

# Belief/Goal Sharing Modules for BDI Languages

Michal Cap<sup>\*†‡</sup>, Mehdi Dastani<sup>\*</sup> and Maaïke Harbers<sup>\*</sup>

<sup>\*</sup>Institute of Information and Computer Sciences, Utrecht University, Utrecht, Netherlands

Email: {mehdi,maaike}@cs.uu.nl

<sup>†</sup>ATG, Dept. of Cybernetics, FEE, Czech Technical University, Prague, Czech Republic

Email: cap@agents.felk.cvut.cz

**Abstract**—This paper proposes a modularisation framework for BDI based agent programming languages developed from a software engineering perspective. Like other proposals, BDI modules are seen as encapsulations of cognitive components. However, unlike other approaches, modules are here instantiated and manipulated in a similar fashion as objects in object orientation. In particular, an agent’s mental state is formed dynamically by instantiating and activating BDI modules. The agent deliberates on its active module instances, which interact by sharing their beliefs and goals. The formal semantics of the framework are provided and some desirable properties of the framework are shown.

**Index Terms**—Intelligent agent; Software engineering;

## I. INTRODUCTION

The agent oriented programming paradigm promotes a societal view of computation, where solutions are achieved by cooperation of autonomous entities – agents [1]. Several programming languages emerged to facilitate the development of autonomous agents, based on different cognitive and philosophical theories. This paper focuses on a family of programming languages based on the Belief-Desire-Intention (BDI) theory [2], [3]. BDI languages (e.g. 2APL [4], GOAL [5], JACK [6], Jadex [7], Jason [8]) offer constructs inspired by mental notions such as beliefs, goals and plans to implement agent behaviour. As in other programming paradigms, the ability to decompose BDI programs to separate, to some extent independent modules, is crucial for the development of complex software systems. Yet, a widely accepted concept of modularisation for BDI programming languages is still missing (see Section II for existing approaches).

We propose a modularisation framework for logic-based BDI languages to overcome some of the limitations of existing frameworks and unify commonly accepted characteristics of various existing approaches into one single framework. The proposed framework extends earlier work by Dastani et al [9] (see Section 3 for the details) and has the following characteristics. 1) A module is an encapsulation of beliefs, goals, plans and reasoning rules that together specify a functionality, a capability, a role, or a behaviour. 2) An agent’s mental state is modelled as a tree of module instances, in which a link is created when one module instance activates another. Using this mechanism, a set of dependent module instances

can be deactivated and reactivated again by means of a single action. 3) An agent’s module instances are executed in parallel. This allows the agent to play several roles or use several capabilities at the same time. 4) Module instances can be created and released, and added to or removed from an agent’s mental state at run-time. This can be used, e.g., to dynamically enact and deact roles. 5) Inactive and active module instances are distinguished. An inactive module instance is generally used as a named container for beliefs and goals, while an active module instance is typically used for encapsulation of behavioural rules (specifying plans to achieve goals and respond to events). 6) Each module instance is associated with an interface determining its interaction with other module instances, i.e. the beliefs and goals that are shared with other module instances. This way, a module’s public interface is separated from its private internals. 7) An agent’s module instances can be clustered into separate belief/goal sharing scopes, which allows the agent to maintain mutually inconsistent belief bases, e.g. to model different possible worlds or profiles of other agents.

From a methodological point of view, we can identify the following characteristics. 1) The framework is easy to grasp for programmers acquainted with object orientation because module instances are manipulated similarly to objects. 2) Programmers have explicit control over the life cycle of a module, i.e. they can indicate when to create/instantiate modules, how to operate on them, and when to release them. 3) The module interface can be used to denote the intended use of a module. By convention, the use of particular interface should be documented by a semi-formal comment (similar to JavaDoc comments) above the respective interface entry. 4) Active module instances interact by sharing some of their beliefs and goals which promotes loose coupling. A module instance can easily be replaced (even at runtime) as long as the new module uses the same beliefs and goals for interaction with the agent’s other modules.

This paper is organised as follows. Section 2 discusses other approaches to BDI modularisation. Section 3 presents the concept of belief/goal sharing modules and introduces an example program illustrating its use. Section 4 presents the operational semantics of the module-related constructs. In Section 5, we discuss properties of the modular approach, and Section 6 concludes the paper and proposes some directions of further research.

<sup>‡</sup>The research was done at Utrecht University. The first author is now affiliated with CTU Prague and supported by the Czech Ministry of Education, Youth and Sports, grant MSM6840770038 and the Grant Agency of the Czech Technical University in Prague, grant SGS10/189/OHK3/2T/13.

## II. RELATED RESEARCH

There are several proposals for modularization of BDI-based agent programming languages. In all proposals, a BDI module encapsulates cognitive components such as beliefs, goals, plans, and reasoning rules. In the work of Hindriks [10], modules are used to disambiguate the application and execution of plans. A mental state condition (beliefs and/or goals) assigned to each module states when its plans are applied and executed. Madden and Logan [11] revised Jason to develop modular programs. A Jason agent is modelled as a one-level hierarchy of modules, where each module can contain local beliefs, goals, plans and an event queue. Local beliefs and goals can be exported, and then imported by another module. A rather different approach is proposed by Novak [12], in which a module is considered as one specific cognitive component (e.g. an agent's beliefs) instead of a functionality modeled by different cognitive components. This allows a programmer to use the best suited knowledge representation technique for an agent's belief and goal base(s). In JACK [13] and Jadex [14], an agent can be specified as a composition of so called capabilities (modules) that are seen as functional clusters of BDI components. The capabilities process the events received by the agent during its execution. Van Riemsdijk et al [15] propose the concept of goal-oriented modularity for 3APL, where modules encapsulate information about how to achieve a (set of) goal(s). Finally, Dastani et al [9] extended 2APL with modularity, where a module is specified by goals, beliefs, plans and rules. In the approach, agents can instantiate a module instance from a module specification and discard an existing module instance during runtime. Furthermore, the beliefs and goals of a module instance can be updated and queried, and the execution control can be handed over to a given module instance.

While these approaches satisfy some of the characteristics mentioned in the introduction, none of them integrates all in a unified approach. In particular, the notions of belief/goal sharing and a belief/goal sharing scope have not been studied in literature before. In contrast to our approach, a number of proposals [9], [10], [15] only allows one active module at a time. The interpreter of modular Jason [11] does consider rules from multiple modules, but it does not allow adding or removing modules at run-time. Furthermore, in our proposal a programmer can directly control the use of modules, whereas in other approaches the interpreter monitors conditions assigned to modules and automatically activates them (e.g. [10]). The notion of capability used in Jadex [14] is in many respects similar to the notion of module in our proposal, but the interaction between modules and capabilities differs. Jadex uses an import/export mechanism through which beliefs or goals can be imported from an explicitly specified capability. In our approach, in contrast, the programmer specifies beliefs and goals to be shared, but not the specific module instance. Note that the latter approach adheres more to the design principle of loose coupling. Finally, the capability framework in Jadex is defined informally and suited mainly for Java-based

agent programming languages (e.g. Jadex, JACK), while we define our framework using formal semantics and focus on logic-based agent programming languages (e.g. 2APL, Jason, and GOAL).

## III. BELIEF/GOAL SHARING BDI MODULES

We do not present the complete syntax of a modular BDI-based programming language here, since we only focus on modules and module-related constructs. To present the concept of belief/goal sharing modules, we make the following assumptions about the BDI language. An agent's beliefs and goals are implemented by a set of ground atoms and a set of conjunctive ground atoms, respectively. Each conjunction represents a situation the agent wants to realize. The agent's reasoning is governed by the rationality principle, i.e. the agent cannot desire a goal it believes to be achieved. The agent is capable of different action types 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, the agent is assumed to generate plans at runtime by applying rules. Rules have the form  $trigger \mid guard \rightarrow plan$ , where  $trigger$  is a goal query of the form  $G(\varphi)$  and the  $guard$  is a belief query of the form  $B(\varphi)$ . Finally,  $plan$  is the plan to be generated and added to the set of plans if both  $trigger$  and  $guard$  hold, that is, when they are derivable from the goal and belief base, respectively.

As mentioned earlier, the framework presented in this paper can be seen as an extension of the work by Dastani et al [9]. While introducing the framework, we will point out the differences between this earlier proposal and our work.

### *General Description*

In our proposal, an agent's mental state is modelled as a tree-like structure of module instances and contains at least one module instance – the agent's *main module instance*. Module instances that are a part of an agent's mental state are considered *active*, and those that are not are considered *inactive*.

*Concurrency:* We consider an agent as the execution of a deliberation cycle on its mental state. In each cycle, the agent senses its environment (receiving events and messages), reasons (updating its state based on the received events and messages, and generating plans to achieve goals or react to events), and acts (performing actions of generated plans). In the earlier proposal, the deliberation cycle always operates on only one of the agent's modules, and execution control is handed over to another module by the `execute` action. In our proposal, all of the agent's active module instances are processed concurrently. During each deliberation cycle, the interpreter performs the following operations in each active module of an agent: 1) updating its state with received events and messages, 2) applying all applicable behavioural rules and 3) performing one action from each plan. As a result, the agent's active modules can be perceived as being executed concurrently.

Such a functionality can be fruitfully used in a number of situations. E.g., you can imagine an agent that is composed of two modules: one for movement within the environment and the other one for communication with the other agents. In our framework, both activities can be handled simultaneously.

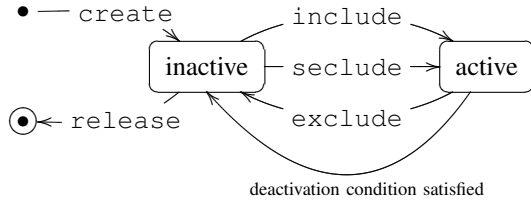


Fig. 1. Lifecycle of a Module Instance

*Module Instantiation:* Figure III shows the life cycle of a module instance. The first module-related action  $\text{create}(s, m)$  is used to create a new inactive module instance  $m$  from a module specification  $s$ . The system generates a unique identifier for each new module instance and unifies it with the provided variable  $m$ . The resulting module instance represents a static collection of data, which can be inspected and altered from any module instance in which the concrete value of the module instance identifier  $m$  is available. In particular, the beliefs and goals of an inactive module instance  $m$  can be queried by means of the  $\text{test}(m, \varphi)$  action. Furthermore, its belief and goal base can be updated and manipulated by the  $\text{updateB}(m, \varphi)$  and  $\text{updateG}(m, \varphi)$  actions, respectively. Finally, an inactive module instance  $m$  can be discarded by the  $\text{release}(m)$  action.

In contrast to the earlier proposal, we allow and encourage programmers to use variables at any place where a module instance identifier is expected. Note that this closely resembles the process of object instantiation as used in the majority of object oriented programming languages.

In the earlier proposal, a module instance is accessible only from its creator. In our proposal, a module instance can be accessed from any other module instance. However, since initially only the creator of a module instance has access to its unique identifier, the access to the module instance is in practice limited to the module instances to which the identifier was disclosed by the creator, e.g. by means of communication.

*Module Activation:* In the earlier proposal, module instances are only activated by the  $\text{execute}$  action. In our proposal, an agent can activate an inactive module instance  $m$  by the  $\text{include}(m, \varphi)$  and  $\text{seclude}(m, \varphi)$  actions, where  $\varphi$  specifies when the module instance should be deactivated. The two methods differ with respect to belief/goal sharing, on which we will elaborate later. The root node of an agent’s module instance tree is always the agent’s main module instance. When an inactive module instance is activated, it becomes a child node of the module instance that activated it. As soon as a module instance is activated, all its descendants are implicitly activated as well.

An active module instance  $m$  can be deactivated, i.e. removed from an agent’s mental state, either explicitly by means

of the  $\text{exclude}(m)$  action or implicitly when a predefined deactivation condition has been satisfied. The parameter  $\varphi$  in the  $\text{include}(m, \varphi)$  and  $\text{seclude}(m, \varphi)$  actions is optional, and if no deactivation condition is provided, it is assumed to be falsum. In this paper, deactivation conditions have the form  $G(\psi)$  or  $B(\varphi)$ , i.e. a deactivation condition holds if  $\psi$  or  $\varphi$  are derivable from the goal or belief base, respectively. When a module instance is deactivated, all its descendants are implicitly deactivated as well. Note that after deactivation, the module instance preserves its beliefs and goals, but cannot share them with the other module instances.

In the earlier proposal, only the creator of a module instance can activate it. We, in contrast, let a module instance activate any other module instance as long as it has access to its unique identifier. This implies that one module instance can belong to the mental state of many different agents during its life-time (but never at the same time).

*Belief/Goal Sharing:* Since the beliefs and goals of an active module instance change at run-time as a consequence of the deliberation process, it is unsafe to access its internals directly. Therefore, actions like  $\text{test}(m, \varphi)$ ,  $\text{updateB}(m, \varphi)$  and  $\text{updateG}(m, \varphi)$  cannot be executed on an active module instance. Instead, the run-time interaction between the active module instances is realized by sharing their beliefs and goals. Each module instance specifies an interface with the beliefs and goals to be shared with the other module instances. As in some situations it is undesirable to let all the agent’s module instances share beliefs and goals with each other, we provide methods to divide module instances into separate clusters, each forming an independent belief/goal sharing scope.

Each agent contains initially one sharing scope, which consist only of the agent’s main module instance. Module instances that are activated by the  $\text{include}(m, \varphi)$  action are added to an existing sharing scope, and can share their beliefs and goals with the other module instances in that sharing scope. Module instances that are activated by the  $\text{seclude}(m, \varphi)$  action, in contrast, yield a new sharing scope and cannot share their beliefs and goals with other module instances until new module instances are included to that sharing scope. The possibility of having several sharing scopes allows an agent, for instance, to keep a profile of another agent, where the profile’s beliefs and goals may be contradictory to the agent’s own. Another application would be an agent who reasons about possible future world states. Such an agent might use an isolated sharing scope to keep each alternative world state it can foresee.

*Module Interface:* Each module instance has a module interface which is defined as a set of entries. Each module interface entry is an atomic formula used as a template matching concrete beliefs and goals. Interface entries typically contain variables. A module interface serves several functions: 1) *Ontological Function* – A module interface specifies the language that is to be used to interact with the module. The module interface language allows to build formulas composed of the atomic formulas that are unifiable with one of the interface entries, and logical connectives  $\text{and}$ ,  $\text{or}$  and  $\text{not}$ .

All beliefs and goals interfaced by the module instance will be expressed in the module interface language. 2) *Export Declaration Function* – A module interface defines which of the local beliefs and goals of the module instance are interfaced and will thus be constitute beliefs and goals of its sharing scope. 3) *Import Declaration Function* – A module interface defines which of the global beliefs and goals will be accessible for the module instance. 4) *Visibility Restriction Function* – The module interface may be used to limit the visibility of the internals of a module instance. Any belief or goal that cannot be expressed in terms of the module interface language stays private and cannot be accessed from outside the module instance.

### Example

We introduce a simple example to demonstrate two typical interaction patterns between an agent’s active module instances. We assume a multi-agent system consisting of several worker agents who have to find bombs in a gridworld environment and dispose them into a designated bomb trap. The functionality needed to perform these two tasks is implemented in separate modules. A *searching* module instance actively searches the environment and maintains a database of believed bomb positions, a *disposing* module instance contains the functionality needed to dispose the given bomb to the trap, and a *gridworld* module instance provides higher-level agent-oriented methods for accessing the gridworld environment.

The main module specification of a worker agent instantiates the aforementioned modules within its initial plan and includes them to its sharing scope:

```
Plans = {create(searching,S);include(S);
create(disposing,D);include(D);
create(gridworld,GW);include(GW)}
```

The four modules interact through their interfaces, as shown in Figure 2.

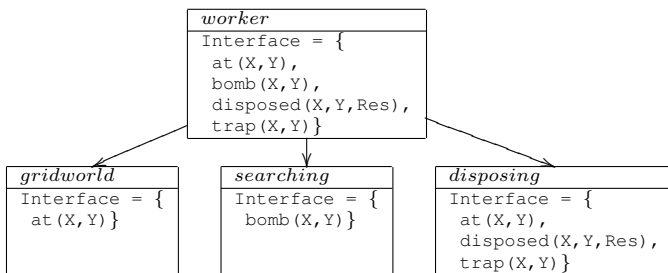


Fig. 2. Modules of the Worker Agent

*Providing Required Beliefs:* If a module instance uses information it cannot obtain itself, the belief sharing mechanism can retrieve the information from outside. In our example, the *disposing* module instance does not have any internal means to determine the position of the trap (i.e. `trap(X,Y)`), but it can use the information provided by the main *worker* module instance instead.

The following code excerpt of the *disposing* module shows a rule which can generate a plan to achieve the goal

`disposed(X,Y,succeeded)`. The plan indicates that a new goal should be adopted (i.e. be at position `TX,TY`, represented by `at(TX,TY)`), and blocks until this goal is believed to be achieved (i.e. `B(at(TX,TY))`). The *disposing* module instance can query the belief `trap(TX,TY)` (the guard of the rule) since the belief will be provided in the form of a global belief.

```

Interface = {trap(X,Y), at(X,Y)...}

Rules = {
  G(disposed(X,Y,succeeded)) | B(trap(TX,TY)) -> {
    ...
    adoptgoal(at(TX,TY));
    B(at(TX,TY));
    ...
  }
}

```

The concrete belief is specified in the local belief base of the *worker* module specification and interfaced, e.g.,

```

Interface = {trap(X,Y),...}
Beliefs = {trap(0,0),...}

```

*Delegating Goal Pursuit:* Another typical design pattern that uses belief/goal sharing is delegating goal pursuit. This involves the adoption of a goal in a module instance that it is incapable to achieve itself, but that another module instance within its sharing scope can achieve. The first module instance can monitor the pursuit of the goal by a query on the corresponding belief. In our example, the *disposing* module instance may e.g. adopt the goal `at(5,7)` although it has no actual means to achieve the goal itself. However, since the atom `at(X,Y)` is declared as an interface entry in the *disposing* module specification, the goal `at(5,7)` will be exported and becomes a global goal of the agent<sup>1</sup>. The *gridworld* module specification also declares the atom `at(X,Y)` in its interface, and therefore imports the global goal. Subsequently, it will generate a plan which performs the action `goto(X,Y)` in the external environment and then updates the agent’s beliefs with its current position by the construct `Belief(-at(OX,OY), at(X,Y))`. After the global beliefs have been updated with the belief `at(X,Y)`, the global goal `at(X,Y)` is automatically dropped due to the rationality principle.

```

Interface = {at(X,Y),...}

Rules = {
  G(at(X,Y)) | B(at(OX,OY)) -> {
    @gridworld(goto(X,Y));
    Belief(-at(OX,OY), at(X,Y))
  }
}

```

## IV. SEMANTICS

The semantics of the proposed actions are defined in terms of a transition system. A transition specifies a single computational/execution step by indicating how one configuration can be transformed into another. We first present the multi-agent system configuration. Then, we present the transition rules from which the possible execution steps of a multi-agent

<sup>1</sup>More precisely, it will become a global goal of the sharing scope of the *disposing* module instance, but since the agent consists of only this sharing scope, it can be regarded as the agent’s global goal.

program can be derived. We focus only on the semantics of the module-related constructs. The semantics of non-module constructs are out of the scope of this paper and depend on the specific BDI programming language.

### Multi-Agent System Configuration

The configuration of a multi-agent system can be represented as a tuple  $\langle \mathcal{M}, \mathcal{A} \rangle$ , where  $\mathcal{M}$  is a set of module instance configurations and  $\mathcal{A}$  is a set of identifiers that represent the main module instance of each agent.

An agent is specified by its mental state, which is modelled as a tree-like structure of module instances which can interact. The agent's main module instance serves as the root node. An agent also comprises a deliberation cycle operating on its mental state.

The state of an individual module instance is captured in the form of a tuple  $(M_\iota, \omega, \epsilon, v)$  representing the module configuration. The mental state of a module instance  $\iota$  (i.e. its local belief base, goal base, rule base, etc.) is denoted by  $M_\iota$ ,  $\omega$  is a test expression representing the deactivation condition of the module instance,  $\epsilon$  is a set of module identifiers representing module instances that the module instance  $\iota$  secludes, and  $v$  is a set of module identifiers representing module instances that the module instance  $\iota$  includes. The sets  $\epsilon$  and  $v$  point to the children of the module instance.

### Module Linearisations

The module instances residing in the multi-agent system form a directed graph of module instances. Such a structure can be linearised in a number of ways, where each can be used for a number of purposes, e.g. to determine which module instances are currently active or belong to one sharing scope.

In the following definitions we will assume a given multi-agent system  $\langle \mathcal{M}, \mathcal{A} \rangle$ . Let  $v_m$  be a set of identifiers of module instances the module instance  $m$  includes and  $\epsilon_m$  be a set of identifiers of module instances the module instance  $m$  secludes. The set  $\mathbb{M}$  contains identifiers of all module instances residing in the given multi-agent system.

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

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

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

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

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

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

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

### Transition rules

In this section we provide transition rules for module-related actions. We will use  $M_\iota \xrightarrow{\alpha!} M'_\iota$  to indicate that the module instance  $M_\iota$  can make a transition to module instance  $M'_\iota$  by performing action  $\alpha$  and broadcasting event  $\alpha!$ .

*Module Instantiation:* A module instance can be created by the `create`( $s, m$ ) action, which instantiates a module specification  $s$  and assigns a system-generated name to it. This name is guaranteed to be unique within the multi-agent system. Then, the system-generated identifier of the module instance gets unified with the variable  $m$ .

$$\frac{t = \text{genid}(\mathcal{M}) \quad (M_\iota, \omega, \epsilon, v) \in \mathcal{M} \quad M_\iota \xrightarrow{\text{create}(s, m)!} M'_\iota[m/t]}{\langle \mathcal{M}, \mathcal{A} \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A} \rangle}$$

where  $\mathcal{M}' = (\mathcal{M} \setminus \{(M_\iota, \omega, \epsilon, v)\}) \cup \{(M'_\iota \tau, \omega, \epsilon, v), (M_t, \perp, \emptyset, \emptyset)\}$ .  $M_t$  is the initial mental state of module instance  $t$  as specified in module specification  $s$ , the function  $t = \text{genid}(\mathcal{M})$  generates a unique identifier  $t$  such that  $\neg \exists \omega', \epsilon', v' : (M_t, \omega', \epsilon', v') \in \mathcal{M}$ . We assume  $M'_\iota \tau$  to be the same as  $M_\iota$  except that the `create` action has been processed and the substitution  $\tau = [m/t]$  has been applied to the plan of  $M_\iota$  to which the `create` action belongs.

The action `release`( $m$ ) removes the module instance  $m$  from the multi-agent system. The release action only succeeds if the module instance  $m$  is inactive. If the action is applied successfully, the mental state of module instance  $m$  (i.e. its local belief base, goal base, plan base etc.) will be discarded.

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

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

*Module Activation:* For successful module activation of module instance  $m$  and all its descendants, all descendants must be inactive module instances forming a tree, i.e. there is only one path from  $m$  to all of its descendants. To ensure that this is the case, we introduce the *inact-tree* predicate, where *inact* stands for inactive.

$$\begin{aligned} \text{inact-tree}(m) &\Leftrightarrow \mathbb{D}_m \cap \mathbb{A} = \emptyset \quad \& \\ &\forall m_1, m_2 \in \mathbb{D}_m : m_1 \neq m_2 \Rightarrow \\ &(v_{m_1} \cup \epsilon_{m_1}) \cap (v_{m_2} \cup \epsilon_{m_2}) = \emptyset \end{aligned}$$

The `include`( $m, \varphi$ ) action, performed by module instance  $\iota$ , adds the identifier of module instance  $m$  to a set of included module instances of module instance  $\iota$  and assigns the deactivation condition  $\varphi$  to it. The action succeeds if module instance  $m$  exists in the multi-agent system and if the module instance and all its descendants form a tree of inactive module instances. Note that since we assume beliefs and goals to be composed only of positive atoms, the belief/goal bases of two module instances cannot be inconsistent.

$$\frac{(M_l, \omega, \epsilon, v) \in \mathcal{M} \quad M_l \xrightarrow{\text{include}(m, \varphi)!} M'_l}{(M_m, \omega', \epsilon', v') \in \mathcal{M} \quad \text{inact-tree}(m)} \frac{}{\langle \mathcal{M}, \mathcal{A} \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A} \rangle}$$

where  $\mathcal{M}' = (\mathcal{M} \setminus \{(M_l, \omega, \epsilon, v), (M_m, \omega', \epsilon', v')\}) \cup \{(M'_l, \omega, \epsilon, v \cup \{m\}), (M_m, \varphi, \epsilon', v')\}$ .

The  $\text{seclude}(m, \varphi)$  action, performed by module instance  $\iota$ , adds the identifier of module instance  $m$  to the set of secluded module instances of module instance  $\iota$  and assigns the deactivation condition  $\varphi$  to it. The action succeeds if module instance  $m$  exists in the multi-agent system and if the module instance  $m$  and all its descendants form a tree of inactive module instances.

$$\frac{(M_l, \omega, \epsilon, v) \in \mathcal{M} \quad M_l \xrightarrow{\text{seclude}(m, \varphi)!} M'_l}{(M_m, \omega', \epsilon', v') \in \mathcal{M} \quad \text{inact-tree}(m)} \frac{}{\langle \mathcal{M}, \mathcal{A} \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A} \rangle}$$

where  $\mathcal{M}' = (\mathcal{M} \setminus \{(M_l, \omega, \epsilon, v), (M_m, \omega', \epsilon', v')\}) \cup \{(M'_l, \omega, \epsilon \cup \{m\}, v), (M_m, \varphi, \epsilon', v')\}$ .

A module instance  $m$  is deactivated 1) when its parent module instance  $\iota$  executes the action  $\text{exclude}(m)$ , or 2) when its deactivation condition  $\omega$  is satisfied. Deactivation of module instance  $m$  removes the identifier  $m$  from the set of included module instances  $v$  and the set of secluded module instances  $\epsilon$  of module instance  $\iota$ . Note that since a module instance cannot be both included and secluded at the same time, the identifier of module instance  $m$  can only be a member of one of the sets. When deactivated, module instance  $m$  and all its descendants remain inactive in the multi-agent system. The following transition rule specifies module instance deactivation by means of the  $\text{exclude}$  action.

$$\frac{(M_l, \omega, \epsilon, v) \in \mathcal{M} \quad M_l \xrightarrow{\text{exclude}(m)!} M'_l \quad m \in v \cup \epsilon}{\langle \mathcal{M}, \mathcal{A} \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A} \rangle}$$

where  $\mathcal{M}' = (\mathcal{M} \setminus \{(M_l, \omega, \epsilon, v)\}) \cup \{(M'_l, \omega, \epsilon \setminus \{m\}, v \setminus \{m\})\}$ .

The following transition rule specifies module instance deactivation by satisfaction of deactivation condition  $\omega$ . This type of deactivation excludes a module instance  $m$  as soon as the deactivation condition  $\omega$  is satisfied by its belief and goal base.

$$\frac{(M_m, \omega, \epsilon, v) \in \mathcal{M} \quad (\sigma_m^*, \gamma_m^*) \models_t \omega}{(M_l, \omega', \epsilon', v') \in \mathcal{M} \quad m \in \epsilon' \cup v'} \frac{}{\langle \mathcal{M}, \mathcal{A} \rangle \longrightarrow \langle \mathcal{M}', \mathcal{A} \rangle}$$

where  $\mathcal{M}' = (\mathcal{M} \setminus \{(M_l, \omega', \epsilon', v')\}) \cup \{(M_l, \omega', \epsilon' \setminus \{m\}, v' \setminus \{m\})\}$ ,  $\sigma_m^*$  stands for the extended belief base of module instance  $m$ , and  $\gamma_m^*$  stands for the extended goal base of module instance  $m$ . These two concepts are explained later in this section.

The entailment relation  $\models_t$ , which evaluates deactivation conditions of the form  $B(\varphi)$  or  $G(\psi)$  with respect to belief and goal bases  $(\sigma, \gamma)$ , is defined as follows.

- $(\sigma, \gamma) \models_t B(\phi) \tau \Leftrightarrow \sigma \models \phi \tau$
- $(\sigma, \gamma) \models_t G(\psi) \tau \Leftrightarrow \gamma \models_g \psi \tau$

## Belief/Goal Sharing

**Module Interface:** For each module instance, a set of interface entries can be specified to control which parts of their local belief and goal bases will be shared with the other module instances in the sharing scope. An interface entry is an atomic formula, typically containing variables. The set of all interface entries specified in a module instance  $m$  is denoted by  $\mathcal{I}_m$ . The functions provided below are used to formalise the beliefs and goals that will pass through a module interface.

The function  $I_m(\varphi)$  determines whether the atomic formula  $\varphi$  is interfaced by module instance  $m$ :

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

The function  $I_m(S)$  returns the subset of atomic formulas interfaced by module instance  $m$  from the set of atomic formulas  $S$ :

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

The interface of a module instance can be used to determine which beliefs in its belief base will be interfaced. The function  $I_m^B(\sigma)$  returns the set of atomic facts entailed by belief base  $\sigma$  that are interfaced by module instance  $m$ :

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

Likewise, the interface of a module instance can be used to determine which goals (or subgoals) will be interfaced. First, we will define how individual goals are interfaced. Recall that we assume that an individual goal  $\gamma_i$  is represented as a conjunction of atomic facts. The function  $I_m^g(\gamma_i)$  returns the subgoal of goal  $\gamma_i$  interfaced by module instance  $m$ :

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

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

We can now define a function  $I_m^G(\gamma)$  which operates on an entire goal base. We assume that a goal base  $\gamma$  is modelled as a set of individual goals  $\gamma = \{\gamma_0, \dots, \gamma_i, \dots, \gamma_n\}$ . The function  $I_m^G(\gamma)$  returns a set containing all subgoals from goal base  $\gamma$  interfaced by  $m$ .

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

**Sharing Scope:** The sharing of beliefs and goals only takes place between module instances within one sharing scope. An agent's main module instance creates a new sharing scope, and each included module instance becomes part of the sharing scope of its parent. Each secluded module instance, in contrast, establishes a new sharing scope. A module instance that established a new sharing scope (i.e. a main module instance or

a secluded module instance) is called a sharing scope root. The function  $ss\text{-root}(m)$  returns the predecessor that established the sharing scope to which module instance  $m$  belongs.

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

In this definition,  $m'$  is either a module instance which is a secluded child of another module instance  $m^*$  (i.e. the root of a sharing scope) or the main module instance of an agent. Moreover,  $m'$  is a descendant of  $m$  through  $v$ -links.

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

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

*Global Belief Base:* The global belief base of a sharing scope refers to the aggregation of the interfaced local beliefs from all module instances in that sharing scope. Since global beliefs can only be accessed by module instances in the given sharing scope, we define a global belief base relative to a module instance that can perform queries on it. The global belief base of the sharing scope of module instance  $m$  contains the interfaced local beliefs of all module instances within that sharing scope. Let the local belief base of module instance  $m'$  be denoted by  $\sigma_{m'}$ . The global belief base of module instance  $m$ , denoted by  $\bar{\sigma}_m$ , is defined as follows.

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

When a belief query is performed in a given module instance, not only the local belief base of that module instance is consulted, but also the global beliefs that are interfaced (imported) by the module instance. The local belief base of a module instance combined with the beliefs that it imports through its interface constitute the extended belief base of that module instance. We define the extended belief base of module instance  $m$ , denoted by  $\sigma_m^*$ , as its local belief base augmented with the global beliefs it interfaces.

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

where  $\sigma_m$  is the local belief base of module instance  $m$ , and  $\sigma \cup S$  denotes the operation that extends belief base  $\sigma$  with the atomic facts from set  $S$ . A module instance  $m$  will always perform its belief queries on its extended belief base  $\sigma_m^*$ .

*Global Goal Base:* The global goal base  $\bar{\gamma}_m$  is defined analogously to the global belief base as an aggregation of the interfaced local goals from all module instances in the sharing scope of module instance  $m$ . The local goal base of module instance  $m'$  is denoted by  $\gamma_{m'}$ .

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

Analogously to the extended belief base, we define the extended goal base of a module instance  $m$ , denoted by  $\gamma_m^*$ ,

as its local goal base augmented with the global goals it interfaces.

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

where  $\gamma_m$  is the local belief base of module instance  $m$ , and  $\gamma \cup S$  denotes the operation that extends goal base  $\gamma$  with goals from set  $S$ . A module instance  $m$  will always perform goal queries on its extended goal base  $\gamma_m^*$ .

## V. PROPERTIES

In this section we describe several properties of the proposed modular system. Proofs are omitted due to space limitations. All properties assume a given multi-agent system configuration  $\langle \mathcal{M}, \mathcal{A} \rangle$ . The set of all atomic facts is denoted by  $\Phi$ .

*Belief Sharing:* If two module instances from one sharing scope interface atom  $\varphi$ , their beliefs with regards to  $\varphi$  will be consistent.

$$\forall m_1 \in \mathbb{M} \forall m_2 \in \mathbb{S}_{m_1} \forall \varphi \in \Phi : \\ (I_{m_1}(\varphi) \ \& \ I_{m_2}(\varphi)) \Rightarrow ((\sigma_{m_1}^* \models \varphi) \Leftrightarrow (\sigma_{m_2}^* \models \varphi))$$

This property follows from the definition of the extended belief base and the fact that we assume a belief language without negative atoms. As soon as a module that interfaces  $\varphi$  starts believing  $\varphi$ , the belief gets propagated to all other module instances in the sharing scope that interface  $\varphi$ . Thus, the module instances in a sharing scope interfacing  $\varphi$  either all believe  $\varphi$ , or none of them believes  $\varphi$ .

*Encapsulation:* If a module instance specifies its interface to be empty, all facts it believes originate from its local belief base.

$$\forall m \in \mathbb{M} \forall \varphi \in \Phi : (\mathcal{I}_m = \emptyset \ \& \ \sigma_m^* \models \varphi) \Rightarrow (\sigma_m \models \varphi)$$

This property follows from the definition of the extended belief base, which combines local beliefs with the global beliefs imported through a module instance's interface. If the interface is empty, no global beliefs can be imported. And therefore, all beliefs in the extended belief base of a module instance must originate from its local belief base.

*Rationality Axiom Preserved:* Non-modular BDI languages maintain consistency with respect to the rationality axiom (goals that are believed to be achieved are dropped). Our proposal extending a BDI language with belief/goal sharing modularity maintains such consistency as well.

$$\forall m \in \mathbb{M} : (\gamma_m^* \models \varphi) \Rightarrow (\sigma_m^* \not\models \varphi)$$

This property holds since in our framework it is not possible to interface a belief without interfacing the respective goal as well (and vice versa).

*Agent's Mental State:* In the proposed framework, an agent's mental state is a tree of active module instances.

$$\forall a \in \mathcal{A} : \mathbb{D}_a \subseteq \mathbb{A} \ \& \\ \forall m_1, m_2 \in \mathbb{D}_a : m_1 \neq m_2 \Rightarrow \\ (v_{m_1} \cup \epsilon_{m_1}) \cap (v_{m_2} \cup \epsilon_{m_2}) = \emptyset$$

This property follows from the semantics of the `include` and `seclude` actions and the definition of the set of active modules. A module instance  $m$  becomes a child of

another module instance  $m'$  by means of module activation ( $\text{include}(m)$  or  $\text{seclude}(m)$ ). The activation only succeeds if all descendants of  $m$  are inactive and form a tree (see the definition of  $\text{inact-tree}(m)$ ). Since  $m'$  must be active, i.e. a descendant of one of the main module instances, in order to perform actions,  $m$  and its descendants will necessarily also be descendants of the same main module instance and thus also active.

## VI. CONCLUSION

In this paper we have introduced a belief/goal sharing modularisation framework suitable for BDI-based agent programming languages with declarative goals. The proposed framework provides constructs for program decomposition that are closer to mainstream programming languages than previous proposals. Yet, it stays conceptually agent-oriented. We presented the concept using an example, defined the semantics of the module-related constructs and analysed properties of the proposed system. Furthermore, we have implemented the proposed framework and incorporated it into the interpreter of 2APL programming language [16]. We do not discuss the implementation here due to space limitations, but we would like to remark that the framework can be implemented in a computationally efficient way.

We consider a module instance as a cluster of cognitive components typically focusing on one well-defined task. The proposed framework thus allows for the *separation of concerns*. A module specifying a solution to some recurring problem is *reusable* in different programs. Moreover, as a module instance is considered as a black-box with a limited public interface, the modular system serves as a useful tool for the *encapsulation* of program logic. The modular system combines ideas from object and agent orientation. On the one hand, we instantiate and manipulate module instances in a similar fashion as objects in object oriented languages; on the other hand, we make use of mentalistic notions such as beliefs or goals to realize the interaction between module instances. We therefore expect that programmers acquainted with some of the main-stream programming languages will find the proposed framework *familiar*, but we also believe that it will help programmers to decompose their code in a conceptually clean, *agent-oriented* manner.

We would like to emphasize that we do not claim that the presented framework allows implementation of applications that could not be implemented in other approaches. Therefore, it would not be realistic to require the presentation of a killer application showing the benefit of the proposed framework. Instead, we illustrated the added value of this approach by means of a simple but characteristic example which shows that the framework respects the fundamental programming principles mentioned in the previous paragraph.

The framework can be a starting point for interesting follow up work. First, we envision an extension to our modular system that would allow the construction of evolving, self-modifying agents. Currently, we provide means for updating of the belief and goal bases of a module instance at runtime. By adding a set

of actions for updating its rule base, an agent would be able to construct a module instance completely from scratch. We can even imagine applying genetic programming algorithms [17] to evolve the internals of a module instance automatically.

A second interesting extension is to support the categorisation of interfaced beliefs and goals with regards to an existing ontology. Currently, it is possible that two module specifications assign an identical name to an interfaced belief/goal, although they represent a different concept. We envision a mechanism that links each interfaced belief/goal to a particular ontology. Each interface entry would be assigned a semantic meaning, e.g. from an OWL ontology. Such an extension would make it possible to detect shared beliefs and goals that match syntactically, but refer to different concepts.

## REFERENCES

- [1] Y. Shoham and Others, "Agent-Oriented Programming," *Artificial Intelligence*, vol. 60, no. 1, pp. 51–92, 1993.
- [2] M. E. Bratman, *Intention, Plans, and Practical Reason*. Cambridge, MA: Harvard University Press, 1987.
- [3] A. S. Rao and M. P. Georgeff, "Modeling rational agents within a BDI-architecture," 1991.
- [4] M. Dastani, "2APL: a practical agent programming language," *Autonomous Agents and Multi-Agent Systems*, vol. 16, no. 3, pp. 214–248, 2008.
- [5] K. V. Hindriks, F. S. de Boer, W. V. D. Hoek, and J.-J. C. Meyer, "Agent programming with declarative goals," in *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop, ATAL 2000*. Springer, 2000.
- [6] P. Busetta, R. Rönquist, A. Hodgson, and A. Lucas, "Jack intelligent agents - components for intelligent agents in Java," 1999. [Online]. Available: <http://eprints.agentlink.org/4671/>
- [7] A. Pokahr, L. Braubach, and W. Lamersdorf, "Jadex: A BDI reasoning engine," in *Multi-Agent Programming*. Springer Science+Business Media Inc., USA, 2005, book chapter.
- [8] R. H. Bordini, M. Wooldridge, and J. F. Hübner, *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.
- [9] M. Dastani, C. P. Mol, and B. R. Steunebrink, "Modularity in BDI-based agent programming languages," in *WI-IAT '09 Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, 2009.
- [10] K. Hindriks, "Modules as policy-based intentions: Modular agent programming in goal," in *In Proceedings of the International Workshop on Programming Multi-Agent Systems (PROMAS '07), number 4908 in LNAI*. Springer, 2008.
- [11] N. Madden and B. Logan, "Modularity and compositionality in Jason," in *Proceedings of International Workshop Programming Multi-Agent Systems (ProMAS 2009)*. Springer, 2009.
- [12] P. Novák and J. Dix, "Modular BDI architecture," in *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM, 2006.
- [13] P. Busetta, N. Howden, R. Rönquist, and A. Hodgson, "Structuring BDI agents in functional clusters," in *ATAL '99: 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL)*. London, UK: Springer-Verlag, 2000, pp. 277–289.
- [14] L. Braubach, A. Pokahr, and W. Lamersdorf, "Extending the capability concept for flexible BDI agent modularization," in *The 3rd International Workshop on Programming Multiagent Systems (PROMAS 2005), in conjunction with AAMAS 2005*. Springer Verlag, Berlin, Heidelberg, 2006.
- [15] M. B. van Riemsdijk, M. Dastani, J.-J. C. Meyer, and F. S. de Boer, "Goal-oriented modularity in agent programming," in *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. New York, USA: ACM, 2006.
- [16] <http://apapl.sourceforge.net/>.
- [17] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.