message which has been received by another module instance. For this reason, we have altered the semantics of the external event/messages handling PC-rule application. In non-modular 2APL, messages and external events are drawn from the event base of the module instance, while in our proposal a module instance can react to any message or external event received by a module instance belonging to the same sharing scope.

Therefore, we define sets ξ_s^E and ξ_s^M that aggregate external events and messages that have been received by any of the module instances in the given sharing scope.

Definition 27 (Unprocessed Events and Messages in Sharing Scope)
The set ξ_s^E contains all unprocessed external events received by any of the module instances in the sharing scope of a module instance s.

$$\xi_s^E = \bigcup_{m \in \mathbb{S}_s} (E \text{ s.t. } (\langle m, \sigma, \gamma, \Pi, \theta, \langle E, I, M \rangle \rangle, \omega, \epsilon, \upsilon) \in \mathcal{M})$$

The set ξ_s^M contains all unprocessed messages received by any of the module instances in the sharing scope of a module instance s.

$$\xi_s^M = \bigcup_{m \in \mathbb{S}_s} (M \text{ s.t. } (\langle m, \sigma, \gamma, \Pi, \theta, \langle E, I, M \rangle \rangle, \omega, \epsilon, \upsilon) \in \mathcal{M})$$

Definition 28 (Procedure Call Rule)
Let $(\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \in \mathcal{M}$, $\iota \in \mathbb{A}$, $\texttt{event}(\phi, env) \in \xi^E_{ss\text{-}root(\iota)}$ be an event broadcasted by an environment env, $\texttt{message}(s, p, l, o, e) \in \xi^M_{ss\text{-}root(m)}$ be a message broadcasted by module instance s and $Unify$ be a function that returns the most general unifier of two atomic formulas (or returns $\perp$ if no unification is possible). Let φ `<-` β `|` π be a variant of a PC-rule of agent ι. The transition rule for applying PC-rules to the events from the event base of the sharing scope of the module instance ι is defined as follows:

$$\frac{\psi \in (\xi_\iota^E \cup \xi_\iota^M) \ \& \ Unify(\psi, \varphi) = \tau_1 \ \& \ \sigma \models \beta\tau_1\tau_2}{(\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \xrightarrow{eventprocessed(\psi, \iota)!} (\langle \iota, \sigma, \gamma, \Pi', \theta, \xi \rangle, \omega, \epsilon, \upsilon)}$$

where id is a new unique plan identifier, $r = (\texttt{true} \texttt{ <- } \beta \texttt{|} \pi), \Pi' = \Pi \cup \{(\pi\tau_1\tau_2, r, id)\}$.

The successful execution broadcasts an event $eventprocessed(\psi, \iota)$ caught by the multi-agent system, which removes an external event/message ψ from the event/message base of the respective module instance in the sharing scope of module instance ι.

If there is no applicable PC-rule in any of the module instances of the sharing scope of module instance ι, then the event or message will be removed from the respective event/message base.

$$\frac{\psi \in (\xi_\iota^E \cup \xi_\iota^M)) \ \& \ \forall_{m \in \mathbb{S}_\iota} : \forall r \in PC_m : Unify(\psi, \mathcal{G}(r)) = \perp}{(\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon) \xrightarrow{eventprocessed(\psi, \iota)!} (\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle, \omega, \epsilon, \upsilon)}$$

where PC_m is the set of PC-rules of module instance m. The successful execution broadcasts an event $eventprocessed(\psi, \iota)$ caught by the multi-agent system, which then removes the event/message ψ from the respective event/message base, such that $\psi \notin (\xi^E_{ss\text{-}root(\iota)} \cup \xi^M_{ss\text{-}root(\iota)})$. Note, the these two transitional rules redefine the respective rules 23 and 24 from non-modular 2APL [6].

Chapter 6

Execution Strategy

This chapter introduces extensions to the 2APL deliberation cycle. It presents the method employed by the 2APL interpreter to execute module instances of each individual agent.

6.1 Deliberation Cycle

In order to execute an individual agent, the 2APL interpreter follows a certain order of deliberation steps repeatedly and indefinitely. Each deliberation step is specified by a list of module identifiers denoting the order in which module instances make transitions and a set of transition rules indicating which transitions can each of the module instances make at the given moment at time. The order of deliberation steps is expressed using a deliberation cycle depicted on Figure 6.1.

Each deliberation step executes transitions on all descendants of the main module instance of agent a as specified by $\mathbb{M}_a$. For the purposes of the formal semantics definition, we considered module instance tree linearisations as non-ordered sets. However, since this chapter explains the method the 2APL interpreter employs to execute such a formal transition system, we can presuppose certain ordering of elements in the set $\mathbb{M}_a$. Specifically, we assume that the interpreter constructs such a set by performing breadth-first traversal of an agent's module instance tree. Therefore, the identifier of the agent's main module instance is always the first element in the set, followed by the identifiers of its immediate children, which are again followed by the identifiers of their children and so on.

Apply all PG-rules

The deliberation cycle starts by applying PG-rules in all the agent's module instances.

for all $m \in \mathbb{M}_a$ **do**
  apply-PG-rules-in(m)
end

Check Stopping Conditions

The deliberation process continues by checking whether the stopping condition of any of the agent's module instances has been satisfied. If this is the case, the module instance is

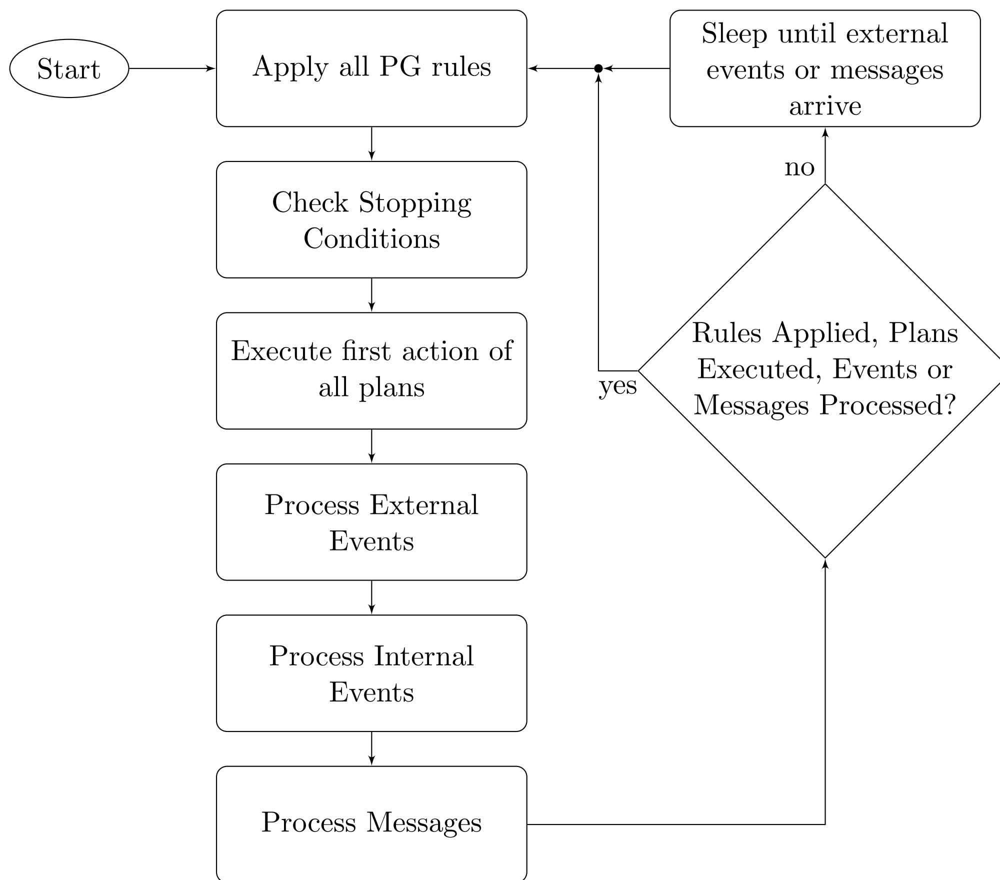


Figure 6.1: The deliberation cycle of an individual agent

deactivated and therefore excluded from the deliberation process.

```
for all $m \in \mathbb{M}_a$ do
  check-stopping-condition(m)
end
```

Execute First Action of All Plans

During the following step the interpreter executes the first actions of all plans in all the agent's module instances.

```
for all $m \in \mathbb{M}_a$ do
  execute-plans-in(m)
end
```

Process External Events

The next deliberation step moves all received external events to one of the temporary lists ξ_s^E based on the sharing scope they were addressed to. The agent's module instances can process events from the sharing scope they belong to. By doing so, a module instance broadcasts *eventprocessed*$(\psi, \iota)!$ event, which results in the removal of the external event from the temporary list.

```
for all $s \in \mathbb{R}_a$ do
  $\xi_s^E = \emptyset$
end

for all $m \in \mathbb{M}_a$ do
  $(\langle m, \sigma, \gamma, \Pi, \theta, \langle E, I, M \rangle \rangle, \omega, \epsilon, \upsilon) \in \mathcal{M}$
  $\xi^E_{ss\text{-}root(\iota)} = \xi^E_{ss\text{-}root(\iota)} \cup E$
  $\mathcal{M} = (\mathcal{M} \setminus \{(\langle m, \sigma, \gamma, \Pi, \theta, \langle E, I, M \rangle \rangle, \omega, \epsilon, \upsilon)\}) \cup \{(\langle m, \sigma, \gamma, \Pi, \theta, \langle \emptyset, I, M \rangle \rangle, \omega, \epsilon, \upsilon)\}$
end

for all $m \in \mathbb{M}_a$ do
  process-external-events-in(m)
  catch eventprocessed(ψ, ι)! do
    $\xi^E_{ss\text{-}root(\iota)} = \xi^E_{ss\text{-}root(\iota)} \setminus \{\psi\}$
  end
end
```

Process Internal Events

During this step the interpreter applies PR-rules for all internal events, which identify failed plans, in all the agent's module instances.

```
for all $m \in \mathbb{M}_a$ do
  process-internal-events-in(m)
end
```

Process Messages

Finally, this deliberation step moves all received messages to one of the temporary lists ξ_s^M based on the sharing scope they were addressed to. The agent's module instances can process messages from the sharing scope they belong to. By doing so, a module instance broadcasts $eventprocessed(\psi, \iota)!$ event, which results in the removal of the message from the temporary list.

```
for all s ∈ ℝ_a do
   ξ_s^M = ∅
end

for all m ∈ 𝕄_a do
   (⟨m, σ, γ, Π, θ, ⟨E, I, M⟩⟩, ω, ϵ, υ) ∈ 𝓜
   ξ^M_{ss-root(ι)} = ξ^M_{ss-root(ι)} ∪ M
   𝓜 = (𝓜 \ {(⟨m, σ, γ, Π, θ, ⟨E, I, M⟩⟩, ω, ϵ, υ)}) ∪ {(⟨m, σ, γ, Π, θ, ⟨E, I, ∅⟩⟩, ω, ϵ, υ)}
end

for all m ∈ 𝕄_a do
   process-messages-in(m)
   catch eventprocessed(ψ, ι)! do
      ξ^M_{ss-root(ι)} = ξ^M_{ss-root(ι)} \ {ψ}
   end
end
```

Chapter 7

Discussion

Since the programming constructs used to build agent-oriented programs are different to the ones offered by the traditional imperative languages, there is no obvious technique one can apply to modularise agent-oriented code. This chapter will discuss some of the options we have considered while looking for the most appropriate approach to the decomposition of agent programs.

7.1 Extending Object-Oriented Inheritance

One of the attractive options might be to reuse the programming constructs widely used in the object-oriented programming. In particular, the notion of inheritance might prove useful when applied to agent programming languages. The notion of inheritance is usually used as a tool that allows a programmer to define a new object as an extension of an object that has been defined earlier. The application of this idea helps a programmer to reuse parts of existing code.

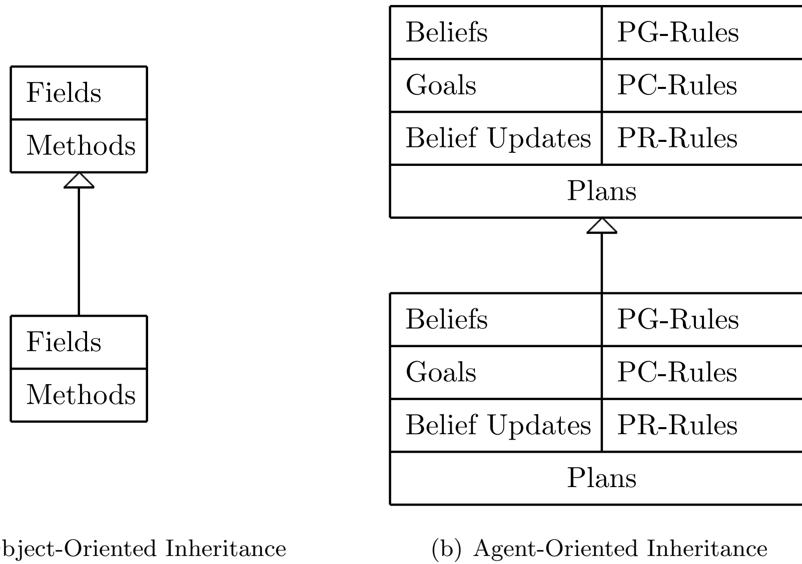
By making use of the inheritance mechanism, a programmer can easily add a new functionality that is not part of the base object, and redefine the functionality that does not exhibit the desired behaviour. In traditional object oriented languages, the inheritance principles can be applied with respect to fields, which are variables storing the program data, and methods, which are functions operating on the data as shown in Figure 7.1(a). One can envision the same principle being applied in 2APL on the level of BDI-concepts as illustrated in Figure 7.1(b).

Analogically to an object-oriented program being a composition of objects, we can assume that an agent program is a composition of BDI-module instances. A BDI-module instance is an encapsulation of a certain part of the agent's belief base, goal base, rule base, plan base and belief update specifications. An inheritance relation can be established between arbitrary module instances. Suppose that a module instance `B` is defined as an extension of a module instance `A`, that is `B` is inherited from `A`. Therefore all the beliefs, goals, rules and actions defined in `A` are also available in `B`.

One would expect that BDI inheritance will support the same style of programming as the object oriented inheritance. For example, abstract actions and belief update actions should resemble object methods. However, this is rather difficult to achieve.

Suppose that module instance `A`, among the others, contains the following PC-rule:

$$\alpha \texttt{ <- } true \texttt{ | } \{B(\phi); print(\phi)\}$$



(a) Object-Oriented Inheritance

(b) Agent-Oriented Inheritance

Figure 7.1: Object-oriented inheritance and envisioned use of the concept in 2APL

Let's further suppose that the module instance `B`, which is inherited from the module instance `A`, tries to execute the abstract action α, because one of its plans is of the following form:

$$\alpha; ...$$

The action α is not defined in module instance `B`, therefore its definition should be searched for in module instance `A`. If the module instance contains a rule that handles this abstract action α, it should be applied, i.e. copied into the plan base of `B`. Suppose that the abstract action α is expanded and changes the plan so it will have the following form:

$$B(\phi); print(\phi); ...$$

However, this will cause the belief test $B(\phi)$ to evaluate against the belief base of module instance `B`. This behaviour certainly does not satisfy the properties one would expect from an inheritance mechanism. The body of the abstract action was specified in the context of module instance `A` implying that the module programmer could have only assumed certain beliefs to be part of the belief base of module instance `A`.

One of the ways to overcome this unintuitive behaviour might be to specify the context in which the action should be executed. In our example, expanding the abstract action would have to also add the relevant context to each plan element:

$$A.B(\phi); A.print(\phi); ...$$

However, there are several problems the latter solution will introduce. Firstly, a module instance can contain several plans to be executed in the interleaved mode. Thus, it is possible to release the module instance while the expanded body of the inherited PC-rule is being executed. Therefore the context of the action may cease to exist in the middle of the plan execution.

More importantly, the encapsulation principle is seriously violated by applying the BDI-inheritance mechanism explained above. According to the encapsulation principle, a module

instance should expose only the interface to its functionalities, not their actual implementation. By expanding a PC-rule inherited from a different module instance for a certain abstract action, the body, which can be seen as an implementation of the action, is carbon-copied to the plan base of the calling module instance. In other words, the exact implementation details are exposed outside the defining module instance and the calling module instance can easily interfere with it, for example by applying PR-rules.

Furthermore, hiding of the internals of a module instance, i.e. the distinction between public and private components, becomes infeasible. The body of a PC-rule expects to have full access to the internals of the module instance in which it was defined. This is possible only if the full access between the calling module instance and the called module instance is allowed. Therefore, the information hiding principle is impossible to satisfy in such a setting.

In our opinion, the presented notion of a BDI-inheritance would not contribute to the maintainability of an agent program. Due to the different nature of the object-oriented programming and agent-oriented programming, the notion of object-oriented inheritance does not seem to be easily transferable to the realm of the agent programming.

7.2 Execution Strategies

BDI agents are assumed to consist of two parts: the agent's mental state and the deliberation cycle that executes transitions on the agent's mental state. In 2APL, the deliberation cycles of different agents are independent of each other as they are executed in parallel in separate threads.

The question that arises here is how does the agent's deliberation cycle deal with the agent's mental state that is partitioned into a number of separate module instances. One can think about several strategies to execute the individual module instances:

7.2.1 Serial Module Execution

The most straightforward way is to allow only one active module instance in an agent a time. The module instance that executes its child module instance looses the execution control in favour of the module instance it is executing. It will regain the execution control back only after the stopping condition of the executed module instance has been satisfied.

This approach resembles the method calling stack in imperative languages - in a given time point, there is always exactly one method that holds the execution control. The advantage of this approach is that the programmer can easily visualise and debug the way the program gets executed. He does not need to worry about the synchronisation issues as all the resources are always accessed from maximum one program thread a time. On the other hand, this serial approach might be limiting in the situations when more than one concern needs to be handled at the same time.

7.2.2 Parallel Module Execution

One can suggest to realize module execution in such a way that each module instance is assigned a separate deliberation cycle executed in its own thread. That would allow us to represent module instances as sub-agents of a particular top-level agent. This approach would be promising for its inherent high level parallelism that can be fruitfully exploited by the expected many-core processor architectures. On the other hand, it also introduces the

problem of data consistency errors, which need to be handled with special care each time two module instances interact. The need for the low-level synchronisation would demand for a limited set of methods two module instances can use to interact. In practice, it would allow only asynchronous messaging and the manipulation of inactive module instances.

Consider the following problem as an illustration: It is not safe to perform a belief query $m.B(\varphi)$ on an active module instance m as it may get executed at the same time as the belief update action affecting the belief φ. This form of interaction leads to unpredictable results. It is therefore only safe to perform queries on a module instance when the deliberation cycle of the module instance is not performing any updates, i.e. when the module instance is inactive.

7.2.3 Interleaved Module Execution

We propose to settle down for a mid-ground between these two strategies. We consider an agent as an execution of the deliberation cycle, which consults all the agent's active module instances and processes them serially in a predefined order. That means, for example, that during "Apply All PG-Rules" deliberation step, all active module instances are processed sequentially so that each of them can apply the PG-rules from its rule base.

Although this approach does not allow for truly parallel system-level processing, it ensures that there is only one module instance performing a transition step in a time and therefore it lets programmers access other module instances without worrying about data consistency. The execution of plans in the module instances of an agent's is synchronized and interleaved. The execution method guarantees to execute exactly one action of each plan in each active module instance during one deliberation cycle. The order in which the deliberation cycle processes the agent's module instances is determined by a breadth-first traversal of the agent's module instance tree. Therefore, any parent module instance will be always processed before its children.

Such an interleaved module execution strategy lets a parent module instance react to events (such as a belief base update or goal base update) originating from one of its descendants immediately after they occur. Moreover, the parent module instance can apply its rules to handle such an event before its descendants get chance to apply their rules. Consequently, the parent module instance may monitor and possibly override plan execution in its descendants. Consider the following example as an illustration:

Imagine a bomb disposing agent that uses an instance of `bombsearching` module specification to search the environment for bombs. When the `bombsearching` module instance is activated, it moves the agent randomly in the environment and senses for bombs. Once it spots a bomb, the module instance adopts a new belief `bomb([X,Y])` containing the coordinates of the newly found bomb. Let's assume that beliefs of the form `bomb([X,Y])` are interfaced by the `bombsearching` module instance. Further, we assume that the agent's main module is specified in the following way:

```
Interface:
  bomb([X,Y])

Plan:
  create(bombsearching, BombSearching);
  include(BombSearching);
  [B(bomb([X,Y]));
```

```
exclude(BombSearching)];
...
```

The initial plan instantiates the module specification `bombsearching` and activates the resulting module instance. Since at first none of the module instances is aware of any bombs in the environment, the test action `B(bomb([X,Y]))` fails. The test action stays in the plan base of the main module instance and will be retried in each subsequent deliberation cycle. In the mean time, the `bombsearching` module instance searches the environment for bombs. As soon as it observes a bomb, it performs a belief update action that adds a belief of the form `bomb([X,Y])` to its basic belief base. Then, during the directly following deliberation cycle, the belief test `B(bomb([X,Y])` will succeed and the `bombsearching` module instance will be deactivated. Since the plans of these two modules are executed synchronously in the interleaved mode, it is always the case that the `bombsearching` module instance will be deactivated immediately, leaving the plan that follows after the belief update action unexecuted.

Note that this would not hold if the agent's module instances were executed in an unsynchronized manner in separate threads. In such a setting, a programmer cannot assume that the main module instance will notice the change in the global belief base right after the belief update has been performed. Instead, the `bombsearching` module instance may proceeded by executing the plan that follows after the belief update action. This is because the precise moment in which will be the belief test retried in the main module instance is decided by the scheduler of the operating system and in general impossible to predict.

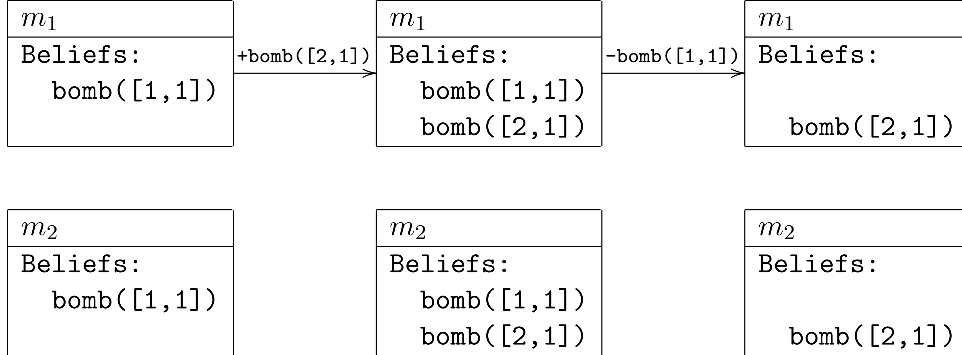
7.3 Updating Beliefs and Golals

7.3.1 Updating the Belief Base

A module instance updates its beliefs by means of belief update actions. Execution of a belief update action may add or remove beliefs stored in the basic belief base of the module instance. In our proposal, we decided that adding and removing beliefs will have identical semantics as in the non-modular version of 2APL. Therefore, it will affect only the basic belief base of the module instance on which was the update executed. However, it is possible to think about several alternative ways to tackle the problem of shared belief synchronisation. In the following paragraphs we will discuss other approaches and point out problems connected with each of them.

Suppose that **all module instances maintain a private copy of the interfaced global beliefs**. That is, adding a belief φ in one module instance will add the same belief φ to the basic belief bases of all module instances in the same sharing scope that interface the belief. The following picture demonstrates how would such a mechanism work. It consists of two module instances: module instance m_1 and module instance m_2, both interfacing beliefs of the form `bomb(_)`. We can see that initially, both module instances hold the identical belief `bomb([1,1])` in their basic belief base. Further, the picture shows that adding the belief `bomb([2,1])` in module instance m_1 will materially add the belief to the basic belief base of both module instance $m1$ and module instance m_2. Likewise, removing the belief

`bomb([1,1])` in m_1 will also materially remove the belief from the basic belief base of m_2.

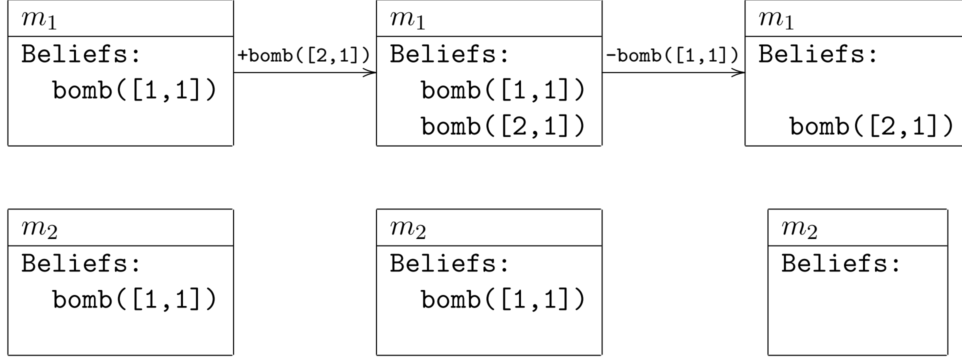


When module instance m_1 performs a belief update action that adds the belief `bomb([2,1]` to its basic belief base, this belief becomes global. Thus, module instance $m2$ also adds the belief `bomb([2,1])` to its basic belief base in order to maintain it consistent with the global beliefs. Notice, that this approach requires that a module instance copies all interfaced global beliefs to its basic belief base right after it has been included. To mention just one problem, this approach suffers from severe memory inefficiency and data duplication. Imagine for example a scenario in which one module instance contains a database of thousands of books in its basic belief base. All those beliefs would have to be materially copied to the basic belief bases of all module instances that would like to use (i.e. interface) the database.

Another approach would be that **a module instance stores just the beliefs it itself adopted**. Adding a belief then only affects the basic belief base of the module instance that has executed the belief update action. The belief querying is performed on the extended belief base of the module instance and therefore still reflects the global beliefs of the sharing scope of the module instance.

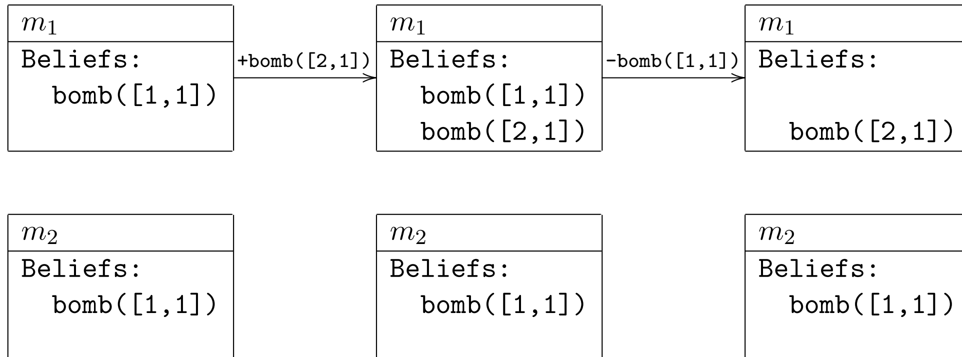
The question that arises here is: What will be the semantics of the belief removal? Again, there are two possibilities. The choice of the most rational approach is dependent on the intuitive interpretation we assign to the act of a belief removal. Since 2APL uses a Prolog based belief base, it cannot store negative facts. This leads to an ambiguity in the way how belief removing can be interpreted. On one hand, removing a belief φ may signify that an agent started to be ignorant of φ, i.e. it believes neither φ nor $\neg\varphi$. On the other hand, the removal of a belief φ may signify that the agent started to deny φ, i.e it started to believe that $\neg\varphi$ holds.

The option we will discuss first is to keep the effect of belief removal global, while the effect of adding a belief stays local. This approach is based on the interpretation that sees removing a belief as believing its negation. Under this interpretation, it is rational for other module instances to also remove the belief from their basic belief bases. The effect of this approach is illustrated in another version of our running example.



The first problem of this approach is that it violates the design principle of locality. A belief update action executed in one module instance can have consequences in other module instances. Furthermore, the consequence of this design is an inconsistency in the way how belief updating works. Adding of a belief updates only the basic belief base of the module instance, while removing of a belief may in general update the basic belief bases of all module instances in the sharing scope in which was the update performed. One may argue that this kind of inconsistency requires special awareness and puts more burden on a programmer analysing the 2APL code.

The last option is based on the interpretation that sees removal of φ as ignorance of φ. Then, after one of the module instances removes the interfaced belief φ from its basic belief base, it might be the case that some other module instance still believes in the validity of φ, therefore it is rational to keep such information and assume that removing of the belief φ affects only the basic belief base of the module instance in which it was performed. This approach also respects the design principle of locality. However, under this setting, a remove action might have no actual effect on the extended belief base of the module instance. Consider the following variant of our running example:



We can see that even after the module instance m_1 has removed the belief `bomb([1,1])`, the test `B(bomb([1,1]))` would succeed. Consequently, there might exist a post-condition of a belief update action that is unsatisfiable, which means that a belief update action may fail both on its pre-condition and its post-condition.

We have rejected the first approach due to its inherent memory inefficiency. We believe that it is impossible to choose a superior option from the remaining two since the interpretation given to the belief removal is application dependent. However, we have decided to adopt the last mentioned approach (the one in which the belief removal is performed locally) for its simpler and more consistent semantics.

7.3.2 Dropping Goals

A module instance can execute one of the actions `dropgoal`, `dropsupergoals` or `dropsubgoals` in order to drop some of its goals. One may wonder what happens when a goal that is to be dropped has been imported through the interface and is therefore not part of the basic goal base of the module instance. One approach would be to assume that any module instance should have control over the agent's global goal base and therefore should be able to manipulate any interfaced goal. The problem we would encounter if we used such approach is that the interface may make available only a certain subgoal of a global goal. Consider for example an agent m consisting of two module instances m_1 and m_2:

m_1

`Interface:`
`  a`
`Plans:`
`  dropgoal(a)`

m_2

`Interface:`
`  a`
`Goals:`
`  a and b`

In this example, module instance m_2 has one goal `a and b`. Because `a and b` $\models$ `a`, the subgoal `a` becomes global goal of the agent. Since module instance m_1 contains the interface entry `a`, the goal `a` also becomes goal of the module instance $m1$. Now consider that the `dropgoal(a)` action is executed in the module instance m_1. The goal `a` is in fact subgoal of the goal `a and b`, which is a part of the basic goal base of the module instance m_2. In order to remove `a` from module instance m_1, one would have to remove either the whole goal `a and b` or to remove subgoal `a` from this goal, so it becomes only `b`. The former approach violates the separation of concerns principle as the action has unrelated consequences in the module instance m_2, the later approach is not consistent with the way the `dropgoal` action works within an individual module instance, i.e. all versions of the goal dropping actions always remove the entire goal from the goal base.

For the reasons demonstrated above and in the sake of conceptual clarity, we have decided to only allow goal manipulation to take place on the basic goal base of a module instance. This means that the module instance can only drop goals it has itself adopted. The semantics of the drop goal actions therefore stays the same as in the non-modular version of 2APL [6].

7.4 Planning goal rule

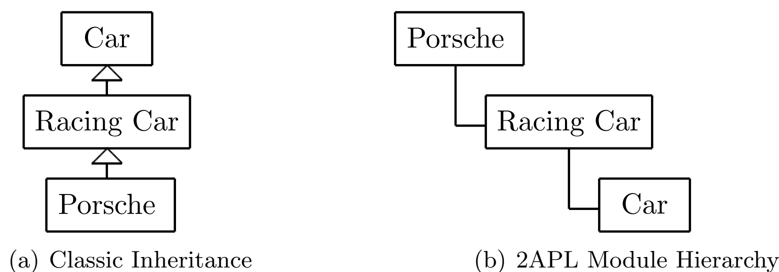
In 2APL, plans are generated by applying PG-rules of the form κ `<-` $\beta \mid \pi$. In the original 2APL semantics [6], a rule can be applied if the same rule has not been already used for the same goal. We restrict this condition even more: a PG-rule can be applied only if none of the subgoals it will pursue is not pursued already by some other module instance.

This is to save a programmer from worrying about the possible interrelations between two plans from different module instances that are trying to achieve the same goal. Within one module instance, the programmer is assumed to be familiar with the code and take the possible interactions between the plans into account. While on the agent/scope level, we assume that the programmer might not be familiar with internals of each of the agent's module instances (e.g. because he is not the author of the code) and therefore should not be expected to foresee possible conflicts between the different plans.

Furthermore, this constraint will allow us to organise code into inheritance-like hierarchies. Consider an example of a typical object-oriented inheritance hierarchy and its 2APL counter-

part in Figure 7.2(a) and 7.2(b). Suppose that a module instance represents a specialisation of some general concept, which is implemented in another module instance. Then a programmer can use the notion of inclusion to inherit the common traits from it. In our example, `Porsche` inherits from `Racing Car`, which in turn inherits from `Car`. In object oriented programming, we would create an inheritance hierarchy from the most general class to more specialised classes as illustrated in Figure 7.2(a). In our modular framework, we can structure the code in a similar manner as shown in the Figure 7.2(b): the specialised module instance `Porsche` includes a more general module instance `Racing Car`, which again includes a more general module instance `Car`.

In object oriented programming, one would expect that the methods not explicitly redefined in the specialised class will be invoked implicitly from one of the super-classes. Analogically in 2APL, goals that are not handled by the specialised module instance will be automatically handled by one of the previously included general module instances. In other words, if there is no PG-rule in the specialised module instance to handle a particular goal adopted by an agent, the interpreter will search further in the module instances that were included by the specialised module. Conversely, if there is an applicable PG-rule to handle a particular goal, the interpreter stops searching for applicable PG-rules in other module instances and handles the goal internally within the module instance in which was the match found first.



(a) Classic Inheritance (b) 2APL Module Hierarchy

Figure 7.2: Example: 2APL Module Hierarchy Mimicking Inheritance

7.5 Communication and Interaction with External Environments

Communication and interaction with external environments are central issues in agent-oriented programming. Traditionally, all messages directed to an agent are assumed to end up in an agent's incoming message queue, which is continuously processed by the agent's deliberation cycle. This also holds for received external events. However, following the introduction of modules, it might be necessary to revise this concept. We can identify several possible ways to handle 1) communication between two modular agents and 2) interaction between a modular agent and an environment.

One approach is to maintain only one message queue and one event queue per agent. Then, the two queues would be shared by all agent's module instances. Alternatively, an agent can maintain separate message queue and event queue for each of his module instances. This means that all messages have to be directed specifically to one of the agent's module instances. Similarly, each module instance would be represented as an independent entity

when interacting with an environment.

We propose a somewhat hybrid approach based on the concept of a sharing scope. For purposes of belief and goal sharing we assume that the module instances belonging to one sharing scope pursue the same goals and believe consistent facts. We can extend this concept also to the reactive components of an agent, i.e. for messages and external events. The module instances within one sharing scope will share a common message queue and a common queue of external events. Further, all messages sent from a certain sharing scope will indicate that the root module instance of the sharing scope is the message sender. Likewise, all actions performed in an external environment will seemingly originate in the root module instance of the sharing scope.

We believe that the aforementioned approach goes in line with the concept of sharing scope as used for belief and goal sharing and therefore keeps the framework consistent and easy to grasp.

Chapter 8

Demonstration

8.1 Bomb Disposal

In the following example we will demonstrate the most common application of our modular system. We will use it to implement the concept of a capability. The example shows a multi-agent system consisting of one manager and three worker agents whose objective is to find bombs in the blockworld environment and dispose them into a designated bomb trap. Each worker agent controls an assigned entity in the blockworld environment that can be used to a) move within the environment, b) sense the surroundings[1] and c) pick-up, carry and drop the bombs. Once the worker agent spots a new bomb, he informs the manager agent who aggregates the knowledge about the found bombs and subsequently orders worker agents to dispose them. A worker agent therefore has two major tasks: 1) to look for the bombs and 2) to dispose them when ordered.

In our example we have divided the functionality needed to perform the two major tasks into separate modules. The `bombsearching` module instance actively searches the environment and maintains a database of the believed bomb positions. Analogically, the `disposing` module instance contains the functionality needed to dispose the given bomb to the trap. Further, the `blockworld` module instance provides higher-level agent-oriented methods for accessing the blockworld environment.

The main module specification of a worker agent lists the specifications of the aforementioned modules in its `Include:` section:

worker

```
Include:
  blockworld, bombsearching, disposing

  ...
```

Once the multi-agent system spawns a worker agent, it instantiates the module specification `worker`, which in turn instantiates the specifications `blockworld`, `bombsearching` and `disposing`, and includes the resulting module instances to its sharing scope. The interaction between the four module instances is realised through their interfaces, which are depicted in the Figure 8.1.

[1] Limited by a predefined sensing range.

In the following sections we will illustrate some of the typical patterns of interaction between module instances. Please see Appendix A for the complete source code of this example program.

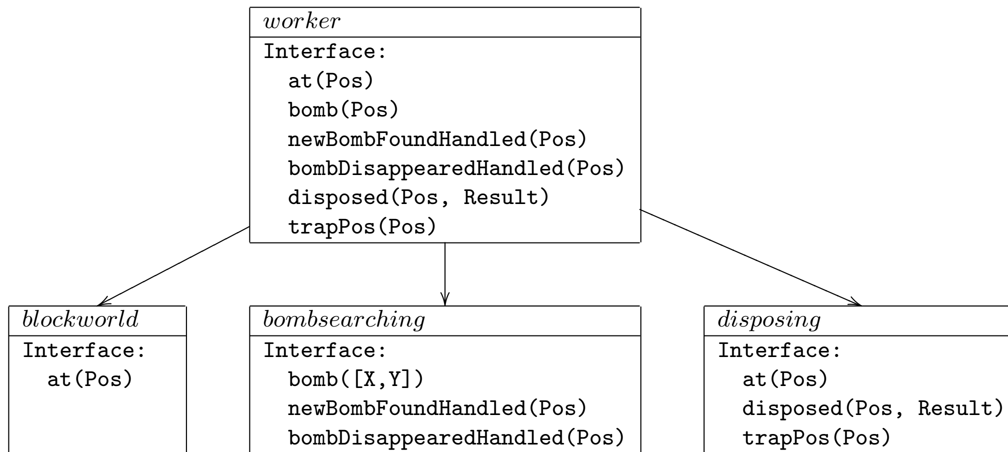


Figure 8.1: Modules of the Worker Agent

Providing Required Belief

In case the function of a module instance relies on the availability of information it cannot itself obtain, the belief sharing mechanism can be used to retrieve such information from outside the module instance. In our example program, the `disposing` module instance, which is responsible for carrying the bombs and disposing them into the trap, needs to first know the location of the trap. As it does not have any internal means to determine the trap's position, the information must be provided from some other module instance, which is in our case the main `worker` module instance.

The `disposing` module instance queries the belief `TrapPos(Pos)` since the programmer can assume that the belief will be provided in the form of a global belief:

disposing

```
Interface:
  trap(TrapPos)

PG-rules:
  disposed(Pos, succeeded) <- trap(TrapPos) | {
    ...
    adopta(at(TrapPos));
    B(at(TrapPos));
    ...
  }
```

The concrete belief is specified in the basic belief base of the `worker` module specification and exported:

```
worker
Interface:
  trap(TrapPos)

Beliefs:
  trap([0,0]).
```

Delegating Goal Pursuit

Another typical design pattern making use of the belief/goal sharing is the adoption of a goal that the module instance itself is incapable to achieve. In such a case the programmer assumes that there is some other module instance within the sharing scope that will be able to pursue and eventually achieve the goal. Once the goal is believed to be achieved, the goal is automatically dropped. The pursuit of the goal can be monitored by query on the corresponding belief. In our example, the `disposing` module instance may e.g. adopt the goal `at([5,7])` in order to initiate movement to the given position in the blockworld, although it has no actual means to achieve it.

Since atom `at(Pos)` is declared as an interface entry in the `disposing` module specification, the goal `at([5,7])` will be exported and becomes a global goal of the agent[2]. The `blockworld` module instance also declares atom `at(Pos)` as an interface entry and therefore imports the global goal. Consequently, it applies the corresponding PG-rule(s) to pursue it. The goal is assumed to be achieved as soon as the agent believes to be at the target position `at([5,7])`.

In the following code excerpt, one of the plans of the `disposing` module instance starts desiring to be at the `BombPos` position and blocks until the goal is believed to be achieved.

```
disposing
Interface:
  at(Pos)

PG-rules:
  disposed(BombPos, succeeded) <- trap(TrapPos) | {
    adopta(at(BombPos));
    B(at(BombPos));
    ...
  }
```

The PG-rules specifying how the goal can be achieved are defined and applied in the `blockworld` module instance. The module instance updates the agent's current position belief after each movement step and exports it using the `at(Pos)` interface. Consequently, the belief is imported into the `disposing` module instance, which is forced to drop the goal as soon as the imported belief states that the goal has been achieved.

```
blockworld
Interface:
  at(Pos)
```

[2]Precisely speaking, it will become a global goal of the sharing scope of the `disposing` module instance, but since the agent consist of just one sharing scope, it is not necessary to make this distinction.

```
BeliefUpdates:
  { at(OLDPOS) } UpdatePosition(POS) { not at(OLDPOS), at(POS) }

PG-rules:
  at([X, Y]) <- at([CurrentX, CurrentY]) and X < CurrentX | {
    [@blockworld( west(), _ );updatePosition()]
  }

  at([X, Y]) <- at([CurrentX, CurrentY]) and X > CurrentX | {
    [@blockworld( east(), _ );updatePosition()]
  }

  at([X, Y]) <- at([CurrentX, CurrentY]) and Y < CurrentY | {
    [@blockworld( north(), _ );updatePosition()]
  }

  at([X, Y]) <- at([CurrentX, CurrentY]) and Y > CurrentY | {
    [@blockworld( south(), _ );updatePosition()]
  }

PC-rules:
  updatePosition() <- true | {
    @blockworld(sensePosition(), RES);
    B(RES = [actionresult(POS)]);
    UpdatePosition(POS)
  }
```

Delegating Action Performance

Since our modular framework is based on the idea of module instances sharing their beliefs and goals, it does not offer an explicit construct that would substitute the notion of calling a method known from object orientated languages. However, in case a programmer needs to perform an action implemented in another module instance, he can do so by representing the action in the form of a performance goal and applying the goal pursuit delegation technique explained in the previous section.

However, as 2APL treats all goals as achievements goals, we need to introduce a temporary belief that describes a world state in which the action was successfully performed. Then, the performance goal can be expressed as a desire to achieve such a world state. In case the action can be performed repetitively during the agent's lifetime, the associated belief should be of transient nature and removed shortly after it has been adopted.

In our example program, one of the plans defined in the `disposing` module specification needs to perform the pick-up bomb action in the blockworld environment. As it does not have the access to the environment itself, it assumes that the agent contains the `blockworld` module instance capable of executing such actions in the blockworld environment and adopts a goal representing its interest in the action being successfully performed. Once the pick-up action

has been performed with a certain result, the agent will believe so for one deliberation cycle. Hence, the belief test following immediately the adoption of the goal serves two purposes: firstly, it blocks the plan execution until the action has been performed, and secondly it allows obtaining the outcome of the action performance.

```
disposing
Interface:
  pickedUp(Result)

PG-rules:
  disposed(BombPos, succeeded) <- trap(TrapPos) | {
    ...
    adopta(pickedUp(succeeded));
    B(pickedUp(Result));
    if B(Result = succeeded) then {
      ...
    }
    if B(Result = failed) then {
      ...
    }
    ...
  }
```

The `blockworld` module specification contains a PG-rule that executes a pick-up action in the blockworld environment and updates the basic belief base of the module instance with the belief `pickedUp(succeeded)`, which signals a successful completion of the action. A possible failure of the action is handled by the corresponding PR-rule, which updates the basic belief base with the belief `pickedUp(fail)`. As both variants of the belief are transitional, the second, reactive PG-rule is applied during the following deliberation cycle to re-establish the default state by removing the belief from the basic belief base of the module instance.

```
blockworld
Interface:
  pickedUp(Result)

BeliefUpdates:
  { } PickedUp(Result)    { pickedUp(Result) }
  { } NotPickedUp(Result) { not pickedUp(Result) }

PG-rules:
  pickedUp(succeeded) <- true | {
    @blockworld( pickup(), _ );
    PickedUp(succeeded)
  }
  <- pickedUp(Result) | NotPickedUp(Result)

PR-rules:
  @blockworld( pickup(), _ ); REST <- true | {PickedUp(failed)}
```

Delegating Event Handling

In some situations it is not desirable to handle events in the same module instance in which they have been recognized. In our example, the `blockworld` module instance generates an event each time a previously unknown bomb has been sensed. However, to keep the module implementation general and reusable, the actual event handling is delegated to another module instance of the agent. One of the approaches for event handling delegation is a variant of the technique explained in the previous section.

Each time the `bombsearching` module instance spots a new bomb, it updates its basic belief base and adopts the goal `newBombFoundHandled(Pos)` representing its desire to handle the event. Since the module instance itself is not able to achieve the goal, it assumes that it will be pursued in some other module instance.

bombsearching
```
Interface:
  newBombFoundHandled(Pos)

PG-rules:
  foundAllBombs <- true | {
    ...
    /* New bomb has been found. */
    Bomb([X,Y]);
    adopta(newBombFoundHandled([X,Y]));
    B(newBombFoundHandled([X,Y]));
    ...
  }
```

The `worker` module specification defines a PG-rule that handles the event. In our case, the module instance informs the manager agent about the new bomb. At the end of the generated plan, its basic belief base is updated with the `newBombFoundHandled(Pos)` belief to signal that the event handling has been completed. Analogically to the action completion beliefs, such belief is transitional. Therefore the dedicated reactive PG-rule is applied in the following deliberation cycle to re-establish the default state by removing the belief.

worker
```
Interface:
  newBombFoundHandled(Pos)

BeliefUpdates:
  {} NewBombFoundHandled(Pos)    {newBombFoundHandled(Pos)}
  {} NotNewBombFoundHandled(Pos) {not newBombFoundHandled(Pos)}

PG-rules:
  newBombFoundHandled(Pos) <- manager(Manager) | {
    send(Manager, inform, bomb(Pos));
    NewBombFoundHandled(Pos)
  }
  <- newBombFoundHandled(Pos) | NotNewBombFoundHandled(Pos)
```

Chapter 9

Conclusion

In this thesis we have introduced a belief/goal sharing modularisation framework suitable for BDI-based agent programming languages with declarative goals. In order to evaluate the viability of the proposed concept, we have formally specified it as an extension of the 2APL programming language, which serves as a representative of such a language family. In chapters 4, 5 and 6, the changes to syntax, semantics and the deliberation cycle of the language have been presented. Further, we have implemented the proposed extensions in the 2APL interpreter and developed a few example programs making use of the new features. In chapter 8, we demonstrate the practical applicability of our framework by giving examples of design patterns that take advantage of language extensions introduced in this work.

We consider a module instance as a cluster of cognitive components typically focusing on one well-defined task. The proposed framework therefore allows for the *separation of concerns*. In case a module specification encodes a solution to some recurring problem, it can be *reused* in different programs. Moreover, as we follow the idea of a module instance being a black-box with a limited public interface, the modular system serves as a useful tool for the *encapsulation* of program logic.

We have developed the modular system that combines ideas from object orientation together with the agent oriented approach. On one hand, we instantiate and manipulate module instances in a similar fashion as objects are instantiated and manipulated in object oriented languages; on the other hand, we make use of mentalistic notions such as beliefs or goals to realize the interaction between module instances. We therefore expect that the programmers acquainted with some of the main-stream programming languages will find the proposed mechanism *familiar*, but we also believe that this system will help programmers to decompose their code in a conceptually clean, *agent-oriented* manner.

9.1 Future Work

We consider our framework as a starting point for a possibly interesting follow up work. Firstly, we envision an extension to our modular system that would allow the construction of evolving, self-modifying agents. Currently, we provide means for the run time updating of the belief base and goal base of a module instance. If we introduce a set of actions for updating its belief update base, plan base and rule base, an agent would be able to construct a module instance completely from scratch. We can even imagine applying genetic programming algorithms [12] to evolve the internals of a module instance automatically.

A second interesting extension might be support for the categorisation of interfaced beliefs and goals with regards to an existing ontology. Currently, it is still possible that two module specifications assign identical name to an exported belief/goal although it represents a different concept and thus cause a name-clash. We envision a mechanism that would link each interfaced belief/goal to a particular ontology. We may employ some of the existing ontology standard, e.g. OWL, and annotate each interface entry with reference to the ontology that defines its semantics. For example like this:

```
Interface:
  vegetarianPizza(Pizza)[http://www.co-ode.org/ontologies/pizza/2007/02/12/pizza.owl]
```

That way, each interface entry would be assigned a semantic meaning and therefore it would be possible to detect shared beliefs and goals that match syntactically, but refer to different concepts. Such incompatible beliefs/goals might be distinguished e.g. using a prefixing approach similar to the one employed in [?].

Bibliography

[1] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[2] M. E. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge, MA, 1987.

[3] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Extending the capability concept for flexible BDI agent modularization. In J. Dix A. El Fallah Seghrouchni R.H. Bordini, M. Dastani, editor, *The 3rd International Workshop on Programming Multiagent Systems (PROMAS-2005), in conjunction with 4th International Joint Conference on Autonomous Agents & Multi-Agent Systems (AAMAS 2005)*, pages 139–155. Springer Verlag, Berlin, Heidelberg, 1 2006.

[4] P Busetta, R Rnnquist, A Hodgson, and A Lucas. Jack intelligent agents - components for intelligent agents in Java. 1999.

[5] Paolo Busetta, Nicholas Howden, Ralph Rönnquist, and Andrew Hodgson. Structuring BDI agents in functional clusters. In *ATAL '99: 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL),*, pages 277–289, London, UK, 2000. Springer-Verlag.

[6] Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[7] Mehdi Dastani, Christian P. Mol, and Bas R. Steunebrink. Modularity in agent programming languages, an illustration in extended 2APL. In *PRIMA '08: Proceedings of the 11th Pacific Rim International Conference on Multi-Agents*, pages 139–152, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Mehdi Dastani and Bas Steunebrink. Modularity in BDI-based multi-agent programming languages. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, 2:581–584, 2009.

[9] Koen Hindriks. Modules as policy-based intentions: Modular agent programming in goal. In *In Proceedings of the International Workshop on Programming Multi-Agent Systems (PROMAS07), number 4908 in LNAI.* Springer, 2008.

[10] Koen V. Hindriks, Frank S. de Boer, Wiebe Van Der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. In *In: Intelligent Agents VII. Agent Theories*

Architectures and Languages, 7th International Workshop, ATAL 2000. Volume 1986 of LNCS, pages 228–243. Springer, 2000.

[11] Brian W. Kernighan. *The C Programming Language.* Prentice Hall Professional Technical Reference, 1988.

[12] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[13] Peter Novák and Jürgen Dix. Modular BDI architecture. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1009–1015, New York, NY, USA, 2006. ACM.

[14] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In J. Dix R. Bordini, M. Dastani and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, pages 149–174. Springer Science+Business Media Inc., USA, 9 2005. Book chapter.

[15] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture, 1991.

[16] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[17] Y. Shoham and Others. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[18] M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. Goal-oriented modularity in agent programming. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1271–1278, New York, NY, USA, 2006. ACM.

Appendix A

Example Programs

Bomb Disposing

bombdisposing.mas

```
m : manager
w : worker 3 @blockworld
```

manager.2apl

```
BeliefUpdates:
  { true }  Bomb(POS)    { bomb(POS) }
  { true }  NotBomb(POS) { not bomb(POS) }

  { true }  Assigned(POS, A)    { assigned(POS, A) }
  { true }  NotAssigned(POS, A) { not assigned(POS, A) }

Beliefs:
  worker(w1).
  worker(w2).
  worker(w3).

PG-rules:
  <- bomb(POS) and not assigned(POS, _) and worker(W) and not assigned(_, W) | {
    send(W, request, disposed(POS));
    Assigned(POS, W)
  }

PC-rules:
  message(A, inform, _, _, bomb(POS)) <- true  |  {
    Bomb(POS)
  }

  message(A, inform, _, _, notBomb(POS)) <- true  |  {
    NotBomb(POS)
  }

  message(A, inform, _, _, disposed(POS, _)) <- true | {
    NotAssigned(POS, A)
  }
```

worker.2apl

```
Include:
  blockworld, bombsearching, disposing

Interface:
  /* blockworld module */
  at(Pos),
  entered(Pos, Color),
  randomPos(Pos, MaxX, MaxY),

  /* bombsearching module */
  bomb(Pos),
  newBombFoundHandled(Pos),
  bombDisappearedHandled(Pos),

  /* disposing module */
  disposed(Pos, Result),
  trap(Pos)

BeliefUpdates:
  {} NewBombFoundHandled(Pos)    {newBombFoundHandled(Pos)}
  {} NotNewBombFoundHandled(Pos) {not newBombFoundHandled(Pos)}

  {} BombDisappearedHandled(Pos)    {bombDisappearedHandled(Pos)}
  {} NotBombDisappearedHandled(Pos) {not bombDisappearedHandled(Pos)}

Beliefs:
  manager(m).
  trap([0,0]).

Plans:
  B(randomPos(Pos, 15, 15));
  adopta(entered(Pos, blue));
  B(entered(Pos, blue))

PG-rules:
  <- true | {
    [
      if G(at(_) or disposed(_,_)) then
        skip
      else {
        /* not G(at(_)) and not G(disposed(_)) */
        B(randomPos(Pos, 15, 15));
        adopta(at(Pos))
      }
    ]
  }

  newBombFoundHandled(Pos) <- manager(Manager) | {
    send(Manager, inform, bomb(Pos));
    NewBombFoundHandled(Pos)
  }
  <- newBombFoundHandled(Pos) | NotNewBombFoundHandled(Pos)
```

```
  bombDisappearedHandled(Pos) <- manager(Manager) | {
    send(Manager, inform, notBomb(Pos));
    BombDisappearedHandled(Pos)
  }
  <- bombDisappearedHandled(Pos) | NotBombDisappearedHandled(Pos)

PC-rules:
  message(A, request, _, _, disposed(Pos)) <- manager(A) | {
    [
    if G(at(OtherPos)) then dropgoal(at(OtherPos));
    adoptz(disposed(Pos, succeeded))
    ];

    B(disposed(Pos, DisposalResult));

    if B(DisposalResult = succeeded) then  {
      send(A, inform, disposed(Pos, succeeded))
    }
    else {
      dropgoal(dispose(Pos, succeeded));
      send(A, inform, disposed(Pos, failed))
    }
  }

PR-rules:
  /* If adoption of the at(Pos) goal fails because the agent is already at(Pos),
     ignore... */
  adopta(at(_));REST <- true | {skip;REST}
```

blockworld.2apl

```
Interface:
  /**
   * @goal
   * Agent pursues goal by inserting robot of the specified color to
   * the given position in the blockworld environment.
   *
   * @belief
   * Agent believes entered([X,Y], Color) if its blockworld robot
   * has successfully entered the blockworld environment.
   *
   * @param Pos the initial position of the agent of the form Pos = [X,Y]
   * @param Color the color of the robot
   */
  entered(Pos, Color),

  /**
   * @goal
   * Agent pursues goal by moving the robot towards the specified target
   * position.
   *
   * @belief
   * Agents's current position in the blockworld of the form Pos = [X,Y]
```

```
 *
 * @param Pos the position of the agent
 */
at(Pos),


/**
 * @goal
 * Agent pursues goal by actively sensing for bombs within the sensing
 * range in the blockworld.
 */
sensedBombs,

/**
 * @belief
 * Contains result of the last sensing of the bombs.
 *
 * @param SensePos the position on which was the sensing was performed of
 *                 the form SensePos = [X,Y]
 * @param BombList the list of sensed bombs of the form BombList = [Bomb|Tail],
 *                 where Bomb = [default, X, Y], where X, Y are the
 *                 coordinates of the bomb
 */
sensedBombs(SensePos,BombList),

/**
 * @goal
 * When the goal pickedUp(suceeded) is adopted, the robot attempts to perform
 * the pick up action in the blockworld.
 *
 * @belief
 * Contains the outcome of the last pick up attempt.
 * The belief is transitional i.e. believed only for one deliberation cycle.
 *
 * @param Result the outcome of the action, one of {suceeded, failed}
 */
pickedUp(Result),

/**
 * @goal
 * When the goal dropped(suceeded) is adopted, the robot attempts to perform
 * the drop action in the blockworld.
 *
 * @belief
 * Contains the outcome of the last drop attempt.
 * The belief is transitional i.e. believed only for one deliberation cycle.
 *
 * @param Result the outcome of the action, one of {suceeded, failed}
 */
dropped(Result),

/**
 * @belief
```

```
   * Believed if the robot in the blockworld is carrying a bomb.
   */
  carrying,

  /**
   * @belief
   * Predicate that asserts random position.
   *
   * @param Pos the random position of the form [X,Y]
   *           s.t. 0 <= X <= MaxX and 0 <= Y <= MaxY.
   * @param MaxX maximum X value
   * @param MaxX maximum Y value
   */
  randomPos(Pos, MaxX, MaxY)

BeliefUpdates:
  { at(OLDPOS) }       UpdatePosition(POS) { not at(OLDPOS), at(POS) }
  { not entered(_,_) } Entered(POS, Color) { entered(POS, Color), at(POS) }

  { sensedBombs(OP, OL) } SensedBombs(Pos, L)  { not sensedBombs(OP, OL),
                                                   sensedBombs(Pos, L) }
  { not sensedBombs(_,_) } SensedBombs(Pos, L) { sensedBombs(Pos, L) }
  { sensedBombs(Pos,L) }   NotSensedBombs()    { not sensedBombs(Pos,L) }

  { } Dropped(Result)     { dropped(Result) }
  { } NotDropped(Result)  { not dropped(Result) }

  { } PickedUp(Result)    { pickedUp(Result) }
  { } NotPickedUp(Result) { not pickedUp(Result) }

  {  }         Carrying()     { carrying }
  { carrying } NotCarrying()  { not carrying }

Beliefs:
  prob(P) :- is(X, rand), X < P.

  randomPos([X,Y], XMax, YMax) :-
    is(X, int(random(XMax))), is(Y, int(random(YMax))).

  randomDirection(north) :- prob(0.25).
  randomDirection(south) :- prob(0.5).
  randomDirection(east) :- prob(0.75).
  randomDirection(west).

PG-rules:
  entered([X,Y], Color) <- true | {
    @blockworld( enter( X, Y, Color ), _ );
    Entered([X,Y], Color)
  }

  at([X, Y]) <- at([CurrentX, CurrentY]) and X < CurrentX |  {
    [@blockworld( west(), _ );updatePosition()]
  }
```

```
  at([X, Y]) <- at([CurrentX, CurrentY]) and X > CurrentX |  {
    [@blockworld( east(), _ );updatePosition()]
  }

  at([X, Y]) <- at([CurrentX, CurrentY]) and Y < CurrentY |  {
    [@blockworld( north(), _ );updatePosition()]
  }

  at([X, Y]) <- at([CurrentX, CurrentY]) and Y > CurrentY |  {
    [@blockworld( south(), _ );updatePosition()]
  }

  sensedBombs <- true | {
    [@blockworld( senseBombs(), BOMBS );
    B(at(SensePos));
    SensedBombs(SensePos, BOMBS)];
    NotSensedBombs()
  }

  pickedUp(succeeded) <- true | {
    @blockworld( pickup(), Result );
    [Carrying();PickedUp(succeeded)]
  }
  <- pickedUp(Result) | {NotPickedUp(Result)}

  dropped(succeeded) <- true | {
    @blockworld( drop(), Result );
    [NotCarrying();Dropped(succeeded)]
  }
  <- dropped(Result) | {NotDropped(Result)}

  randomMove <- true | {[randomMove();updatePosition()]}

PC-rules:
  updatePosition() <- true | {
    @blockworld(sensePosition(), RES);
    B(RES = [actionresult(POS)]);
    UpdatePosition(POS)
  }

  randomMove() <- prob(0.25) | { @blockworld( south(), _) }
  randomMove() <- prob(0.5)  | { @blockworld( north(), _) }
  randomMove() <- prob(0.75) | { @blockworld( west(), _) }
  randomMove() <- true       | { @blockworld( east(), _)  }

PR-rules:
  @blockworld( south(), _); REST <- true | { randomMove(); REST }
  @blockworld( north(), _); REST <- true | { randomMove(); REST }
  @blockworld( west(), _);  REST <- true | { randomMove(); REST }
  @blockworld( east(), _);  REST <- true | { randomMove(); REST }

  @blockworld( pickup(), _ ); REST <- true | { PickedUp(failed)}
```

```
  @blockworld( drop(), _ );   REST <- true | { Dropped(failed) }
```

bombsearching.2apl

```
Interface:
  /**
   * @belief
   * Predicate denoting that there is a bomb on the given position. This module
   * is constantly updating the database of bombs, although it is still limited
   * by the available sensing range. It is also able to spot bombs that have
   * been removed. I.e. believed bombs that have not been confirmed during
   * subsequent sensing.
   *
   * @param Pos the position of the bomb of the form Pos = [X,Y]
   */
  bomb(Pos),

  /**
   * @goal
   * The goal adopted after a new bomb has been spotted. The goal must be
   * handled outside this module.
   *
   * @param Pos the position of the new bomb of the form Pos = [X,Y]
   */
  newBombFoundHandled(Pos),

  /**
   * @goal
   * The goal adopted after an agent realizes that one of the believed bombs
   * disappeared. The goal must be handled outside this module.
   *
   * @param Pos the position of the bomb that disappeared of the form Pos = [X,Y]
   */
  bombDisappearedHandled(Pos),

  /* module blockworld */
  at(Pos),
  sensedBombs,
  sensedBombs(_,_)

BeliefUpdates:
  { } Bomb(POS)    { bomb(POS) }
  { } NotBomb(POS) { not bomb(POS) }

Beliefs:
  /* Succeeds with the probability P; 0.0 <= P <= 1.0 */
  prob(P) :- is(X, rand), X < P.

  sensingRange(4).
  inSensingRange([AtX, AtY], [X, Y]) :-
```

```
     sensingRange(Range),
     Dist is sqrt(pow(AtX - X, 2) + pow(AtY - Y, 2)),
     Dist < Range.

Goals:
  foundAllBombs

PG-rules:
  foundAllBombs <- true |  {
    adopta(sensedBombs);

    B(sensedBombs(SensePos,Bombs));
    dropgoal(sensedBombs);

    B(Bombs = [actionresult(BombList)]);

    while B(member([default,X,Y], BombList)
            and not bomb([X,Y])) do {
       Bomb([X,Y]);

       adopta(newBombFoundHandled([X,Y]));
       B(newBombFoundHandled([X,Y]))
    };

    while B(bomb([X, Y])
            and inSensingRange(SensePos, [X, Y])
            and not member([default,X,Y], BombList)) do {
        adopta(bombDisappearedHandled([X, Y]));
        B(bombDisappearedHandled([X, Y]));
        NotBomb([X, Y])
    }
  }
```

disposing.2apl

```
Interface:
  /**
   * @goal
   * The goal disposed(Pos, suceeded) is pursued in the following way:
   * First, agent moves to the given position, performs pickup, moves towards
   * trap position and drops the bomb.
   *
   * @belief
   * Contains the outcome of the last disposal attempt.
   * The belief is transitional i.e. believed only for one deliberation cycle.
   *
   * @param Pos the position of the bomb to dispose of them form Pos = [X,Y]
   * @param Result the outcome of the disposal attempt, one of {suceeded, failed}
   */
  disposed(Pos, Result),

  /**
   * @belief
   * The position of the trap. Belief is only queried in this module and
   * therefore must be set outside this module.
```

```
   *
   * @param Pos the position of the trap of the form Pos = [X,Y]
   */
  trap(Pos),

  /* module blockworld */
  at(Pos),
  pickedUp(Result),
  dropped(Result)

BeliefUpdates:
  { true } Disposed(Pos, Result)    { disposed(Pos, Result) }
  { true } NotDisposed(Pos, Result) { not disposed(Pos, Result) }

Beliefs:
  notDisposed(Pos) :- not disposed(Pos).

PG-rules:
  disposed(Pos, succeeded) <- trap(TrapPos) | {
    adopta(at(Pos));
    B(at(Pos));

    adopta(pickedUp(succeeded));
    B(pickedUp(PickUpResult));

    if B(PickUpResult = failed) then
      [dropgoal(pickedUp(succeeded));Disposed(Pos, failed)];

    adopta(at(TrapPos));
    B(at(TrapPos));

    adopta(dropped(succeeded));
    B(dropped(DropResult));

    if B(DropResult = failed) then
      [dropgoal(dropped(succeeded));Disposed(Pos, failed)];

    Disposed(Pos, succeeded)
  }
  <- disposed(Pos, Result) | {NotDisposed(Pos, Result)}

PR-rules:
  /* If adoption of the at(Pos) goal fails because the agent is already at(Pos),
     ignore... */
  adopta(at(Pos)); REST <- at(Pos) | {skip; REST}
```