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 modules is generic and facilitates the implementation of different agent concepts such as agent roles and agent profiles, and enables common programming techniques such as encapsulation and information hiding for BDI-based agents. 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 intuitive meaning of these programming constructs are provided as well as the operational semantics of one of the programming constructs. Some informal properties of the programming constructs are discussed and it is explained how these modules can be used to implement agent roles, agent profiles, or the encapsulation of BDI concepts.

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 to BDI-based agent programming languages. There have been many proposals for supporting modules in BDI-based programming languages, e.g., [2, 1, 7, 6]. 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 ways 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 to create events and to generate plans that handle the events. It should be noted that Jadex extends the notion of capability by providing an import/export mechanism to connect different capabilities. In other approaches [6, 7], modules are used to realize a specific policy or mechanism in order to control nondeterminism in agent execution. For example, in [6] modules are considered as the ‘focus of execution’, which can be used to disambiguate the application and execution of plans. In [7] 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 are 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 in a limited way 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 processed. In [6, 7], belief or goal conditions are assigned to modules such that an agent’s interpreter uses the modules when the respective conditions hold.

Like in other approaches, we consider a module as an encapsulation of cognitive components. However, the added value of our approach is that an agent programmer has more control in determining how and when modules are used. In contrast to the abovementioned approaches, we propose a set of generic programming constructs that can be used by an agent programmer to perform a variety of operations on modules. In this way, the proposed notion of module can be used to implement a variety of 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

2APL is a multi-agent programming language that facilitates the implementation of BDI-based agents. The ‘classical’ (i.e. non-modular) version of this programming language is presented in [4, 3]. In this paper, we extend 2APL with modules. In this extension, a 2APL multi-agent program is specified in terms of a set of modules each having a unique name. Initially, a subset of these modules is identified as the specification of individual agents. The execution of a 2APL multi-agent program is therefore the instantiation and execution of this subset of modules. A 2APL module is an encapsulation of cognitive components including beliefs, goals, plans, action specifications, and different sets of rules that generate and repair plans when they are applied. A 2APL module can create, instantiate, and process other 2APL modules. This implies that a 2APL module can include (be specified by) other 2APL modules. There are several operations that a module instance 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 a module instance based on a declared module in the multi-agent program. One module instance can be created by another module instance or an agent that is initially created as an instance of a declared module. In such a case, the creating module instance (also called the owner module instance) will assign a unique name to the created module instance. The owner module instance is the only module instance that can operate on the created module instance until the created module instance is released.

One module instance can create several instances of one and the same module, e.g., an instance $m_i$ of a declared module $M$ can create instances $k_1, \ldots, k_j$ of another declared module $K$. Also, two module instances can create two instances of one and the same module, e.g., instances $m_i$ and $n_j$ of declared modules $M$ and $N$ can create instances $k_i$ and $k_j$ of another declared module $K$. Finally, one and the same module instance can be used by two different module instances, e.g., an instance $k_i$ of a declared $K$ can be used by instances $m_i$ and $n_i$ of declared modules $M$ and $N$, respectively. For this purpose, a special type of module, called a singleton module, is introduced. While the ownership of a singleton module instance can be changed through create and release
operations performed by different module instances, the state of the singleton module instance is invariant with respect to these operations, i.e., the state of a singleton module instance is maintained after one module instance releases it and another one owns it again.

The owner of a module instance can execute it in two different ways. First, the owner can execute its owned module instance and wait until the execution of the owned instance stops. In order to indicate when the owned instance stops (such that the owner’s execution can be resumed), a stopping condition is provided as the argument of the execution operation. This condition, which is specified in terms of the internals of the owned module instance, is evaluated by the overall multi-agent system interpreter. Second, an owner can execute its owned module instance in parallel to its own execution. The execution of the owned module instance stops either by means of a stop condition (evaluated on the internals of the owned module instance) or explicitly by means of a stop action performed by the owner. The execute operations can be used to implement ‘focus of execution’ and goal processing as discussed in [6] and [7], respectively.

Besides executing a module instance, the internals of a module instance can be accessed by its owner module instance. In particular, an owner instance can test and update the beliefs and goals of its owned module instance. In order to control the access to the internals of a module instance, two types of modules are introduced: public and private. A private module instance 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 instance are accessible to its owner. These operations can be used to implement capabilities as discussed in [2, 1]. It is worth noticing that a multi-agent system is the (only) owner of all module instances that initially constitute the individual agents.

3 Syntax

This section presents the complete syntax of the 2APL programming language. As the syntax of the 2APL programming language without modules is presented elsewhere [4, 3], we here highlight the modifications and discuss only the module-related programming constructs. The 2APL syntax for the multi-agent issues is presented by means of a specification language. Using this specification language, one can 1) declare a set of modules, 2) assign external environments to the modules which are then allowed to access the assigned environments, and 3) specify the creation of individual agents as instances of some of the declared modules. 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. A 2APL multi-agent program can thus indicate which modules could be created during the execution of the multi-agent program. This is done by the declaration of a list of module names preceded by the keyword Modules (how the declared modules are implemented will be explained in section 3.2). From the set of declared modules, some will initially be instantiated as individual agents that constitute the implemented multi-agent system. The list of the names of

\[
\begin{align*}
\langle\text{MAS}_\text{Prog}\rangle & ::= \text{"Modules :" }\langle\text{module}\rangle^+ \\
& \quad \text{"Agents :" }\langle\text{agentname} \text{ moduleIdent } \langle\text{int}\rangle\rangle^+ \\
\langle\text{module}\rangle & ::= \langle\text{moduleIdent}\rangle\text{.2apl}\langle\text{environments}\rangle \\
\langle\text{agentname}\rangle & ::= \langle\text{ident}\rangle \\
\langle\text{moduleIdent}\rangle & ::= \langle\text{ident}\rangle \\
\langle\text{environments}\rangle & ::= \text{"@"}\langle\text{ident}\rangle^+
\end{align*}
\]

Fig. 1. The EBNF syntax of 2APL multi-agent systems extended with modules.
the agents that should be created together with their corresponding module names and the number of to be created agents (i.e., the number of module instances to be created) is preceded by the keyword `Agents`. For 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 ⟨ini⟩ 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 has 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.

### 3.1 A 2APL Example

Suppose we need to build a multi-agent system in which one single manager and two workers cooperate to collect gold items in a simple cellular environment called block-world. The manager coordinates the activities of the two workers by asking them either to explore the blockworld environment to detect the gold items or to carry the detected gold items and store them. For this example, which can be implemented as the following 2APL program, the manager module (i.e., `manager.2apl`) specifies the initial state of the manager agent with the name m (the implementation of the manager module is explained later on). The manager module, and thus the manager agent m, can access the database environment. Note that only one manager agent will be initialized and created (line 7). Moreover, the worker module (`worker.2apl`) specifies the initial state of two worker agents. Note that the names of the worker agents in the implemented multi-agent system will be indexed with numbers 1 and 2, i.e., there will be two worker agents with names w1 and w2 (line 8). Finally, two additional modules are declared to implement the explorer and carrier functionalities (line 4, 5). As we will see later on, these functionalities will be used by the worker agents. Note that both functionalities can access the blockworld environment.

```plaintext
1 Modules: // example.mas
2 manager.2apl @database
3 worker.2apl
4 explorer.2apl @blockworld
5 carrier.2apl @blockworld
6 Agents:
7 m manager
8 w worker 2
```

### 3.2 2APL Module Specification

A 2APL module, which is also used to create individual agents, is implemented by means of a 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 are already presented in [4, 3]. In this specification, we use ⟨atom⟩ to denote a Prolog-like atomic formula starting with a 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 is not the focus of this paper, we give a brief and general idea of the basic non-module constructs. 2APL provides programming constructs to implement a module in terms of beliefs, goals, action specifications, plans, and reasoning rules. An agent’s beliefs is a set of Horn-clauses and represent information the agent believes about itself and its surrounding environments. An agent’s goals is a set of conjunctive ground atoms, where each
conjunct 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, and to change the state of external environments. Besides these programming constructs, 2APL provides constructs to implement three types of reasoning rules. The planning goal rules (PG-rules) can be used to generate plans based on the agent's beliefs and goals. The procedure call rules (PC-rules) can be used to generate plans for the received internal and external events including messages. Finally, the plan repair rules (PR-rules) can be used to repair a plan whose execution has failed.

The first module-related construct is the use of keywords public/private and singleton. The owner of a public module instance can both execute as well as access the internals of the owned public module instance. However, the owner of a private module instance can only execute the module instance and cannot access its internals.

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

When a public or private module m is created/instantiated, the created instance can be executed by its owner through the action m.execute(⟨test⟩) or m.executeasync([⟨test⟩]). The execution of a module instance by means of an execute action, performed by an owner, has two effects: 1) it suspends the execution of the owner module instance, and 2) it starts the 2APL deliberation process based on the internals of the owned module instance. The execution of the owner module instance will be resumed as soon as the execution of the owned module instance is terminated. The termination of the owned module instance is based on the mandatory test condition (i.e., the argument of the execute action), which is continuously evaluated by the overall multi-agent system interpreter. As soon as this condition holds, a stop event stop! is sent to the owned module instance. The module instance could then start a cleaning operation after which it should broadcast a return event. For this we introduce an action return that can be executed by an owned module instance after which its execution is terminated. The execution of this final action broadcasts an event return! that is received by the overall multi-agent system interpreter after which the execution of the owner module instance is resumed. The owner agent will be notified about the return event immediately after its execution if resumed. The return event can be used by the owner to, e.g., release the owned module instance.

The execution of a module instance by means of the executeasync action is identical to execute action, except that the owner does not have to wait until the execution of its owned module instance terminates. In fact, the owner continues with its own execution in parallel with the execution of the owned module instance. The execution of the module instance can be halted either by providing a test condition (i.e., the optional

---

1 Note that the owner itself can be an instance of either a public or a private module.
2 The owner cannot do this because its execution has been suspended.
Fig. 2. The EBNF syntax of 2APL extended with modules.
argument of the `executeasync` action) or by means of the `stop` action performed by the owner module instance. Like the `execute` action, the test will be evaluated at the multi-agent system level and based on the internals of the module instance. The `stop` action performed by the owning module instance will send the `stop!` event to the owned module instance.

The owner of a public module instance can access and update the internals of the instance. In particular, the owner can test whether certain beliefs and goals are entailed by the beliefs and goals of its owned public module instance \( m \) through action \( m.B(\varphi) \land G(\psi) \). Also, the beliefs of a module instance \( m \) can be updated by means of action \( m.updateBB(\varphi) \). A goal can be added to the goals of a module instance \( m \) by means of \( m.adopt(\varphi) \) and \( m.adoptz(\varphi) \) actions. Finally, the goals of a module instance \( m \) can be dropped by means of \( m.dropgoal(\varphi) \), \( m.drobsubgoals(\varphi) \) and \( m.dropsupergoals(\varphi) \) actions. As explained in [4, 3], these actions can be used to drop from an agent’s goals, respectively, all goals identical to \( \varphi \), all goals that are a logical subgoal of \( \varphi \), and all goals that have \( \varphi \) as a logical subgoal.

### 3.3 Example Revisited

Given our working example, the manager module can be implemented as follows:

```plaintext
Private // manager.2apl
BeliefUpdates:
{ carryGold(A) } Ready(A) { not carryGold(A) }
{ not carryGold(A) } Busy(A) { carryGold(A) }
Beliefs:
worker(w1).
worker(w2).
divided(L,L1,L2) :- append(L1,L2,L), evenlySized(L1,L2).
Goals:
haveGold()
Plans:
send(w1,request,play(exp))
PG-rules:
haveGold() <- worker(A) and not carryGold(A) |
{ @database(findGoal(A),L); if B(not L=[]) then { send(A,request,play(car,L)); Busy(A) } }
PC-rules:
message(w1,inform,gold(L)) <- divided(L,L1,L2) |
{ @database(addGold(L1,w1),_); @database(addGold(L2,w2),_) }
message(A,inform,done(L)) <- worker(A) | { @database(removeGold(L,A),_); Ready(A) }
```

As illustrated, the goal of the manager \( m \) is to have gold items (line 18). Moreover, it has one initial plan through which it sends a request to worker \( w1 \) to explore the blockworld environment (line 20). The first PC-rule of the manager agent indicates that when it receives a list of detected gold items (i.e., \( \text{gold}(L) \)) from worker \( w1 \), then it divides the received list into two evenly sized lists of gold items and stores them in its database (line 25, 26) (the manager agent could also add these lists to its beliefs, but the aim of the example is to show the use of different environments). The second PC-rule indicates that when a worker informs the manager that it has collected and carried its assigned gold items to a safe depot, then the manager removes the goal items from its database and updates its beliefs with the fact that the worker is ready to carry new gold items (line 27). In order to achieve its goal, the manager agent checks its database continuously to see if it has information about gold items to be collected by one of the worker agents that is not carrying gold (note that the information about gold items should be received from the worker agent that initially was asked to explore the blockworld). If it can find such information (a non-empty list of gold items) in its database, then it will send a request to the corresponding agent asking to carry the gold items and store them safely in the blockworld. The manager agent will update its beliefs to contain information that the agent is busy carrying gold.
The worker agent is an agent that waits for requests to either explore the blockworld environment or carry the gold items and store them. When it receives a request to explore the blockworld environment from the manager (line 34), it creates an explorer module instance and executes it (line 35, 36). Note that the halting condition of this module instance is the belief that gold items are detected. When the execution of the module instance is halted, the worker agent sends the information about the detected gold items to the manager (line 37), updates its beliefs with the information about the detected gold items (this action illustrates the use of abstract action), and finally releases the explorer module instance (line 38). The third PC-rule (line 45) implements the execution of the abstract action `adminGold(L)` by going recursively through the list of gold items and adding each of them to its beliefs. Finally, the second PC-rule of the worker agent (line 39) is responsible for carrying gold items by creating a carrier module instance (line 40), adding the gold item information to its beliefs (line 41), and executing it until either it has found the gold items (`done()` condition) or an error has occurred (`error()` condition).

The explorer module, which is a public module, has the goal to find gold items (line 51). In order to achieve this goal, it performs a sense gold action in the blockworld and adds the information about the detected gold items (i.e., `gold(L)`) to its beliefs (line 55). Note that this belief information is the halting condition of the module instance. In this example, the final PC-rule (line 57) is to react to the stop event that is broadcasted by the platform when the explorer’s stopping condition holds. The reception of this event causes a clean-up operation to be performed by deleting all information about gold items from its beliefs and performing a return action. This return action causes the execution to be handed back to the worker module. Note that the goal `foundGold()` is achieved as soon as `gold(L)` is added to its beliefs.
Domly, the carrier module (also a public module) has a goal to store a list of gold items safely. This goal can be achieved by picking one gold item from the list, store it in a blockworld depot, and remove that gold item from the list of stored gold items. Note the use of two PG-rules (lines 67 and 69) to handle empty and non-empty lists of gold items. Similar to the explorer module, the carrier module does a clean-up operation and performs the return action when it receives a stop event (line 71). The plan repair rule (line 73) adds error information (i.e., \texttt{error()}) to its beliefs when the execution of the \texttt{pickUpGold} action in the blockworld environment fails. Note that \texttt{error()} in the beliefs was one of the halting conditions to stop the execution of the carrier module instance. It is also important to note that it is up to the blockworld programmer to determine when the execution of the \texttt{pickUpGold} action fails.

4 \hspace{1em} 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, due to space limitation, we present only one transition rule to illustrate how a multi-agent system transition (an execution step) can be derived. Here, we do neither present the configuration nor the transitions rules for individual agents. Elsewhere [4] we have presented the semantics of 2APL without modules. The execution of module-related programming constructs affect mainly the multi-agent system configuration. The only effect of the module-related actions at the individual agent level is that these actions are removed from the agent's plans upon execution. It is important to note that individual agent transitions are used as conditions of the multi-agent system transition rules.

The configuration of a multi-agent system is defined in terms of the configuration of modules instances (including agents) and the state of the external environments. The configuration of a module instance includes 1) an instance of the module (consisting of 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, 4) a flag indicating whether 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.

\footnote{Note that there may be several instances of a module specification in a multi-agent system.}
Definition 1 (multi-agent system configuration). Let \((A_i, p, r, e, \varphi)\) 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 \(\varphi\) is the execution stopping condition. Let \(\mathcal{A}\) be a set of module configurations and \(\chi\) be a set of external environments, each a consistent set of atoms \(\langle \text{atom} \rangle\). The configuration of a 2APL multi-agents system is then 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 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 at specific positions in a blockworld environment.

In particular, for each individual agent implemented as \((i : m N)\) (which is preceded by the keyword \texttt{Agents:} in the multi-agent program, \(N\) module/agent instances \((A_{i_0}, \text{mas}, m, t, \bot)\) are created and added to the set of module instances \(\mathcal{A}\). Also, all environments that are assigned to a module in the multi-agent program are initialized and collected in the set \(\chi\). Note that all module instances that are created when the multi-agent program is initialized have \texttt{mas} as parent, \(t\) (true) 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 transition rules. 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.

Due to space limitation, we will present here only the transition rule for the creation of a non-singleton module. For the complete presentation of the formal semantics see [5].

In this transition rule, which is presented below, we use \(A_i \xrightarrow{\alpha} A'_i\) to indicate that the module instance configuration \(A_i\) can make a transition to module instance configuration \(A'_i\) when its execution results in the performance of action \(\alpha\) (and thus broadcasting event \(\alpha'\)). Finally, we assume that \texttt{singleton}(r) holds if and only if the module \(r\) is a singleton module.

\[
\begin{align*}
(A_i, p, r', t, \varphi) & \in \mathcal{A} \& A_i \xrightarrow{\text{create(r,n)!}} A'_i \& \neg\text{singleton}(r) \& \neg\exists r'', e, \varphi': (A_{i_n}, i, r'', e, \varphi') \in \mathcal{A} \\
\hline
& \langle A_i, \chi \rangle \rightarrow \langle A', \chi \rangle
\end{align*}
\]

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

The transition rule indicates the effect of the \texttt{create(r,n)} action performed by the execution of module instance \(A_i\) (the owner module instance), where \(r\) is the identifier of a non-singleton module specification (of which an instance should be created), and \(n\) is the name that will be assigned to the created module instance. This transition rule requires that the owner module instance \(i\) is in the execution mode (i.e., the execution flag equals \(t\)) and that there is no module instance with the same name already created by the same module (i.e., \(\neg\exists r'', e, \varphi': (A_{i_n}, i, r'', e, \varphi') \in \mathcal{A}\)). 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 \texttt{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 (i.e., the execution flag equals \(f\)) and its stopping condition is set to \texttt{falsum} \(\bot\). Note also that the stopping condition will be changed when the module is executed.
5 Roles, Profiles, and Task Encapsulation

The proposed module extension of 2APL is general enough to be useful for the implementation of several agent-oriented programming topics. These include the implementation of agent roles, agent profiles, and encapsulation of cognitive attitudes.

5.1 Agent Roles

The run-time creation and execution of a module instance can be used to implement the activation and enactment of a role. The module specification should then be considered as the specification of the role. In particular, the action `create(role, name)` can be seen as the activation of a role, by which the activating agent (owner) 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. Note that when the role has been declared as singleton, this property of locking is important, because other agents cannot 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 terminating condition, at which point control is returned to the owner agent. Alternatively, the role may be executed using `name.executeasync(ϕ)`, meaning that role will run parallel to the owner agent. Note that supplying as terminating condition ϕ = ⊥ 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 module instances. 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 instance. 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 [2, 1]) is that the contents of an included module instance are simply added to the including agent, whereas the contents of a created module instance 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 to implement modules in BDI-based agent programming languages. We have illustrated this mechanism by extending the syntax and (operational) semantics of 2APL with transition rules for module-related actions that allow module instances to be created, executed, queried, modified, and to be released again. Each module instance is 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 a proof of concept of the proposed extension by implementing the presented operational semantics in the current 2APL platform. It should be noted that modularity in programming languages is not new. Our proposed notion of modules is inspired on the concepts found in many languages, particularly object oriented languages. As a consequence some properties are the same, e.g. modules have an owner, which dictate the life cycle of the module. Also a module is designed with a particular task in mind, hiding the detail for the owner.

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. First, the execute and executesync 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. Second, 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. Third, new actions add and remove can be introduced that accept as arguments a module instance and a plan or rule, so that all types of contents of 2APL module instances can be modified during runtime. In particular, by creating an empty module instance and using add actions, modules instances can be created from scratch with custom components available at runtime.

References