Modularity in Agent Programming Languages: An Illustration in Extended 2APL

Mehdi Dastani, Christian P. Mol, and Bas R. Steunebrink

Technical Report UU-CS-2008-022
August 2008

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Modularity in Agent Programming Languages
An Illustration in Extended 2APL

Mehdi Dastani, Christian P. Mol, and Bas R. Steunebrink

Utrecht University
The Netherlands
{mehdi,christian,bass}@cs.uu.nl

Abstract. This paper discusses a module-based vision for designing BDI-based multi-agent programming languages. The introduced concept of module is generic and facilitates the implementation of different agent concepts such as roles and agent profiles, or to adopt common programming techniques such as encapsulation and information hiding. This vision is applied to 2APL, which is an existing BDI-based agent programming language. Specific programming constructs are added to 2APL to allow the implementation of modules. The syntax and the operational semantics of these programming constructs are provided. Some informal properties of the programming constructs are discussed and it is briefly explained how these modules can be used to implement roles, agent profiles, or simply for encapsulation of task and information hiding.

1 Introduction

Modularity is an essential principle in structured programming in general and in agent programming in particular. This paper focuses on the modularity principle applied in BDI-based agent programming languages. There have been many proposals for supporting modules in BDI-based programming languages[2, 1, 5, 4]. In these proposals, modularization is considered as a mechanism to structure an individual agent's program in separate modules, each encapsulating cognitive components such as beliefs, goals, events, and plans that together can be used to handle specific situations. However, the way the modules are used in these approaches are different.

For example, in Jack[2] and Jadex[1], modules (which are also called capabilities) are employed for information hiding and reusability by encapsulating cognitive components that implement a specific capability/functionality of the agent. In these approaches, the encapsulated components are used during an agent's execution when an event should be created or when a plan should be generated to handle an event. It should be noted that Jadex extends the notion of capability by provide an import/export mechanism to connect different capabilities. In other approaches[4, 5], modules are used to realize a specific policy or mechanism in order to control nondeterminism in agent execution. In [4], modules are used to disambiguate the application and execution of plans by being considered as the 'focus of execution'. However, in [5] a module is associated with a specific goal indicating which and how planning rules should be applied to achieve that specific goal.

In these approaches, decisions such as when and how modules should be used during an agent's execution is controlled by the agent's execution strategy, usually implemented in the agent's interpreter. An agent programmer can control the use of modules during an agent's execution either in terms of the functionality of those components or through conditions assigned to the modules. For example, in Jack[2] and Jadex[1] the interpreter searches the modules in order to determine how an event can be processes. In [4, 5], belief or goal conditions are assigned to modules such that an agent's interpreter uses the modules when the respective conditions hold.

In this paper, we introduce an alternative notion of modules that, besides the aforementioned functionalities, provides an agent programmer with more control over determining how and when modules should be used. This is done by considering a module as an encapsulation of cognitive components. We propose a set of generic programming constructs that can be used by an agent programmer to perform a variety of operations on modules. Moreover, the proposed notion of module can be used to implement agent concepts

such as agent role and agent profile. In fact, in our approach a module can be used as a mechanism to specify a role that can be enacted by an agent during its execution. We also explain how the proposed notion of modules can be used to implement agents that can represent and reason about other agents.

In order to illustrate our approach we explain in the next section an extension of the agent programming language 2APL with modules. The syntax and operational semantics of the module-based 2APL are presented and sections 3 and 4, respectively. In section 5, we discuss how the proposed notion of modules can be used to implement agent roles and agent profiles. Finally, in section 6, we conclude the paper and indicate some future research directions.

2 Extending 2APL with Modules

A 2APL multi-agent program is specified in terms of a set of 2APL modules each having a unique name. Initially, a subset of these modules is identified as the specification of individual agents that constitute the implemented multi-agent system. The execution of a 2APL multi-agent program is therefore the instantiation and execution of these modules. A 2APL module can be used by another 2APL module, e.g., a module can be used by an agent. There are several operations that a module can perform on another one. These operations can be implemented by means of 2APL programming constructs designed to operate on modules.

One of these operations is to create/instantiate a module specification from the multi-agent program. One module specification can be created/instantiated more than once by one or more modules. In such a case, the creating module will assign a unique name to the module instantiation. One module can thus create several instantiations of one and the same module. Also, two modules can create two instantiations of one and the same module specification. A creating module becomes the owner of the created module. The creating module is the only module that can operate on the created module instance until the created module is released.

In addition, 2APL allows one and the same module instantiation to be used by two different modules. For this purpose, a special type of module, called singleton module, is introduced. While the ownership of a singleton module can be changed through create and release operations performed by different modules, the state of the singleton module is invariant with respect to these operations, i.e., the state of a singleton module is maintained after one module release it and another one owns it again.

The owner of a module can execute the module in two different ways. First, a module can execute another one and wait until the execution of the module is halted. A condition should then be given to indicate when the execution of a module must halt. This condition is evaluated by the overall multi-agent system interpreter and based on the internals of the executed module. Second, a 2APL module can execute another one in parallel. The executed module can be halted either by means of a condition evaluated on the internals of the executed module (by the overall multi-agent system interpreter) or explicitly by means of a stop action performed by the owner of the module. The execute operations can be used to implement 'focus of execution' and goal processing as discussed in [4] and [5], respectively.

Besides executing a module, the internals of a module can be accessed by its owner module. In particular, a module can test and update the beliefs and goals of a module that it owns. In order to control the access to the internals of a module, two types of modules are introduced: public and private. A private 2APL module can only be executed by its owner and does not allow access to its internals. In contrast to private modules, the internals of a public module are accessible to its owner module. These operations can be used to implement capabilities as discussed in [2, 1].

3 Syntax

This section presents the complete syntax of the 2APL programming constructs. As the syntax of 2APL programming language without modules are presented elsewhere[3], we here highlight and discuss only the module-related programming constructs. 2APL is a multi-agent programming language that provides programming constructs to implement both multi-agent as well as individual agent issues. The multi-agent issues are implemented by means of a specification language. Using this language, one can specify which agents should be created to participate in the multi-agent system and to which external environments each

module has access. The syntax of this specification language is presented in Figure 1 using the EBNF notation. In the following, we use ⟨*ident*⟩ to denote a string and ⟨*int*⟩ to denote an integer.

⟨*MAS_Prog*⟩ ::= "Modules :" ⟨*module*⟩$^+$
"Agents :" (⟨*agentname*⟩ ":" ⟨*moduleIdent*⟩ [⟨*int*⟩])$^+$
⟨*module*⟩ := ⟨*moduleIdent*⟩".2apl" [⟨*environments*⟩]
⟨*agentname*⟩ := ⟨*ident*⟩
⟨*moduleIdent*⟩ := ⟨*ident*⟩
⟨*environments*⟩ := "@"⟨*ident*⟩$^+$

Fig. 1. The EBNF syntax of 2APL multi-agent systems.

A correct 2APL multi-agent program should specify which modules can be instantiated during the execution of the multi-agent program. This is specified by a list of modules that is listed by the keyword `Modules`. From the set of specified modules, some will initially be instantiated as individual agents. The list of agents that initially constitute the multi-agent system is listed by the keyword `Agents`. In the specification of each agent, ⟨*agentname*⟩ is the name of the individual agent to be created, ⟨*module*⟩ is the name of the module specification that implements the agent when it is instantiated, and ⟨*int*⟩ is the number of agents that should to be created. When the number of agents is $n > 1$, then n identical agents are created. The names of these agents are ⟨*agentname*⟩ extended with a unique number. Finally, ⟨*environments*⟩ is the list of environment names to which the module(s) have access. Note that this programming language allows one to create a multi-agent system consisting of different numbers of different agents each having access to one or more environments.

Suppose we need to program a multi-agent system with a manager and a system administrator. The manager, let's call him Richard, at some point would like the system administrator to add a new user to the central computer system. The system administrator then starts the user creation module to actually do the update in the database. This multi-agent system can be implemented by the following 2APL program.

```
Modules:
  manager.2apl @clientdatabase
  admin.2apl
  userCreator.2apl @userdatabase

Agents:
  richard: manager.2apl
  administrator: admin.2apl
```

In this program, an agent named Richard is instantiated with the manager module specification and the administrator agent is instantiated based on the admin module specification. Furthermore the manager module has access to the clientdatabase and the userCreator has access to the userdatabase. The administrator does not have access to any environments on its own.

A 2APL module (that is also used to implement individual agents) is implemented by means of another specification language. The EBNF syntax of this specification language is illustrated in Figure 2. The gray parts of the syntax are not related to modules and is already presented in [3]. In this specification, we use ⟨*atom*⟩ to denote a Prolog-like atomic formula starting with lowercase letter, ⟨*Atom*⟩ to denote a Prolog-like atomic formula starting with a capital letter, ⟨*ground_atom*⟩ to denote a ground atom and ⟨*Var*⟩ to denote a string starting with a capital letter.

Although explaining the complete set of 2APL programming constructs for individual agents is not the focus of this paper, we give a brief and general idea of the basic constructs. 2APL provides programming constructs to implement an individual agent in terms of beliefs, goals, and actions. An agent's beliefs represent information the agent believes about itself and its surrounding environments, including other agents. An agent's goals represents a situation the agent wants to realize. The programming language provides different types of actions such as belief update actions (to modify beliefs), belief and goal test

actions (to query beliefs and goals), actions to adopt and drop goals, to send messages to other agents, and to perform actions in external environments. Besides these programming constructs, 2APL provides constructs to implement three types of rules. The planning goal rules (PG-rules) can be used to generate plans based on the agent's goals. The procedure call rules (PC-rules) can be used to generate plans for the received internal and external events. Finally, the plan repair rules (PR-rules) can be used to repair a plan whose execution has failed.

In this paper, we only explain 2APL programming constructs that are related to modules. The first construct related to modules is the use of keywords `public`/`private` and `singleton`. A module characterized as public allows its owner to execute it and access its internals, while a module characterized as private only allows itself to be executed by its owner. Note that modules for initial agents can be either public or private as their owner (i.e., the multi-agent system interpreter) can only execute them.

The `create(mod-name, mod-ident)` action can be used to instantiate the module specification with the name `mod-name`. The name that is assigned to the created instantiation of the module is given by the second argument `mod-ident`. The owner of a module instantiation can use this name to perform further operations on it. A module instantiation `m` can be released by means of the `release(m)` action. If the module is not a singleton, then its instantiation will be removed/lost. However, if the module is a singleton, then its instantiation will be maintained in the multi-agent system such that it can be used by another module using the create action. It is important to note that a singleton module can only have one instantiation at a time such that it can always be accessed by means of the module name `mod-name`. It is also important to note that the subsequent creation of a singleton module by another module, which may be assigned a different name, will refer to the same instantiation of the module as when it was released by its last owner.

When a public or private module `m` is created/instantiated, it can be executed by its owner through the action `m.execute(`$\langle test \rangle$`)` or `m.executeasync(`$\langle test \rangle$`?)`. The execution of a module instantiation by means of `execute` action starts the 2APL deliberation process based on the internals of the module instantiation. The execution of the owning module halts until the execution of the owned module halts and a return event is received from the owned module. In order to notify a module that it should stop its execution,[1] the test condition (i.e., the argument of the execute action) is evaluated by the overall multi-agent system interpreter and a stop event `stop!` is sent to the module. The module that receives a stop event may start a cleaning operation and send a return event back when it is ready. We thus introduce an action `return` that can be executed by an owned module as the last action after which the execution of this module is halted. The execution of this action broadcasts an event `return!` that can be received by the overall multi-agent system interpreter after which the owning module execution is started and notified about the return event from the owned module. After the reception of this event, the owning module's deliberation process is continued, after which it may decide to release the owned module.

The execution of a module instantiation by means of the `executeasync` action is identical to `execute`, except that the owner does not have to wait until the execution of the module halts. In fact, the owner continues with its own execution in parallel with the execution of the owned module instantiation. The execution of the module instantiation can be halted by providing a test expression as argument of the `executeasync` action, or by the owning module performing the `stop` action on the module instantiation. Like the `execute` action, the test will be evaluated at the multi-agent system level and based on the internals of the module instantiation. The `stop` action performed by the owning module will send event `stop!` to the owned module.

The owner of a public module instance can access and update the internals of the module instance. In particular, a module can test whether certain beliefs and goals are entailed by the beliefs and goals of a public owned module instance `m` through action `m.B(`φ`) & G(`ψ`)`. Also, the beliefs of a module instance `m` can be updated by means of `m.updateBB(`φ`)` action. A goal can be added to the goals of a module instance `m` by means of `m.adopta(`φ`)` and `m.adoptz(`φ`)` actions. Finally, the goals of a module instance `m` can be dropped by means of `m.dropgoal(`φ`)`, `m.dropsubgoals(`φ`)` and `m.dropsupergoals(`φ`)` actions. As explained in [3], these actions can be used to drop from an agent's goal base, respectively, all goals identical to φ, all goals that are a logical subgoal of φ, and all goals that have φ as a logical subgoal.

Given our previous example, the actual code implementing the administrator may include a plan expression of the following form:

[1] The owning module cannot do this because its execution is halted.

⟨2*APL_Prog*⟩ ::= (”private” | ”public”) ”singleton”?
(”Include:” ⟨*ident*⟩
| ”BeliefUpdates:” ⟨*BelUpSpec*⟩
| ”Beliefs:” ⟨*belief*⟩
| ”Goals:” ⟨*goals*⟩
| ”Plans:” ⟨*plans*⟩
| ”PG-rules:” ⟨*pgrules*⟩
| ”PC-rules:” ⟨*pcrules*⟩
| ”PR-rules:” ⟨*prrules*⟩)*

⟨*BelUpSpec*⟩ ::= (”{”⟨*belquery*⟩ ”}” ⟨*beliefupdate*⟩ ”{”⟨*literals*⟩”}”)+

⟨*belief*⟩ ::= (⟨*ground_atom*⟩ ”.” | ⟨*atom*⟩ ”: −” ⟨*literals*⟩”.”)+

⟨*goals*⟩ ::= ⟨*goal*⟩ (”,” ⟨*goal*⟩)*

⟨*goal*⟩ ::= ⟨*ground_atom*⟩ (”and” ⟨*ground_atom*⟩)*

⟨*baction*⟩ ::= ”skip” | ⟨*beliefupdate*⟩ | ⟨*sendaction*⟩ | ⟨*externalaction*⟩
| ⟨*abstractaction*⟩ | ⟨*test*⟩ | ⟨*adoptgoal*⟩ | ⟨*dropgoal*⟩
| ⟨*createaction*⟩ | ⟨*releaseaction*⟩ | ⟨*return*⟩ | ⟨*moduleaction*⟩

⟨*createaction*⟩ ::= ”create(” ⟨*ident*⟩ ”,” ⟨*ident*⟩ ”)”

⟨*releaseaction*⟩ ::= ”release(” ⟨*ident*⟩ ”)”

⟨*return*⟩ ::= ”return”

⟨*moduleaction*⟩ ::= ⟨*ident*⟩ ”.” ⟨*maction*⟩

⟨*maction*⟩ ::= ”execute(” ⟨*test*⟩ ”)” | ”executeasync(” ⟨*test*⟩? ”)”
| ”stop” | ⟨*test*⟩ | ⟨*adoptgoal*⟩ | ⟨*dropgoal*⟩ | ⟨*updBB*⟩

⟨*updBB*⟩ ::= ”updateBB(” ⟨*literals*⟩ ”)”

⟨*plans*⟩ ::= ⟨*plan*⟩ (”,” ⟨*plan*⟩)*

⟨*plan*⟩ ::= ⟨*baction*⟩ | ⟨*sequenceplan*⟩ | ⟨*ifplan*⟩ | ⟨*whileplan*⟩ | ⟨*atomicplan*⟩
| ⟨*mifplan*⟩ | ⟨*mwhileplan*⟩

⟨*beliefupdate*⟩ ::= ⟨*Atom*⟩

⟨*sendaction*⟩ ::= ”send(” ⟨*iv*⟩ ”,” ⟨*iv*⟩ ”,” ⟨*atom*⟩ ”)”
| ”send(” ⟨*iv*⟩ ”,” ⟨*iv*⟩ ”,” ⟨*iv*⟩ ”,” ⟨*iv*⟩ ”,” ⟨*atom*⟩ ”)”

⟨*externalaction*⟩ ::= ”@” ⟨*iv*⟩”(” ⟨*atom*⟩ ”,” ⟨*Var*⟩ ”)”

⟨*abstractaction*⟩ ::= ⟨*atom*⟩

⟨*test*⟩ ::= ”B(” ⟨*belquery*⟩ ”)” | ”G(” ⟨*goalquery*⟩ ”)” | ⟨*test*⟩ ”&” ⟨*test*⟩

⟨*adoptgoal*⟩ ::= ”adopta(” ⟨*goalvar*⟩ ”)” | ”adoptz(” ⟨*goalvar*⟩ ”)”

⟨*dropgoal*⟩ ::= ”dropgoal(” ⟨*goalvar*⟩ ”)” | ”dropsubgoals(” ⟨*goalvar*⟩ ”)”
| ”dropsupergoals(” ⟨*goalvar*⟩ ”)”

⟨*sequenceplan*⟩ ::= ⟨*plan*⟩ ”;” ⟨*plan*⟩

⟨*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” ⟨*scopeplanvar*⟩)?
| ”while” ⟨*test*⟩ ”do” ⟨*scopeplanvar*⟩ | ⟨*planvar*⟩ ”;” ⟨*planvar*⟩

⟨*mifplan*⟩ ::= ”if” ⟨*ident*⟩”.”⟨*test*⟩ ”then” ⟨*scopeplan*⟩ (”else” ⟨*scopeplan*⟩)?

⟨*mwhileplan*⟩ ::= ”while” ⟨*ident*⟩”.”⟨*test*⟩ ”do” ⟨*scopeplan*⟩

⟨*scopeplanvar*⟩ ::= ”{” ⟨*planvar*⟩ ”}”

⟨*literals*⟩ ::= ⟨*literal*⟩ (”,” ⟨*literal*⟩)*

⟨*literal*⟩ ::= ⟨*atom*⟩ | ”not” ⟨*atom*⟩

⟨*ground_literal*⟩ ::= ⟨*ground_atom*⟩ | ”not” ⟨*ground_atom*⟩

⟨*belquery*⟩ ::= ”true” | ⟨*belquery*⟩ ”and” ⟨*belquery*⟩ | ⟨*belquery*⟩ ”or” ⟨*belquery*⟩
| ”(” ⟨*belquery*⟩ ”)” | ⟨*literal*⟩

⟨*goalquery*⟩ ::= ”true” | ⟨*goalquery*⟩ ”and” ⟨*goalquery*⟩ | ⟨*goalquery*⟩ ”or” ⟨*goalquery*⟩
| ”(” ⟨*goalquery*⟩ ”)” | ⟨*atom*⟩

⟨*iv*⟩ ::= ⟨*ident*⟩ | ⟨*Var*⟩

Fig. 2. The EBNF syntax of 2APL extended with modules.

```
{
  create(userCreator, u);
  u.updateBB(user(dave, hopkins));
  u.adopta(registered(dave));
  u.execute( B(registered(dave)) );
  release(u)
}
```

Here we see that an instantiation of the userCreator is made and named `u`. The beliefbase of this module instantiation is then updated with some information of the new user to be registered. Next a goal is added to have the user registered in the database after which the instantiation is executed. The stop condition is set to `B(registered(dave))`. This should be entailed by the belief base of the module instantiation when the goal is reached, i.e. when the user information is written to the database. The module definition of the userCreator could be specified as follows:

```
public

BeliefUpdates:
  { true } AddUser(FirstName) { registered(FirstName) }

PG-Rules:
  registered(FirstName) <- user(FirstName, LastName) |
  {
    @userdatabase(adduser(FirstName, LastName));
    AddUser(FirstName)
  }

PC-Rules:
  event(stop) <- true | return
```

The module is declared public so that we can add information to the beliefbase and the goalbase. We also have a PG-rule that implements the functionality of the module in a very basic way. It executes an action on the "userdatabase" environment (assuming that this will succeed) and then sets the belief that makes the goal reached. Because this is also the stopping condition specified by the administrator agent, a stop event will be sent to this module. The PC-rule can then be applied to perform the action `return` through which the execution control is returned to the administrator agent. This then releases the instantiation of the userCreator. The actual user data is stored in the user database persistently.

We could have chosen to make the `userCreator` a singleton. The instantiation that is created when `create(userCreator, u)` is executed for the first time, would then not be deleted by executing `release(u)`. Instead the singleton module will persist in the multi agent configuration. The next time a create action is executed for this module, this very same instance is used. This could impose the problem that the administrator agent does not know beforehand what the state of the module is. For instance, when a user fact is already present in the beliefbase of the module, executing the example could result in the addition of two users in the database.

4 Semantics

The semantics of 2APL is defined in terms of a transition system, which consists of a set of transition rules for deriving transitions. A transition specifies a single computation/execution step by indicating how one configuration can be transformed into another. In this paper, we first present the multi-agent system configuration, which consists of the configurations of individual agents and the state of the external shared environments. Then, we present transition rules from which possible execution steps (i.e. transitions) for multi-agent systems can be derived. Here, we do neither present the configuration nor the transitions rules

for individual agents. Elsewhere[3] we have presented the semantics of 2APL *without modules*. In this section, we focus on the semantics of 2APL programming constructs that are designed to implement modules. As the execution of these programming constructs affect mainly the multi-agent system configuration, we do not present the individual agent configuration and their transition rules. The only effect of the module related programming constructs at the individual agent level is that they are removed from the agent's plan base upon execution. It is important to note that individual agent *transitions* are used as conditions of the multi-agent system *transition rules*.

4.1 2APL Multi-Agent System Configuration

The configuration of a multi-agent systems is defined in terms of the configuration of modules (agents) and the state of their shared external environments. The configuration of a module includes (1) an instance of the module (beliefs, goals, plans, events, and reasoning rules) with a unique name, (2) the name of the (parent) module that has created the module instance, (3) the identifier of the module specification (the module instance is an instantiation of this specification[2]), (4) a flag indicating if the module instance is executing, and (5) the stopping condition for the module instance. Finally, the state of a shared environment is a set of facts that hold in that environment.

Definition 1 (multi-agent system configuration). *Let (A_i, p, r, e, φ) be a module configuration, where A_i is a module instance with the unique name i, p is the name of the owner of the module instance, r is an identifier referring to the module specification, e is the execution flag, and φ is the execution stop condition. Let $\mathcal{A}$ be a set of module configurations and χ be a set of external shared environments each of which is a consistent set of atoms $\langle atom \rangle$. The configuration of a 2APL multi-agents system is defined as $\langle \mathcal{A}, \chi \rangle$.*

The initial configuration of a multi-agent system consists of the initial configuration of its individual agents and the initial state of the shared external environments as specified in the multi-agent program. The initial configuration of each individual agent is determined by the 2APL module that is assigned to the agent in the multi-agent program. The initial state of the shared external environment is set by the programmer, e.g., the programmer may initially place gold or trash at certain positions in a blockworld environment. In particular, for each individual agent implemented as $(i : m@env_1 \dots env_k, N)$ in the multi-agent program, N agent instantiations $(A_{i_1}, \texttt{mas}, m, \mathbf{t}, \bot), \dots, (A_{i_N}, \texttt{mas}, m, \mathbf{t}, \bot)$ are created and added to the set of module instantiations $\mathcal{A}$. Also, all environments env_j from the multi-agent system program are collected in the set χ. Thus, modules created when the multi-agent program is started must have `mas` as parent, $\mathbf{t}$ as execution flag, and $\bot$ as stopping condition.

The execution of a 2APL multi-agent program modifies its initial configuration by means of transitions that are derivable from the transition rules presented in the following subsection. In fact, each transition rule indicates which execution step (i.e., transition) is possible from a given configuration. It should be noted that for a given configuration there may be several transition rules applicable. An interpreter is a deterministic choice of applying transition rules in a certain order. Before we present the transition rule, we will define the following auxiliary functions.

$children_{\mathcal{A}}(i) = \{ (A_j, k, r, e, \varphi) \in \mathcal{A} \mid k = i \}$,

$descendants(i) = children(i) \cup \{ d \in descendants(j) \mid (A_j, k, r, e, \varphi) \in children(i) \}$,

$active\_descendants(i) = \{ (A_j, k, r, e, \varphi) \in descendants(i) \mid e = \mathbf{t} \}$,

$saved\_descendants(i) = \{ (A_r, \_, r, \mathbf{f}, \bot) \mid (A_j, k, r, e, \varphi) \in descendants(i),\ singleton(r) \}$,

$$saved\_singleton(A_i, r) = \begin{cases} \{(A_r, \_, r, \mathbf{f}, \bot)\} & \text{if } singleton(r) \\ \emptyset & \text{otherwise.} \end{cases}$$

In the above functions, $singleton(r)$ is valid if and only if the module r is a singleton module. Also, the underscore sign _ is used to indicate that the instantiation of a singleton module is not owned by any module.

[2] Note that there may be several instances of a module specification in a multi-agent system.

4.2 Transition Rules for Module Actions

In the following, we provide the transition rules for deriving a multi-agent system transition based on the execution of a module related action by one of the involved module instantiations. We will use $A_i \xrightarrow{\alpha!} A_i'$ to indicate that the module instantiation A_i can make a transition to module instantiation A_i' by performing action α and *broadcasting* event $\alpha!$. When $\alpha?$ is used, instead of $\alpha!$, the agent performs α and *receives* the event $\alpha?$. We also write $A_i \models \varphi$ to indicate that the test formula φ holds in the module instance A_i. Without defining the entailment relation formally, we note that a test formula holds in a module instance if it holds in the beliefs and goals of the module instance.

The first transition indicates the effect of the `create`(r, n) action performed by the module A_i, where r is the identifier of a non-singleton module specification and n is the name that will be associated to the created module instance. The module instance $A_{i.n}$ that should be created is assumed to be a non-singleton module (i.e., $\neg singleton(r)$). This transition rule indicates that a non-singleton module instance can be created by another module instance if the creating module is in the execution mode (the execution flag equals $\mathbf{t}$ and $A_i \not\models \varphi$) and there is no module instance with the same name already created by the same module ($\neg\exists r'', e, \varphi' : (A_{i.n}, i, r'', e, \varphi') \in \mathcal{A}$). The result is that the set of modules $\mathcal{A}$ is modified and extended. In particular, the creating module instance is modified as it has performed the `create` action and the newly created module instance is added to the multi-agent system configuration. Note that the newly created module is not in execution mode and its execution stopping condition is set to $\bot$.

$$\frac{(A_i, p, r', \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \xrightarrow{\texttt{create}(r,n)!} A_i' \;\&\; \neg singleton(r) \;\&\; \neg\exists r'', e, \varphi' : (A_{i.n}, i, r'', e, \varphi') \in \mathcal{A}}{\langle \mathcal{A}, \chi\rangle \longrightarrow \langle \mathcal{A}', \chi\rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r', \mathbf{t}, \varphi)\}) \cup \{(A_i', p, r', \mathbf{t}, \varphi), (A_{i.n}, i, r, \mathbf{f}, \bot)\}$.

It should be noted that a module is only allowed to create another module twice (or more) if the module to be created is not singleton and different names are used to identify it. This will result in two different instantiations of the module, each with its own name and state.

The next transition rule is to specify the creation of a singleton module. As noted, a singleton module can be instantiated only once and its state will be maintained once it is created. An attempt to create an instance of a singleton module specification r is only successful if either there is no instance of r in the multi-agent system configuration or there exists one instance of r which is not owned by any another module ($\neg\exists k \neq \_, n', e, \varphi' : (A_{n'}, k, r, e, \varphi') \in \mathcal{A}$). Note that a singleton module instance that is not owned by any other module has underscore $\_$ as the name of its parent module. The effect of creating a singleton module can be determined by distinguishing two cases. If there exists already an instance of the singleton module specification r that is not owned by any module, then the creating module will become the owner of the existing module instance. No additional module instance is created in the multi-agent system configuration. However, if there is no instantiation of the singleton module specification r in the configuration, then the module instance is created and added to the multi-agent system configuration (as it was the case with the creation of non-singleton modules).

$$\frac{(A_i, p, r', \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \xrightarrow{\texttt{create}(r,n)!} A_i' \;\&\; singleton(r) \;\&\; \neg\exists k \neq \_, n', e, \varphi' : (A_{n'}, k, r, e, \varphi') \in \mathcal{A}}{\langle \mathcal{A}, \chi\rangle \longrightarrow \langle \mathcal{A}', \chi\rangle}$$

where

$$\mathcal{A}' = \begin{cases} (\mathcal{A} \setminus \{(A_i, p, r', \mathbf{t}, \varphi), (A_r, \_, r, \mathbf{f}, \bot)\}) & \\ \quad \cup \{(A_i', p, r', \mathbf{t}, \varphi), (A_{i.n}, i, r, \mathbf{f}, \bot)\} & \text{if } (A_r, \_, r, \mathbf{f}, \bot) \in \mathcal{A} \\ (\mathcal{A} \setminus \{(A_i, p, r', \mathbf{t}, \varphi)\}) \cup \{(A_i', p, r', \mathbf{t}, \varphi), (A_{i.n}, i, r, \mathbf{f}, \bot)\} & \text{otherwise.} \end{cases}$$

It should be noted that a module is not allowed to create a new module with the same name as a module it has under control already. This would otherwise result in a name clash in the controlling module.

A module A_i that owns another module named n ($\exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A}$) can release it, provided this owned module is not currently in an executing state (i.e. its execution flag is $\mathbf{f}$) and the same for all modules

owned by $A_{i.n}$ etc. ($active\_descendants(i.n) = \emptyset$). It can do this by performing the action `release`(n). If the released module is non-singleton, it will simply be removed from the set of module configurations $\mathcal{A}$. If a module releases a singleton module, the released module is kept in the set of module configurations with an underscore owner and with the module specification identifier as its module instance name (i.e. its configuration becomes $(A_{r'}, \_, r', \mathbf{f}, \bot)$). Of course, the same holds for all modules owned by $A_{i.n}$ that are singleton.

$$\frac{(A_i, p, r, \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \overset{\texttt{release}(n)!}{\longrightarrow} A_i' \;\&\; \exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A} \;\&\; active\_descendants(i.n) = \emptyset}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi), (A_{i.n}, i, r', \mathbf{f}, \bot)\} \setminus descendants(i.n))$
$\cup \{(A_i', p, r, \mathbf{t}, \varphi)\} \cup saved\_singleton(A_{i.n}, r') \cup saved\_descendants(i.n)$.

It should be noted that a non-singleton module is always created privately for the creating agent. Therefore, such a module will not retain its state when it is released and created again (only a singleton module can be used for this purpose). Also, the creating agent is the only one that can release and thereby delete the module. On the other hand, when creating a singleton module, the creating agent acquires a lock on the created module until it releases it. This may seem counterintuitive given the notion of 'singleton' in an object-oriented programming language, where access to the same instance is provided to more than one object. However we feel that, in order to allow the module to keep itself in a consistent state, this is an acceptable tradeoff.

A module that owns another module can execute it, meaning that the owned module's execution flag is set to **t** so that it can perform actions by itself. In doing so, the owning module's execution flag is set to **f**. In effect, control is 'handed over' from the owner module to the owned module. As part of the `execute` action, a stopping condition φ is provided with which the owner module can specify when it wants control returned, i.e., as soon as the owned module satisfies the stopping condition ($A_{i.n} \models \varphi$; a transition rule for this case is provided later on).

$$\frac{(A_i, p, r, \mathbf{t}, \varphi') \in \mathcal{A} \;\&\; A_i \not\models \varphi' \;\&\; A_i \overset{\texttt{n.execute}(\varphi)!}{\longrightarrow} A_i' \;\&\; \exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A} \;\&\; active\_descendants(i.n) = \emptyset}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi'), (A_{i.n}, i, r', \mathbf{f}, \bot)\}) \cup \{(A_i', p, r, \mathbf{f}, \varphi'), (A_{i.n}, i, r', \mathbf{t}, \varphi)\}$.

Alternatively, a module can execute another module that it owns asynchronously using the `executeasync` action. This action works just as `execute` above, the only difference being that the execution flag of the owning module remains at **t**. In effect, both modules will be running simultaneously from that point on. The transition rule below is therefore almost equal to the one above.

$$\frac{(A_i, p, r, \mathbf{t}, \varphi') \in \mathcal{A} \;\&\; A_i \not\models \varphi' \;\&\; A_i \overset{\texttt{n.executeasync}(\varphi)!}{\longrightarrow} A_i' \;\&\; \exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A} \;\&\; active\_descendants(i.n) = \emptyset}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi'), (A_{i.n}, i, r', \mathbf{f}, \bot)\}) \cup \{(A_i', p, r, \mathbf{t}, \varphi'), (A_{i.n}, i, r', \mathbf{t}, \varphi)\}$.

It should be noted that executing a module twice (or more) is not allowed. If a module is running, either by means of `execute` or `executeasync`, no other execute statement can be performed on that particular module, nor can any tests or updates be performed on it.

As soon as the stopping condition of an executing module holds ($A_i \models \varphi$), it will receive a *stop* event from multi-agent level requesting it to stop its execution, possibly after first performing some cleanup operations. Note that it is assumed that a module is always able to receive a *stop* event ($A_i \overset{stop?}{\longrightarrow} A_i'$). It is not guaranteed by the system that a module will actually ever stop; it must perform a `return` action (see below) itself in order to have its execution flag set to **f**.

$$\frac{(A_i, p, r, \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \models \varphi \;\&\; A_i \xrightarrow{stop?} A_i'}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi)\}) \cup \{(A_i', p, r, \mathbf{t}, \varphi)\}$.

A module A_i that owns another module named n ($\exists r', e, \varphi' : (A_{i.n}, i, r', e, \varphi') \in \mathcal{A}$) can request $A_{i.n}$ to stop its execution by performing a `stop` action. The owned module then receives a *stop* event ($A_{i.n} \xrightarrow{stop?} A_{i.n}'$), but module $A_{i.n}$ is allowed to perform cleanup operations before performing a `return` action (see below). However, note that although it is assumed the owned module will always successfully receive the *stop* event, it is not guaranteed by the system that the owned module will (ever) act on it. So the transition rule below merely expresses that the owned module is updated with the *stop* event.

$$\frac{\begin{array}{c}(A_i, p, r, \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \xrightarrow{\texttt{n.stop!}} A_i' \;\&\; \exists r', e, \varphi' : (A_{i.n}, i, r', e, \varphi') \in \mathcal{A} \;\&\; \\ A_{i.n} \xrightarrow{stop?} A_{i.n}'\end{array}}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi), (A_{i.n}, i, r', e, \varphi')\}) \cup \{(A_i', p, r, \mathbf{t}, \varphi), (A_{i.n}', i, r', e, \varphi')\}$.

A module can return control to its parent module by performing a `return` action. This will cause the execution flag of the parent module to be set to $\mathbf{t}$ (although it may have been $\mathbf{t}$ already, if both were running asynchronously), while the execution flag of the module itself is set to $\mathbf{f}$. If the module performing a `return` action does not have a parent module, it is simply removed from the set of module configurations, or retained with a blank owner if it is singleton. It is up to the programmer to ensure that a `return` action is performed by a module in response to a *stop* event.

$$\frac{(A_i, p, r, \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \xrightarrow{\texttt{return!}} A_i'}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where

$$\mathcal{A}' = \begin{cases} (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi), (A_p, p', r', e, \varphi')\}) & \\ \quad \cup \{(A_i', p, r, \mathbf{f}, \bot), (A_p, p', r', \mathbf{t}, \varphi')\} & \text{if } \exists p', r', e, \varphi' : (A_p, p', r', e, \varphi') \in \mathcal{A} \\ (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi)\}) \cup \{(A_r, \_, r, \mathbf{f}, \bot)\} & \text{if } \mathit{singleton}(r) \\ \mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi)\} & \text{otherwise.} \end{cases}$$

It should be noted that a module's execution has to be finished before it can be released. This can be done either by the `stop` statement, in case of `executeasync`, or by waiting for the running module to reach its stopping condition, in case of `execute`.

Also, note that singleton modules that are created will never be removed during the lifetime of the multi-agent system. A singleton module is persistent, i.e., it will keep its internal state. Another agent that acquires this singleton module will get it in the state it was in when it was released by its last owner. Of course, in case the singleton module had not been created before, it will be created new.

Next we consider several actions that a module can perform on a module that it owns that do not pertain to control, but to the state of the owned module. Specifically, a module can query the beliefs and goals of an owned module, update the beliefs of an owned module, and adopt and drop goals in an owned module. However, these actions are only allowed on modules that contain the *public* flag in their specification; indeed, the internal state of modules specified with the *private* flag remains inaccessible to any module that becomes to own it. First we consider the belief and goal queries. These may be combined as usual; for example, the test `bas.B(p(X)) & G(q(Y))` succeeds if `p(X)` can be derived from `bas`'s beliefs and `q(Y)` from `bas`'s goals, yielding a substitution for `X` and `Y`. Thus a module A_i that owns another module named n ($\exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A}$) which is currently inactive (including all of $A_{i.n}$'s descendants, i.e. $\mathit{active\_descendants}(i.n) = \emptyset$) can perform a belief/goal query $t(\bar{x})$ on $A_{i.n}$ if the module $A_{i.n}$ was specified as being public ($\mathit{public}(r')$). If the query succeeds ($A_{i.n} \models t(\bar{x})\tau$), it yields a substitution τ which is concatenated to A_i''s substitution θ. The following transition rule captures this.

$$\frac{(A_i, p, r, \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \xrightarrow{n.t(\bar{x})!} A_i' \;\&\; \exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A} \;\&\; \textit{active\_descendants}(i.n) = \emptyset \;\&\; \textit{public}(r') \;\&\; A_{i.n} \models t(\bar{x})\tau}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi)\}) \cup \{(A_i'', p, r, \mathbf{t}, \varphi)\}$,
$A_i' = \langle i, \sigma, \gamma, \Pi, \theta, \xi \rangle$, and $A_i'' = \langle i, \sigma, \gamma, \Pi, \theta\tau, \xi \rangle$. A detailed explanation and definition of individual agent configuration $\langle i, \sigma, \gamma, \Pi, \theta, \xi \rangle$ can be found in [3].

Next we consider belief updates. It is assumed that a formula φ can represent a belief update and that $\sigma \oplus \varphi$ yields a belief base which is σ updated with φ. Note that if φ contains any negated terms, these will be deleted from σ. Similar to the transition rule for queries above, the owned module on which the belief update is performed must not be active and must have been specified as *public*.

$$\frac{(A_i, p, r, \mathbf{t}, \varphi') \in \mathcal{A} \;\&\; A_i \not\models \varphi' \;\&\; A_i \xrightarrow{n.\texttt{updateBB}(\varphi)!} A_i' \;\&\; \exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A} \;\&\; \textit{active\_descendants}(i.n) = \emptyset \;\&\; \textit{public}(r')}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi'), (A_{i.n}, i, r', \mathbf{f}, \bot)\}) \cup \{(A_i', p, r, \mathbf{t}, \varphi'), (A_{i.n}', i, r', \mathbf{f}, \bot)\}$,
$A_{i.n} = \langle i.n, \sigma, \gamma, \Pi, \theta, \xi \rangle$, and $A_{i.n}' = \langle i.n, \sigma \oplus \varphi, \gamma, \Pi, \theta, \xi \rangle$.

To make an owned module named n adopt a goal formula κ, the action $n.\texttt{adoptgoal}(\kappa)$ action can be used; similarly, to make it drop a goal formula κ, the action $n.\texttt{dropgoal}(\kappa)$ can be used. Both adopting and dropping of a goal are expressed in the same transition rule below, because they are almost equal. This transition rule is also mostly analogous to the one for belief updates above.

$$\frac{(A_i, p, r, \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \xrightarrow{n.\texttt{adopt/dropgoal}(\kappa)!} A_i' \;\&\; \exists r' : (A_{i.n}, i, r', \mathbf{f}, \bot) \in \mathcal{A} \;\&\; \textit{active\_descendants}(i.n) = \emptyset \;\&\; \textit{public}(r')}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi), (A_{i.n}, i, r', \mathbf{f}, \bot)\}) \cup \{(A_i', p, r, \mathbf{t}, \varphi), (A_{i.n}', i, r', \mathbf{f}, \bot)\}$,
$A_{i.n} = \langle i.n, \sigma, \gamma, \Pi, \theta, \xi \rangle$, and $A_{i.n}' = \langle i.n, \sigma, \gamma \oplus/\ominus \kappa, \Pi, \theta, \xi \rangle$.

Finally, a general transition rule is needed for all actions α not equal to one of the newly introduced actions above (i.e. 'classical' 2APL actions). Otherwise the transitions of these α would be undefined at the multi-agent level because of the changes that have been introduced to the module configurations. The transition rule below expresses that only modules that have $\mathbf{t}$ as their execution flag and a non-satisfied stopping condition φ are allowed to execute a 'classical' 2APL action α. Note that the execution of α possibly leads to a change in the environment χ (as expressed by the subscript χ').

$$\frac{(A_i, p, r, \mathbf{t}, \varphi) \in \mathcal{A} \;\&\; A_i \not\models \varphi \;\&\; A_i \xrightarrow{\alpha!}_{\chi'} A_i'}{\langle \mathcal{A}, \chi \rangle \longrightarrow \langle \mathcal{A}', \chi' \rangle}$$

where $\mathcal{A}' = (\mathcal{A} \setminus \{(A_i, p, r, \mathbf{t}, \varphi), \}) \cup \{(A_i', p, r, \mathbf{t}, \varphi)\}$.

5 Roles, Profiles and Task Encapsulation

The proposed extension to 2APL is general enough to be useful for the implementation of several agent-oriented programming topics. These include taking on different roles, making profiles of other agents, and the general programming technique of task encapsulation. We will provide an example for each of these topics in the following subsections.

5.1 Roles

The run-time creation of new modules can be used to implement roles, if the newly created module is seen as a role that can be played by the creating agent. The file that is used to create the new module is then a

specification of the role that can be played. The action `create`(*role*, *name*) can be seen as the activation of a role, by which the activating agent acquires a lock on the activated role, i.e. it becomes the role's owner and gains the exclusive right to manipulate the activated role. If *role* has been declared as *singleton*, this property of locking is important, because other agents may attempt the acquire the *role* as well. If *role* is not singleton, the role is created new and private to the creating agent anyway. Upon releasing a singleton role, the role is not deleted but retained with a blank owner, so that another agent may activate (using `create`(*role*, *name*′)) and use it.

An agent that has successfully performed the action `create`(*role*, *name*) is the owner of *role* and may *enact* this role using *name*.`execute`(φ), where φ is a stopping condition, i.e. a composition of belief and goal queries. The owner agent is then put on hold until the role satisfies the stopping condition, at which point control is returned to the owner agent. Alternatively, the role may be executed in parallel with the owner agent using *name*.`executeasync`(φ), meaning that a new thread is created for *role* to run in. Note that supplying as stopping condition $\varphi = \bot$ means that the role can only be stopped by executing *name*.`stop`, which of course is only possible if the role was enacted using `executeasync`.

In principle, it is allowed for a role to activate and enact a new role, and repeat this without (theoretical) depth limits. However, this is usually not allowed in literature on roles. But it is up to the programmer to prevent roles from enacting other roles.

5.2 Agent Profiles

An agent can easily create and maintain profiles of other agents by creating non-singleton modules. For example, assume agent *bas* executes the actions `create`(*profile_template*, *chris*) and `create`(*profile_template*, *mehdi*), i.e., it uses a single template (specified as being *public*) to initialize profiles of the (hypothetical) agents *chris* and *mehdi*. These profiles can be updated by *bas* using e.g. *chris*.`updateBB`(φ) and *mehdi*.`adoptgoal`(κ) when appropriate. *bas* can even 'wonder' what *chris* would do in a certain situation by setting up that situation using belief and goal updates on *chris* and then performing *chris*.`execute`(φ) (or `executeasync`) with a suitable stopping condition φ. The resulting state of *chris* can be queried afterwards to determine what *chris* 'would have done'.

5.3 Task Encapsulation

Modules can also be used for the common programming techniques of encapsulation and information hiding. Modules can encapsulate certain tasks, which can be performed by its owning agent if it performs an `execute` action on that module. Moreover, a module that has been declared to be *private* cannot be modified (e.g. by `updateBB`) by its owning agent. Such a module can thus hide its internal state and keep it consistent for its task(s). An important difference between *creating* a module (in the sense proposed here) and *including* a module (in the sense of [3]) is that the contents of an included module are simply added to the including agent, whereas the contents of a created module are kept in a separate scope. So when using the `create` action, there can be no (inadvertent) clashes caused by equal names being used in different files for beliefs, goals, actions, and rules.

6 Conclusions and Future Work

In this paper we have introduced a mechanism for creating modules in BDI-based agent programming languages. We have illustrated this mechanism by extending the operational semantics of 2APL with transition rules for actions that allow modules to be created, executed, queried, modified, and to be released again. Each module is a first-class agent that is itself allowed to create other modules, and so on, up to a (theoretically) unlimited depth. Furthermore, by using the *public*/*private* and *singleton* flags in the specification of a module, the programmer can use these modules for common programming techniques such as data hiding and singleton access. We have also shown how modules can be used to facilitate the implementation of notions relevant to agent programming; namely, the implementation of agent roles and agent profiles. We intend to provide an proof of concept of the proposed extension by implementing the presented operational semantics in the current 2APL platform.

For future work, there are several extensions to this work on modularization that can make it more powerful for encapsulation and implementation of roles and agent profiles. Firstly, the `execute` and `executeasync` actions may not be entirely appropriate for the implementation of profile execution, i.e., when an agent wonders "what would agent *X* (of which I have a profile) do in such and such a situation?". This is because executing a profile should not have consequences for the environment and other agents, so a module representing an agent profile should not be allowed to execute external actions or send messages. To remedy this problem, new actions `dryrun` and `dryrunasync` can be introduced which work just like `execute` and `executeasync`, respectively, with the difference that all external actions which a 'dry-running' module attempts to perform are blocked. Of course, care must be taken that any `execute(async)` actions performed by a dry-running module are also converted to `dryrun(async)` actions.

Secondly, the notion of singleton can be generalized by introducing the possibility of specifying a minimum and maximum amount of instances of a module that can be active at one time. This can be used for ensuring that, e.g., there must always be three to five agents in the role of security guard.

Thirdly, new actions `add` and `remove` can be introduced that accept as arguments a module and a plan or rule, so that all types of contents of 2APL modules can be modified during runtime. In particular, by creating an empty module and using `add` actions, modules can be created from scratch with custom components available at runtime. However, for this to work, plans and rules will have to be first-class citizens, which is currently not the case in 2APL.

References

1. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. In *Proc. of ProMAS '05*, pages 139–155, 2005.
2. P. Busetta, N. Howden, R. Ronnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In N. Jennings and Y. Lesperance, editors, *Intelligent Agents VI: Theories, Architectures and Languages*, pages 277–289, 2000.
3. M. Dastani and J.-J. Meyer. A practical agent programming language. In *Proc. of ProMAS '07*, volume 4908. Springer, 2008.
4. K. Hindriks. Modules as policy-based intentions: Modular agent programming in goal. In *Proc. of ProMAS '07*, volume 4908. Springer, 2008.
5. M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. Goal-Oriented Modularity in Agent Programming. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1271–1278, 2006.