Modularity in BDI-based Multi-Agent Programming Languages

Mehdi Dastani and Bas Steunebrink
Utrecht University
The Netherlands
Email: {mehdi,bass}@cs.uu.nl

Abstract—This paper proposes a module-based vision for designing BDI-based multi-agent programming languages. The introduced concept of modules enables common programming techniques such as encapsulation and information hiding for BDI-based programs, and facilitates the implementation of agent roles and profiles. This vision is applied to a BDI-based agent programming language to which specific programming constructs are added to allow the implementation of modules. The syntax and intuitive semantics of module-based programming constructs are explained. An example is presented to illustrate how modules can be used to implement BDI-based multi-agent systems.

I. 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 some proposals for supporting modules in BDI-based programming languages, e.g., [2], [3], [5], [8]. 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, and plans that together model a specific functionality and can be used to handle specific situations or tasks. However, the way the modules are used in these programming approaches are different.

For example, in Jack [3] and Jadex [2], modules (which are also called capabilities) are employed for information hiding and reusability by encapsulating different cognitive components that together implement a specific capability/functionality of the agent. In these approaches, the encapsulated components are used during an agent’s execution to process the events received by the agent. Moreover, in GOAL [5] modules are considered as the ‘focus of execution’, which can be used to disambiguate the application and execution of plans. This is done by assigning a mental state condition (beliefs and/or goals) to each module. The modules whose conditions are satisfied form the focus of an agent’s execution such that only plans from these modules are applied and executed. Finally, in 3APL [8] a module can be associated with a specific goal indicating which planning rules can be applied to achieve the goal. In other words, a module implements specific means for achieving specific goals. It should also be noted that the concept of module as used in [6] is different than in other approaches. A module in [6] is considered as one specific cognitive component (e.g., an agent’s beliefs) and not as a functionality modeled by different cognitive components.

In these proposals, most module-related 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 (i.e., agent deliberation cycle). An agent programmer can control the use of modules during an agent’s execution indirectly and implicitly either based on the predetermined functionality given to the modules or through conditions assigned to them. For example, in Jack [3] and Jadex [2] the agent’s interpreter uses modules to process the received events. In [5], belief or goal conditions are assigned to modules such that an agent’s interpreter uses the modules when the respective conditions hold. Finally, in [8] a programmer has only a limited control over the modules by indicating which modules (i.e., which planning rules) should be used to achieve a goal.

Like in other approaches, we consider a module as an encapsulation of different cognitive components that together implement a specific agent functionality. However, the added value of our approach is that a programmer can perform a variety of operations on modules. These module-related operations enable a programmer to directly and explicitly control when and how modules are used. Thus, 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. 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.

II. BDI PROGRAMMING WITH MODULES

Programming a BDI-based individual agent amounts to specifying its initial (cognitive) state in terms of beliefs (information), goals (objectives), and plans (means). In programming terminology, the beliefs, goals, and plans can be considered as (cognitive) data structures specifying the state of the agent program. The execution of a BDI-based agent program, which is supposed to modify the state of the agent program, is based on a cyclic process called deliberation cycle (sense-reason-act cycle). Each iteration of this process starts with sensing the environment (i.e., receive events and
messages), reasoning about its state (i.e., update the state with received events and messages, and generate plans to either achieve goals or to react to events), and performing actions (i.e., perform actions of the generated plans). Similar BDI ingredients and deliberation cycles are used in existing BDI-based programming languages such as Jason [1], 2APL [4], JadeX [7], and Jack [9].

A multi-agent program consists of a set of modules with unique names, each specifying a state in terms of cognitive concepts. Initially, a subset of these modules is identified as the specification of the initial state of individual agents. The execution of a multi-agent program is then the instantiation of this subset of modules followed by performing a deliberation process on each module instance. In this way, an instance of a module forms the initial state of an individual agent. It should be emphasized that a module instance specifies the cognitive state of an agent while the agent itself is the deliberation process working on the cognitive state.

We do not present here the complete syntax of a modular BDI-based agent programming language as we aim at focusing on modules and module-related actions. In fact, we assume that a module is just like an agent program specifying a cognitive state by means of programming constructs (for beliefs, goals, and plans) of existing BDI-based programming languages extended with module-related actions. Moreover, we assume that the proposed module-related actions can be added to any existing BDI-based agent programming language [1], [4], [7], [9].

For the sake of presenting an example, however, we consider an agent’s beliefs being implemented by a set of Horn-clauses. An agent’s goals are assumed to be implemented by a set of conjunctive ground atoms, where each conjunction represents a situation the agent wants to realize. An agent is assumed to be capable of performing different types of actions such as update actions (to modify beliefs and adopt and drop goals), belief and goal test actions (to query beliefs and goals), and actions to send messages and to change the state of external environments. Moreover, an agent is assumed to generate plans at runtime by applying rules. These rules can be used to generate plans based on either the agent’s beliefs and goals, or the received internal and external events including messages from other agents. Rules have the form

\[ \text{trigger} | \text{guard} \rightarrow \text{plan}, \]  

where \( \text{trigger} \) is either a goal or an event query of the form \( G(\phi) \) or \( E(\phi) \), respectively, and the \( \text{guard} \) is a belief query of the form \( B(\phi) \). Finally, the \( \text{plan} \) is the plan to be generated and added to the set of plans if both \( \text{trigger} \) and \( \text{guard} \) hold. Similar BDI related programming constructs occur in many existing BDI-based agent programming languages such as Jason [1], JadeX [7], Jack [9], and 2APL [4].

The first module-related action is \( \text{create} \left( \text{mod-name}, \text{ins-ident} \right) \), which can be used to create an instance of the module specification named \( \text{mod-name} \). The name that is assigned to the created module instance is given by the second argument \( \text{ins-ident} \). 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 \( \text{release} \left( m \right) \) action. This means that its instance is removed/lost.

A module instance \( m \) can be executed by its owner through the \( \text{execute} \left( m, \text{test} \right) \) action. The execution of a module instance, performed by its owner, has two effects: 1) it suspends the execution of the owner module instance, and 2) it starts the execution 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. In a sense, an agent that executes an owned module instance, stops deliberating on its current cognitive state and starts deliberating on a new cognitive state.

The termination of the owned module instance\(^1\) is based on the mandatory test condition (i.e., the second argument of the \( \text{execute} \) action). When this condition holds, a stop event is sent to the owned module instance. The module instance can use the received event and start a cleaning operation after which it should broadcast a return event. For this we introduce an action \( \text{return} \) that can be executed by an owned module instance after which its execution is halted and the execution of the owner module instance is resumed.

The owner of a module instance can access, query, 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 module instance \( m \) through action \( \text{test} \left( m, \phi \right) \). Also, the beliefs and goals of a module instance \( m \) can be updated by means of the actions \( \text{updateB} \left( m, \phi \right) \) and \( \text{updateG} \left( m, \phi \right) \), respectively. A typical life cycle of a module in terms of these operations is illustrated in Figure 1.

### III. An Example

In order to illustrate the idea of module-related constructs the following example is provided. This example is not

---

\(^1\) The owner cannot force the owned module instance’s execution to stop because its own execution has been suspended.
intended to demonstrate the practical use of the constructs for which we may need substantially more space. Suppose we need to build a multi-agent system in which one single manager and three workers cooperate to collect gold items in an environment called gridworld. The manager coordinates the activities of the three workers by asking them either to explore the gridworld environment to detect the gold items or to carry the detected gold items to a depot and store them. For this example, which can be implemented as the program illustrated in Figure 2, the module declaration includes a manager module (i.e., manager.mod) which specifies the initial state of the manager agent with the name m (the implementation of the manager module is presented in Figure 3). Note that only one manager agent will be initialized and created (line 7). Moreover, the worker module (worker.mod; see Figure 4) specifies the initial state of three worker agents. The names of the worker agents in the implemented multi-agent system will be indexed with numbers, i.e., there will be three worker agents with names w1, w2, and w3 (line 8). Finally, two additional modules are declared to implement the explorer and carrier functionalities (line 4, 5). As we will see, these functionalities will be used at runtime by the worker agents. Note that both functionalities can access the ‘gridworld’ environment.

Fig. 2. The multi-agent program of the running example.

The manager module can be implemented as in Figure 3. The goal of the manager module is to have gold items (line 10). Moreover, it has one initial plan through which it sends a request to worker w3 to explore the gridworld environment (line 11). The first rule of the manager agent (lines 13-18) indicates that the goal to have a gold item (i.e., G(haveGold())) can be achieved if the agent believes that there is a gold item and there is a free worker agent to collect the gold item (i.e., B(gold(POS) & worker(A))). The plan to achieve this goal is by sending a message to the free agent asking him to play the role of carrier to collect the gold item and modify its beliefs to record the fact that the free agent is not free anymore (i.e., Belief(assigned(POS,A))), as it now has an assigned task. The second rule (lines 19-21) indicates that whenever the manager receives an event (message) containing the information about the position of a gold item (i.e., gold(POS)), it updates its beliefs with this information (line 20). The third rule (lines 22-23) indicates that when a worker informs the manager that it has collected and carried its assigned gold items to the depot, the manager updates its beliefs (atoms preceded by a minus sign are removed) with the fact that the worker is ready to carry new gold items again (line 23).

Here we assume that the manager is aware of the three created workers, i.e., it has the identities of the workers. This assumption can be relaxed by making a query to a possibly existing agent management system to get the identifier of a worker.

Fig. 3. The code of the manager module.

The worker agent, as implemented in Figure 4, is an agent that waits for requests to either explore the gridworld environment or carry the gold items and store them. When it receives a request to explore the gridworld environment from the manager (line 26), it creates an explorer module instance and executes it (line 28-29). Note that the stopping condition of this module instance is the belief that gold has been found. When the execution of the module instance halts, the worker agent queries the beliefs of the explorer module instance (line 30) to retrieve the position of the detected gold item, sends this information to the manager (line 31), and finally releases the explorer module instance (line 32). The second rule of the worker agent (line 34) is responsible for carrying gold items by creating a carrier module instance (line 36), adding the gold item information to its beliefs (line 37), and executing it until either it has found the gold items (done() condition) or an error has occurred (error() condition); see line 38.

Fig. 4. The code of the worker module.

The explorer module, as implemented in Figure 5, has the goal to find gold items (line 44). In order to achieve this goal, it proceeds to a random location in the gridworld, performs a sense gold action there and, if successful, adds the position of the detected gold item (i.e., gold(POS)) to its beliefs (line 49). Note that this belief information satisfies the stopping condition of the module instance (line 29) since the goal foundGold() is achieved as soon as gold(POS) is added.
to its beliefs (line 43). In this example, the final rule (line 51) is to react to the stop event broadcast when the explorer’s stopping condition holds. The reception of this event causes the explorer module to perform a return action, which in turn causes the execution to be handed back to the worker module.

```prolog
Beliefs = [ foundGold() :- gold(_) ];
Goals = [ foundGold() ]
Rules = [
  G( foundGold() ) | true ->
  { @gridworld( getToRandomPosition() ) ;
  @gridworld( senseGold() , POS ) ;
  if POS != nil then Belief( gold(POS) ) ;
  },
  E( stop ) | true -> { return; }
];

Fig. 5. The code of the explorer module.

Finally, the carrier module as implemented in Figure 6 has a goal to store a gold item (line 53). This goal can be achieved by fetching the gold item, storing it in the depot, and removing that gold item from the beliefs (lines 56-58). Similar to the explorer module, the carrier module performs a return action when it receives a stop event (line 60). The third rule (line 61) adds error information (i.e., error()) to its beliefs when the execution of an action in the gridworld environment fails. Note that error() in the beliefs was one of the stopping conditions to stop the execution of the carrier module instance (line 38). It is also important to note that it is up to the gridworld programmer to determine when the execution of a gridworld action fails.

```prolog
Beliefs = [ goldStored() :- not gold(_) ]
Goals = [ goldStored() ]
Rules = [
  G( goldStored() ) | B( gold(POS) ) ->
  { @gridworld( fetchGold(POS) ) ;
  @gridworld( storeGold() ) ;
  Beliefs{ +gold(POS), done() };
  },
  E( stop ) | true -> { return; }
];

Fig. 6. The code of the carrier module.

IV. CONCLUSIONS AND FUTURE WORK

This paper introduced a vision for designing and integrating modules in BDI-based agent programming languages. We have illustrated how modules can be used to facilitate the implementation of notions relevant to agent programming. 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. module instances have an owner, which dictate the life cycle of the module. Also a module is designed with a particular task in mind, hiding the details from the owner.

For future work, there are several extensions to this work on modularization that can make it more powerful for encapsulation in general and implementation of agent roles and agent profiles in particular. Firstly, as agents can be specified in terms of beliefs, goals and plans, we can use modules to represent agents. An agent can thus create and maintain profiles of other agents by creating module instances. The execute action may then 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. Also, the execute action can be generalized to allow the simultaneous execution of multiple module instances. Doing so one may be able to implement agents that can play several roles simultaneously.

Secondly, the notion of module 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. Additionally, one may want to be able to pass ownership of a module instance from one agent to another (especially when the module in question models a role) without losing its internal state.

Thirdly, additional actions such as updateP and updateR can be introduced that accept as arguments a module instance and a plan or rule, so that all types of contents of module instances can be modified during runtime. In particular, by creating an empty module instance and using update* actions, modules instances can be created from scratch with custom components available at runtime. A related issue is the access to the internals of module instances by means of test and update actions. In order to manage the access to the internals of module instances, modules can be specified as private or public allowing restricted access to the internals of modules.

REFERENCES