Goal-Oriented Modularity in Agent Programming

M. Birna van Riemsdijk[1] Mehdi Dastani[1] John-Jules Ch. Meyer[1] Frank S. de Boer[1,2]
Utrecht University[1]
CWI, Amsterdam[2]
The Netherlands
[birna | mehdi | jj | frankb]@cs.uu.nl

ABSTRACT

Modularization is widely recognized as a central issue in software engineering. In this paper we address the issue of modularization in cognitive agent programming languages. We discuss existing approaches to modularity in cognitive agent programming. Then, we propose a new kind of modularity, i.e., goal-oriented modularity, which takes the goals of an agent as the basis for modularization. Further, we present a formal semantics of goal-oriented modularity in the context of the 3APL agent programming language.

Categories and Subject Descriptors

I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent agents, languages and structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.2 [**Software Engineering**]

General Terms

Theory, Languages

Keywords

Agent programming languages, modularity, semantics, declarative goals

1. INTRODUCTION

Modularization is widely recognized as a central issue in software engineering [11, 9, 2]. A system which is composed of modules, i.e., relatively independent units of functionality, is called modular. A programming language which adheres to this principle of modularization supports the decomposition of a system into modules. The principle lies, e.g., at the basis of procedural programming and object-oriented programming, in which respectively (libraries of) procedures and classes form the modules.

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.

An important advantage of modularity in a programming language is that it can increase the *understandability* of the programs written in the language. The reason is that modules can be separately understood, i.e., the programmer does not have to oversee the entire workings of the system when considering a certain module. Further, modularization enables *reuse* of software, since modules can potentially be used in different programs or parts of a single program. Finally, we mention an important principle which any kind of modularity should stick to, namely the principle of *information hiding*. This means that information about a module should be private to the module, i.e., not accessible from outside, unless it is specifically declared as public. The amount of information declared public should typically be relatively small. The idea behind information hiding is that a module can be changed (or at least the non-public part of a module), without affecting other modules.

In this paper we address the issue of modularization in *cognitive agent programming languages*. Cognitive agents are agents endowed with high-level mental attitudes such as beliefs, goals, plans, etc. Several programming languages and platforms have been introduced to program these agents, such as 3APL [10, 6], AgentSpeak(L) [14, 3], JACK™ [20], Jadex [13], etc. Some of these languages incorporate support for modularization, which we discuss in some detail in section 2.1.

Our contribution is the proposal of a new kind of modularity, i.e., *goal-oriented modularity*, which is, as we will argue, particularly suited for agent programming languages (section 2). Further, we present a *formalization* of goal-oriented modularity in the context of the 3APL programming language (section 3). We conclude the paper with directions for future research in section 4.

2. GOAL-ORIENTED MODULARITY

In this section, we explain the general idea of goal-oriented modularity. First, we discuss which kinds of modularity are present in today's cognitive agent programming languages and platforms (section 2.1). Then, we explain the general idea of goal-oriented modularity (section 2.2).

As for programming in general, modularization is also an important issue for agent programming. One could argue that the agent paradigm provides inherent support for modularity, since a complex problem can be broken down and solved by a team of autonomous agents. Constructing a team of agents to solve a problem rather than creating a single more complex agent, might however not always be the appropriate approach. The team approach will likely gener-

ate significant communication overhead, and having several independent agents can make it difficult to handle problems that require global reasoning [5, 4]. Following [5, 4], we thus argue that modularization is important also at the level of individual agents.

2.1 Related Work

With regard to cognitive agent programming languages such as 3APL and AgentSpeak(L), one could argue that these languages support modularization: an agent is typically composed of a number of components such as a belief base, a plan or intention base, a plan library, a goal or event base, etc. These components however do not provide the appropriate modularization, since their workings are closely intertwined and therefore they cannot be considered as relatively independent units of functionality. Cognitive agent programming languages thus need more support for structuring the internals of an agent, in order for them to deserve the predicate "modular".

One possible approach to addressing this issue of modularity of cognitive agent programming languages has been proposed by Busetta et al. in [5]. In that paper, the notion of *capability* was introduced and its implementation in the JACK cognitive agent programming language was described.

JACK extends the Java™ programming language in several ways, such as by introducing constructs for declaring cognitive agent notions like beliefs, events, plans, etc. Events are used to model messages being received, new goals being adopted, and information being received from the environment. Plans are used to handle events, i.e., if an event is posted, the reasoning engine tries to find an appropriate plan which has this event as its so-called triggering condition.

A capability in JACK is a cluster of components of a cognitive agent, i.e., it encapsulates beliefs, events (either posted or handled by the capability), and plans. Examples of capabilities given in [5] are a buyer and seller capability, clustering functionality for buyer and seller agents, respectively. A capability can import another capability, in which case the latter becomes a sub-capability of the former. Using the importation mechanism for capabilities, the beliefs of a capability can also be used by its super-capability, that is, if they are explicitly imported by the latter. Also, the beliefs of a capability can be used by its sub-capabilities if they are explicitly declared as exported (and if they are also imported by the sub-capabilities). For events, a similar mechanism exists, by means of which events posted from one capability can also be handled by plans of its sub- and possibly super-capabilities.

The notion of capability as used in JACK has been extended in the context of the Jadex platform [4]. Jadex is a cognitive reasoning engine which is built on top of the Jade [1] agent platform. A Jadex agent has beliefs, goals, plans, and events. Like capabilities in JACK, a Jadex capability clusters a set of beliefs, goals, plans, and events. Its most important difference with the notion of capability as used in JACK, is the fact that a general import/export mechanism is introduced for all kinds of elements of a capability, i.e., the mechanism is the same for beliefs, events, etc.

Another approach which could be viewed as addressing the issue of modularity in cognitive agent programming languages, has been proposed in the context of 3APL in [7]. In that paper, a formalization of the notion of a role is given. Similar to capabilities, a role clusters beliefs, goals, plans, and reasoning rules. The usage of roles at run-time however differs from that of capabilities. In the cited paper, a role can be enacted and deacted at run-time, which is specified at the level of the 3APL reasoning engine or deliberation cycle. If a role is enacted, the agent pursues the goals of the role, using the plans and reasoning rules of the role.[1] Further, the agent has a single belief base, and if a role is enacted, the beliefs associated with the role are added to the agent's beliefs. This is in contrast with JACK and Jadex where beliefs are distributed over capabilities, and can only be used in other capabilities if explicitly imported and exported. Also, only one role at the time can be active. This is in contrast with the way capabilities are used, since a JACK or Jadex agent can in principle use any of its capabilities at all times.

2.2 Our Proposal

While the approaches to modularization as described in section 2.1 are interesting in their own right, we propose an alternative which we argue to be particularly suited for cognitive agent programming languages. As the name suggests, goal-oriented modularity takes the goals of an agent as the basis for modularization. The idea is, that *modules encapsulate the information on how to achieve a goal*, or a set of (related) goals. That is, modules contain information about the plans that can be used to achieve a (set of) goal(s). At run-time, the agent can then dispatch a goal to a module, which, broadly speaking, tries to achieve the dispatched goal using the information about plans contained in the module.

This mechanism of dispatching a goal to a module can be used for an agent's top-level goals[2], but also for the *subgoals* as occurring in the plans of an agent. Plans are often built from actions which can be executed directly, and subgoals which represent a state that is to be achieved before the agent can continue the execution of the rest of the plan. An agent can for example have the plan to take the bus into town, to achieve the goal of having bought a birthday cake, and then to eat the cake.[3] This goal of buying a birthday cake will have to be fulfilled by executing in turn an appropriate plan of for example which shops to go to, paying for the cake, etc., before the agent can execute the action of eating the cake.

Before continuing, we remark that goals and subgoals in this paper are *declarative* goals, which means that they describe a state that is to be reached. This is in contrast with procedural goals, which are directly linked to courses of action. We refer to [21] for a discussion on declarative and procedural goals in general, and to [19, 16] for more background on declarative goals. Further, [17] explores semantics of subgoals, and relates declarative and procedural interpretations of subgoals.

Returning to our treatment of goal-oriented modularity, the idea is thus that agents try to achieve subgoals of a plan by dispatching the subgoal to an appropriate module, i.e., by *calling* a module. The module should then define the plans that can be used for achieving the (sub)goal. If a module is

[1] We simplify somewhat, since the details are not relevant for the purpose of this paper.

[2] Top-level goals are goals that the agent, e.g., has from start-up, or that it for example has adopted because of requests from other agents, etc.

[3] Assuming that both taking the bus into town and eating cake are actions that can be executed directly.

called for a goal, these plans are tried one by one until either the goal is achieved, or all plans have been tried. Control then returns to the plan from which the module was called. Depending on whether the subgoal is achieved or not upon returning from the module, the plan respectively continues execution, or fails. If the plan fails, another plan is selected (if it exists), for achieving the goal for which the failed plan was selected, etc.

2.3 Discussion

An advantage of our proposal is the *flexible* agent behavior with respect to handling plan failure, which comes with the usage of declarative goals. As argued in [21, 16], the usage of declarative goals facilitates a decoupling of plan execution and goal achievement. If a plan fails, the goal that was to be achieved by the plan remains a goal of the agent, and the agent can select a different plan to try to achieve the goal. While these ideas regarding declarative goals are not new, we contribute by proposing to use modules to *encapsulate* this mechanism for achieving goals by trying different plans. We thus exploit the advantages of declarative goals for obtaining modularization. Since in our view goals, proactiveness and flexible behavior are at the heart of (cognitive) agenthood, we argue that goal-oriented modularity, which builds on these notions, is a kind of modularity *fitting* for cognitive agents.

Comparing goal-oriented modularity with capabilities, we point out two major differences. Firstly, in the case of capabilities, there is no notion of *calling* a capability for a subgoal, thereby passing control to another capability. An event (or subgoal) posted from the plan of one capability, will be handled by the plans of this capability itself. That is, unless the capability imports other capabilities, in which case the plans of these other capabilities are *added* to the set of plans considered for handling the posted event (given appropriate import and export declarations of events). This is thus in contrast with goal-oriented modularity, where, in case of calling a module, *only* the plans of the called module are considered. These plans are not added to, e.g., some other set of plans, thereby preventing possibly unforeseen interactions between these plans. One could thus consider the idea of goal-oriented modularity to provide a higher degree of modularity or encapsulation of behavior at run-time, compared with the way in which capabilities are used. As we will explain in section 3, this is especially advantageous in the case of 3APL.

Secondly, modules in goal-oriented modularity contain only information about the plans which can be used to achieve certain goals. This is in contrast with capabilities, which can also contain beliefs. The idea for goal-oriented modularity is that the agent has one global belief base, rather than defining beliefs inside modules. When using capabilities, beliefs can by contrast be distributed over these capabilities, and only the beliefs of a certain capability can be accessed from this capability.

While from a software engineering perspective it might be convenient to be able to define beliefs inside capabilities (or modules), it can be considered less intuitive from a conceptual point of view. When testing, e.g., from within a plan whether the agent believes something, one could argue that the agent would have to consider *all* of its beliefs, rather than just the ones available in the capability. Also, if logical reasoning is involved as in the case of 3APL, it is more intuitive to let an agent have just one belief base. Consider for example that the formula p is in the belief base of one module, and that $p \rightarrow q$ is in the belief base of another. When testing whether q holds from the latter module, one would probably want the test to succeed.

Nevertheless, in JACK and Jadex, beliefs can be used in other capabilities if they are imported and exported in appropriate ways. Also, beliefs (or beliefsets) in those languages are effectively databases which store elements representing the beliefs of the agent. The definition of beliefs inside a capability could thus be viewed as the specification of a *part* of the larger database (or set of databases) comprising the total set of beliefs of the agent. From within a capability, an agent can then only refer to the part of its beliefs defined in this capability. It is then up to the programmer to make sure that the beliefs of a capability are the only ones relevant for the plans of this capability. The possibility of storing beliefs inside modules in a way which is somewhere inbetween the current proposal and the way it is done for capabilities, is discussed in section 4.

Comparing goal-oriented modularity with the notion of roles as used in [7], we remark the following. As in the case of capabilities, roles can, in contrast with modules, not call each other. Also, beliefs can be part of the definition of a role. However, a role does not have its own beliefs once it is enacted at run-time. If a role is enacted, its beliefs are added to the global belief base of the agent. This is more in line with goal-oriented modularity, but it is in contrast with the way capabilities are used. Further, contrary to roles, modules do not have goals. That is, a goal can be dispatched to a module, but goals are not part of the definition of a module.

3. GOAL-ORIENTED MODULARITY IN 3APL

In this section, we make the idea of goal-oriented modularity as presented in section 2 precise. In particular, we present a formalization in the context of 3APL. Nevertheless, we stress that we consider the general idea of goal-oriented modularity to be suited for cognitive agent programming languages in general, rather than for 3APL only.

3.1 Syntax

We build on a single agent and propositional version of 3APL, which comes closest to the one presented in [19]. We refer to [6, 8] for first order, multi-agent, and otherwise extended versions. This simplified version of 3APL is sufficient to present the ideas of the formalization of goal-oriented modularity. We conjecture that extending the formalization to a richer version of 3APL will not give rise to fundamental difficulties.

A 3APL agent in [19] has beliefs, a plan, and goals. Beliefs represent the current state of the world and information internal to the agent. Goals represent the desired state of the world, and plans are the means to achieve the goals. Further, a 3APL agent has rules for selecting a plan to achieve a certain goal given a certain belief, and it has rules for revising its plan during execution. We use these ingredients for our formalization of goal-oriented modularity.

Throughout this paper, we assume a language of propositional logic $\mathcal{L}$ with negation and conjunction. The symbol $\models$ will be used to denote the standard entailment relation for

$\mathcal{L}$. Further, we assume a belief query language $\mathcal{L}_{\mathsf{B}}$ with typical element β. A belief query can be used to test whether the agent has a certain belief. We assume an entailment relation on belief bases and belief query formulas, denoted by $\models_{\mathcal{L}_{\mathsf{B}}}$.[4]

Below, we define the language of plans Plan. A plan is a sequence of basic actions and module calls. Informally, basic actions can change the beliefs of an agent if executed. A module call is of the form $m(\phi)$, where m is the name of a module (to be defined in definition 4), and ϕ is a propositional formula representing the goal which is dispatched to module m.

Further, we define an auxiliary set of plans Plan$'$ with the additional construct $m(\phi)\downarrow$. This construct is used in the semantic definitions for recording whether module m has already been called for goal ϕ (see section 3.2 for further explanation). It should not be used by the agent programmer for programming plans, which is why we define two different plan languages.

Definition 1. *(plan) Let* BasicAction *with typical element* a *be the set of basic actions, let* ModName *with typical element* m *be a set of module names, and let* $\phi \in \mathcal{L}$. *The set of plans* Plan *with typical element* π *is then defined as follows.*

$$\pi ::= a \mid m(\phi) \mid \pi_1; \pi_2$$

We use ϵ *to denote the empty plan and identify* $\epsilon;\pi$ *and* $\pi;\epsilon$ *with* π. *The set* Plan$'$ *is defined as follows.*

$$\pi ::= a \mid m(\phi) \mid m(\phi)\downarrow \mid \pi_1; \pi_2$$

This simple language of plans could be extended with, e.g., if-then-else and while constructs as was done in [19], but the language as given here suffices for the purpose of this paper.

3APL uses so-called plan generation rules for selecting an appropriate plan for a certain goal. A plan generation rule is of the form $\phi \mid \beta \Rightarrow \pi$. This rule represents that it is appropriate to select plan π for goal ϕ, if the agent believes β.[5]

Definition 2. *(plan generation rule) The set of plan generation rules* $\mathcal{R}_{\mathsf{PG}}$ *is defined as follows:* $\mathcal{R}_{\mathsf{PG}} = \{\phi \mid \beta \Rightarrow \pi \ : \ \phi \in \mathcal{L},\ \beta \in \mathcal{L}_{\mathsf{B}},\ \pi \in \mathsf{Plan}\}$.[6]

The other type of rule which comes with 3APL, is the rule for plan revision. Plan revision rules are used to revise an agent's plan during execution. A plan revision rule $\pi_h \mid \beta \rightsquigarrow \pi_b$ represents that in case the agent believes β, it can replace the plan π_h by the plan π_b.

Definition 3. *(plan revision rule) The set of plan revision rules* $\mathcal{R}_{\mathsf{PR}}$ *is defined as follows:* $\mathcal{R}_{\mathsf{PR}} = \{\pi_h \mid \beta \rightsquigarrow \pi_b \ : \ \beta \in \mathcal{L}_{\mathsf{B}},\ \pi_h, \pi_b \in \mathsf{Plan}\}$.

[4]See, e.g., [19] for an example of a belief query language, in which one can express queries such as $\mathbf{B}(p)$ and $\neg\mathbf{B}(p \wedge q)$ for testing whether the agent believes p or does not believe $p \wedge q$, respectively.

[5]Note that it is up to the programmer to specify appropriate plans for a certain goal. 3APL agents can thus be viewed as a kind of reactive planning agents.

[6]We use the notation $\{\ldots \ : \ \ldots\}$ instead of $\{\ldots \mid \ldots\}$ to define sets, to prevent confusing usage of the symbol $\mid$ in this definition and definition 3.

Plan generation rules capture the information about which plan can be selected for which goal, and plan revision rules can be used during the execution of a plan. These rules thus specify the information on how to achieve goals, and therefore we propose to have these rules make up a module, as specified below. It is important to remark that non-modular versions of 3APL have one set of plan generation rules, and one set of plan revision rules, rather than an encapsulation of these into modules.

Definition 4. *(module) A module is a tuple* $\langle m, \mathsf{PG}, \mathsf{PR}\rangle$, *consisting of a module name* m, *a finite set of plan generation rules* $\mathsf{PG} \subseteq \mathcal{R}_{\mathsf{PG}}$, *and a finite set of plan revision rules* $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$.

The mechanism of calling a module is formalized using the notion of a stack. This stack can be compared with the stack resulting from procedure calls in procedural programming, or method calls in object-oriented programming. During execution of the agent, a single stack is built (see definition 7). Each element of the stack represents, broadly speaking, a module call.

To be more specific, each element of the stack is of the form $(\phi, \pi, \mathsf{PG}, \mathsf{PR})$, where ϕ is the goal for which the module was called, π is the plan currently being executed in order to achieve ϕ, and PG and PR correspond with the plan generation and plan revision rules of the module which was called for achieving ϕ. Rather than using the name of the module to refer to the relevant plan generation and plan revision rules, we copy these rules into the stack element. The reason is that we want to remove plan generation rules if they are tried once. This will be explained further in section 3.2.

Definition 5. *(stack) The set of stacks* Stack *with typical element* S *to denote arbitrary stacks, and* s *to denote single elements of a stack, is defined as follows, where* $\phi \in \mathcal{L}$, $\pi \in \mathsf{Plan}'$, $\mathsf{PG} \subseteq \mathcal{R}_{\mathsf{PG}}$, *and* $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$.

$$\begin{array}{rcl} \mathrm{s} & ::= & (\phi, \pi, \mathsf{PG}, \mathsf{PR}) \\ \mathrm{S} & ::= & \mathrm{s} \mid \mathrm{s}.\mathrm{S} \end{array}$$

*E is used to denote the empty stack (or the empty stack element), and E.*S *is identified with* S.

Note that the plan π of a stack element is from the extended set of plans Plan$'$, since a stack is a run-time construct which is not specified by the programmer when programming an agent (see definition 6). The plan π might thus, in contrast with the plans of plan generation and plan revision rules, contain a construct of the form $m(\phi)\downarrow$. The empty stack E is introduced for technical convenience when defining the semantics in section 3.2. Stacks as used here differ from intention stacks of AgentSpeak(L), as the elements comprising the stacks are essentially different: stack elements in this paper correspond with module calls, whereas in AgentSpeak(L) they represent (parts of) plans or intentions. Like non-modular 3APL, AgentSpeak(L) has one large set of plans (corresponding with the rules of 3APL).

An agent, as defined below, consists of a belief base, a goal base, a set of modules, and a function which specifies how beliefs are updated if actions are executed. As in non-modular 3APL, the belief base σ is a consistent set of propositional formulas. The goal base γ is essentially also a set of propositional formulas, and forms the top-level goals

of the agent. In contrast with non-modular 3APL however, each goal is associated with a module which should be called for achieving the goal, i.e., goals are of the form $m(\phi)$. The set of modules Mod form the modules which can be called to achieve (sub)goals. Defining how beliefs are updated through action execution by assuming a function $\mathcal{T}$, is standard for the way 3APL's semantics is usually defined. This function is used for technical convenience. We assume that $\mathcal{T}$ maintains consistency of the belief base.

DEFINITION 6. *(agent) Let* $\Sigma = \{\sigma \mid \sigma \subseteq \mathcal{L},\ \sigma \not\models \bot\}$ *be the set of belief bases. An agent is a tuple* $\langle \sigma, \gamma, \mathsf{Mod}, \mathcal{T}\rangle$ *where* $\sigma \in \Sigma$ *is the belief base,* $\gamma \subseteq \{m(\phi) \mid m \in \mathsf{ModName}, \phi \in \mathcal{L}\}$ *is the initial goal base, and* Mod *is a set of modules such that each module in this set has a distinct name.* $\mathcal{T}$ *is a partial function of type* $(\mathsf{BasicAction} \times \Sigma) \to \Sigma$ *and specifies the belief update resulting from the execution of basic actions.*

The notion of a configuration, as defined below, is used to represent the state of an agent at each point during computation. It consists of the elements which may change during execution of the agent, i.e., it consists of a belief base, a goal base, and a stack. Note that an agent initially has an empty stack.

DEFINITION 7. *(configuration) A configuration is a tuple* $\langle \sigma, \gamma, \mathrm{S}\rangle$ *where* $\sigma \in \Sigma$, $\gamma \subseteq \{m(\phi) \mid m \in \mathsf{ModName},\ \phi \in \mathcal{L}\}$, *and* $\mathrm{S} \in \mathsf{Stack}$. *If* $\langle \sigma, \gamma, \mathsf{Mod}, \mathcal{T}\rangle$ *is an agent, then* $\langle \sigma, \gamma, E\rangle$ *is the initial configuration of the agent.*

3.2 Semantics

The semantics of modular 3APL agents is defined by means of a transition system [12]. A transition system for a programming language consists of a set of axioms and transition rules for deriving transitions for this language. A transition is a transformation of one configuration into another and it corresponds to a single computation step. In the transition rules below, we assume an agent with a set of modules Mod, and a belief update function $\mathcal{T}$.

The first transition rule specifies how a transition for a composed stack can be derived, given a transition for a single stack element. It specifies that only the top element of a stack can be transformed or executed.[7]

DEFINITION 8. *(stack execution) Let* $s \neq E$.

$$\frac{\langle \sigma, \gamma, \mathrm{s}\rangle \to \langle \sigma', \gamma', \mathrm{S}'\rangle}{\langle \sigma, \gamma, \mathrm{s.S}\rangle \to \langle \sigma', \gamma', \mathrm{S}'.\mathrm{S}\rangle}$$

In the transition rule for stack execution, we specify that s cannot be the empty stack. The reason is related to the transition rule for stack initialization of definition 9 below. In that rule, we want to specify that a stack initialization transition can only be derived if the current stack is empty. We however do not want to use that rule in combination with the rule for stack execution, since that would result in the possibility of deriving an "initialization" transition, even if the current stack is not empty.

An initialization transition can thus only be derived if the current stack is empty. The idea of initialization is that a (top-level) goal $m(\phi)$ from the goal base is (randomly) selected, and then the module m is called with the goal ϕ. The resulting stack is then of the form $(\phi, \epsilon, \mathsf{PG}, \mathsf{PR})$, where PG and PR are the plan generation and plan revision rules of m. Note that the plan of the resulting stack element is empty. The idea is that the plan generation rules of the module should now be used to generate an appropriate plan.

DEFINITION 9. *(initialization of stack)*

$$\frac{m(\phi) \in \gamma \qquad \langle m, \mathsf{PG}, \mathsf{PR}\rangle \in \mathsf{Mod}}{\langle \sigma, \gamma, E\rangle \to \langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle}$$

Note that a consequence of this way of defining the goal base is that multiple goals from the goal base cannot be dispatched to the same module in one initialization. Thinking about alternative ways to define the goal base which might allow this, is left for future research.

The following transition rule is the rule for plan generation, defining when a plan generation rule can be applied. A prerequisite for applying a plan generation rule is that the plan of the relevant stack element is empty. A rule $\phi' \mid \beta \Rightarrow \pi$ from PG can then be applied if β is true, ϕ' follows from ϕ (i.e., the goal for which the module was called), and ϕ' is not believed to be achieved. Further, for a plan generation rule to be applicable, the goal ϕ' should not already be achieved, since there is no need to generate a plan for such a goal. Note that since ϕ' follows from ϕ, it is the case that ϕ is also not reached, if ϕ' is not reached. With respect to the second condition for plan generation rule application, we remark that a plan generation rule with goal antecedent p (i.e., $\phi' = p$) can thus be applied if the agent has, e.g., a goal $p \wedge q$ (i.e., $\phi = p \wedge q$). This interpretation of plan generation rules is standard for 3APL. It is convenient when programming, since plan generation rules can be specified for parts of a composed goal.

If a plan generation rule is applied, the plan π in its consequent becomes the plan of the resulting stack element. Further, the applied plan generation rule is removed from the set of plan generation rules of the stack element.

DEFINITION 10. *(plan generation)*

$$\frac{\phi' \mid \beta \Rightarrow \pi \in \mathsf{PG} \qquad \sigma \models_{\mathcal{L}_\mathsf{B}} \beta \qquad \phi \models \phi' \qquad \sigma \not\models \phi'}{\langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle \to \langle \sigma, \gamma, (\phi, \pi, \mathsf{PG}', \mathsf{PR})\rangle}$$

where $\mathsf{PG}' = \mathsf{PG} \setminus \{\phi' \mid \beta \Rightarrow \pi\}$

The reason for removing plan generation rules is that we do not want the agent to try the same plan generation rule twice, to achieve a certain goal. Otherwise, the agent could get stuck "inside" a module trying to achieve a subgoal, if all its plans keep failing to reach the goal. The idea is that modules contain various plans for achieving a certain goal. If the agent cannot reach a certain subgoal of a plan with the designated module, the agent should thus at a certain point give up trying to reach the subgoal. It should just try another plan with possibly different subgoals. Wanting to remove plan generation rules of a stack element is the reason that we copy the rules into stack elements, rather than just referring to the rules of a module using the name of the module. If we would extend this semantics to a first order version, we would have to record which instances of a plan generation rule are tried, rather than just removing the rule. This mechanism is comparable to the mechanism

[7] For technical convenience, we overload the "." operator in definition 8. We use it to "push" a stack onto a stack, rather than to push a single stack element onto a stack, as it was, strictly speaking, defined in definition 5.

for selecting plans for subgoals in JACK. In JACK it is also the case that the same plan is not tried for the same subgoal twice.

The following two transition rules are standard for 3APL, and define how a plan is executed. The only difference is that the plan is now inside a stack element. A basic action a at the head of a plan can be executed if the function $\mathcal{T}$ is defined for a and the belief base σ in the configuration. The execution results in a change of belief base as specified through $\mathcal{T}$, and the action is removed from the plan. Further, goals from the goal base which are reached, are removed. Note that actions, which are executed from within a module, operate on the global belief base of the agent.

Definition 11. *(action execution)*

$$\frac{\mathcal{T}(a,\sigma) = \sigma'}{\langle \sigma, \gamma, (\phi, a; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma', \gamma', (\phi, \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

where $\gamma' = \gamma \setminus \{m(\phi) \mid \sigma \models \phi\}$.

The transition below specifies the application of a plan revision rule of the form $\pi_h \mid \beta \Rightarrow \pi_b$ to a plan of the form $\pi_h; \pi$. The rule can be applied if β holds. If the rule is applied, the plan π_h is replaced by the body of the rule, yielding the plan $\pi_b; \pi$.

Definition 12. *(plan revision)*

$$\frac{\pi_h \mid \beta \rightsquigarrow \pi_b \in \mathsf{PR} \qquad \sigma \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, \gamma, (\phi, \pi_h; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \pi_b; \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

We now revisit the point made in section 2.3, that the encapsulation provided by modules at run-time is especially advantageous in the case of 3APL. The reason is that modules encapsulate not only plan generation rules, but also plan revision rules. These plan revision rules provide very flexible ways for revising a plan during execution, but a large number of rules can interact in possibly unforeseen ways. This has to do with the semantics of plans not being compositional in case of plan revision, as explained in [18]. Being able to cluster plan revision rules into modules thus reduces the chances of unforeseen interactions with other rules: the number of plan revision rules in a module will be small compared with the global set of plan revision rules of a non-modular 3APL agent.

The next two transition rules specify the cases in which a stack element can be popped from the stack. Both transition rules specify that an element can be popped if its plan has finished execution, i.e., if the plan is empty. The idea is, that an agent should always finish the execution of an adopted plan, even though, e.g., the goal for which it was selected might already be reached. This is standard for 3APL, and the reason is that the programmer might have specified some necessary "clean-up" actions. Consider as an example the case where an agent still has to pay after refueling, even though the goal of having gas is already reached.

The first transition rule for popping a stack element specifies that the element can be popped if the goal ϕ of the stack element is reached.

Definition 13. *(goal of stack element reached)*

$$\frac{\sigma \models \phi}{\langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, E\rangle}$$

The second transition rule for popping a stack element specifies that the element can be popped if there are no more applicable plan generation rules (regardless of whether the goal is reached).

Definition 14. *(no applicable plan generation rules)*

$$\frac{\neg\exists(\phi' \mid \beta \Rightarrow \pi) \in \mathsf{PG} :\ (\sigma \models_{\mathcal{L}_\mathsf{B}} \beta \text{ and } \phi \models \phi' \text{ and } \sigma \not\models \phi')}{\langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, E\rangle}$$

The next three definitions specify the semantics of the construct $m(\phi')$ for calling a module. If this construct is encountered in a plan, and it is not annotated with the symbol $\downarrow$, the module m should be called for the goal ϕ'. That is, only if ϕ' is not yet reached. If a module with the name m exists,[8] a new element with goal ϕ', an empty plan, and the rules of m, is pushed onto the stack. Further, the construct $m(\phi')$ is annotated with $\downarrow$ to indicate that a module has been called for this subgoal. This is important, since we do not want to call module m again upon returning from m.

Definition 15. *(calling a module)*

$$\frac{\langle m, \mathsf{PG}', \mathsf{PR}'\rangle \in \mathsf{Mod} \qquad \sigma \not\models \phi'}{\begin{array}{l}\langle \sigma, \gamma, (\phi, m(\phi'); \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \\ \quad \langle \sigma, \gamma, (\phi', \epsilon, \mathsf{PG}', \mathsf{PR}').(\phi, m(\phi')\downarrow; \pi, \mathsf{PG}, \mathsf{PR})\rangle\end{array}}$$

The transition rules of the next definition specify that the agent can continue the execution of the rest of its plan, if the subgoal ϕ' occurring at the head of the plan is reached. The agent should continue if it has already called the module m for the subgoal, i.e., if the construct is of the form $m(\phi')\downarrow$, or if the module has not yet been called, i.e., if the construct is of the form $m(\phi')$. The latter case is however probably less likely to occur.

Definition 16. *(subgoal reached)*

$$\frac{\sigma \models \phi'}{\langle \sigma, \gamma, (\phi, m(\phi')\downarrow; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

$$\frac{\sigma \models \phi'}{\langle \sigma, \gamma, (\phi, m(\phi'); \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

The next transition rule specifies that if a module has been called for a subgoal, i.e., if a construct of the form $m(\phi')\downarrow$ is at the head of the plan of a stack element, and the subgoal ϕ' has not been reached, the plan fails. That is, the plan is replaced by an empty plan. If there are any applicable plan generation rules left, another plan can then be selected to try to achieve the goal ϕ of the stack element.

Definition 17. *(subgoal dispatched and not reached)*

$$\frac{\sigma \not\models \phi'}{\langle \sigma, \gamma, (\phi, m(\phi')\downarrow; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle}$$

In this version of modular 3APL, a plan can fail to reach a goal only in case the programmer has specified the "wrong" actions in the plans. Since all actions (as is usually the case for 3APL's formal semantics) operate on the belief base and there is no notion of an external environment, there is no notion of the environment preventing an action from being

[8] And it should if the programmer has done a good job.

executed, thereby possibly causing a plan to fail. One could thus argue that it is not necessary to have a mechanism for selecting a different plan upon plan failure, since it is the job of the programmer to make sure that the plans do not fail and reach the goals. Any practical implementation of these ideas however would involve actions being executed in the environment, and it is thus important to incorporate a mechanism for handling plan failure..

The final transition rule below is very similar to the previous, in the sense that it specifies another reason for plan failure. In particular, it specifies that a plan fails if the action at its head cannot be executed, i.e., if the function $\mathcal{T}$ is undefined for the action and the belief base of the stack element.

DEFINITION 18. *(failure of action execution)*

$$\frac{\mathcal{T}(a,\sigma) \text{ is undefined}}{\langle \sigma, \gamma, (\phi, a; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle}$$

3.3 Example

For illustration, we present a simple example of a modular 3APL agent. The agent has to bring a rock from the location where the rocks are $loc(rock)$[9], to its base location $loc(base)$. It has three modules, i.e., *collectRock* for the general goal of collecting the rock, and *goTo* and *pickUp* for going to a location and picking up a rock, respectively. Initially, the agent believes that it is at the base location, and that it does not have a rock, i.e., $\neg have(rock)$. Further, it has the goal $collectRock(have(rock) \wedge loc(base))$, i.e., it wants to achieve the goal $have(rock) \wedge loc(base)$ using module *collectRock*.

Below, we define the initial belief base σ and goal base γ of the rock collecting agent.

$$\begin{array}{lcl} \sigma & = & \{loc(base), \neg have(rock)\} \\ \gamma & = & \{collectRock(have(rock) \wedge loc(base))\} \end{array}$$

The plan generation rules PG_{cr} of the module *collectRock* are defined as below. The set of plan revision rules is empty.

$$\begin{array}{lcl} \mathsf{PG}_{cr} & = & \{have(rock) \wedge loc(base) \mid \top \Rightarrow \\ & & goTo(loc(rock)); pickUp(have(rock)); \\ & & goTo(loc(base))\} \end{array}$$

There is one plan generation rule which can be used for the goal $have(rock) \wedge loc(base)$, i.e., the initial goal of the agent. The plan of the rule consists entirely of calls to other modules. Note that the *goTo* module is called with two different goals. The rule does not specify a condition on beliefs (other than $\top$). In particular, there is no need to specify that the agent should, e.g., believe that it is at the base location, rather than at the rock location. If the agent would already be at the rock location, the first module call of the plan, i.e., $goTo(loc(rock))$, would be skipped, since the subgoal $loc(rock)$ is already reached (see definition 16).

The plan generation rules PG_{gt} and plan revision rules PR_{gt} of the module *goTo* are defined as follows.

$$\begin{array}{lcll} \mathsf{PG}_{gt} & = & \{loc(rock) & \mid \mathbf{B}(loc(base)) \Rightarrow toRock \\ & & loc(base) & \mid \mathbf{B}(loc(rock)) \Rightarrow toBase\} \\ \mathsf{PR}_{gt} & = & \{toRock & \mid \mathbf{B}(loc(rock)) \rightsquigarrow skip \\ & & toRock & \mid \neg\mathbf{B}(loc(rock)) \rightsquigarrow east; toRock \\ & & toBase & \mid \mathbf{B}(loc(base)) \rightsquigarrow skip \\ & & toBase & \mid \neg\mathbf{B}(loc(base)) \rightsquigarrow west; toBase\} \end{array}$$

[9]Note that all formulas are propositional, although we use brackets for presentation purposes.

This module has two plan generation rules: one for selecting a plan to get from the base location to the rock location, and one for the other way around. The plans *toRock* and *toBase* in the bodies of the plan generation rules are non-executable basic actions, which are used as procedure variables.[10] Assuming that the rock is located east from the base location, we specify plan revision rules for moving east until the rock location is reached, and moving west until the base location is reached. The action *skip* is a special action which does nothing if executed.

In this simple example, we specify separate plan revision rules for moving east and west respectively. The plans for moving to the rock location and to the base location thus use different plan revision rules. We could therefore have created two separate modules, i.e., one for going to the rock, and one for going to the base. In a more realistic setting however, one could imagine to have *one* set of plan revision rules for moving to any given location. In that case, it would be advantageous to specify these rules only in one module.

The plan generation rules PG_{pu} of the module *pickUp* are defined as below, and the set of plan revision rules is empty.

$$\begin{array}{lcl} \mathsf{PG}_{pu} & = & \{have(rock) \mid \mathbf{B}(loc(rock)) \Rightarrow pickUp1 \\ & & have(rock) \mid \mathbf{B}(loc(rock)) \Rightarrow pickUp2\} \end{array}$$

The plan generation rules of this module can be applied if the agent believes it is at the rock location. Note that the call $pickUp(have(rock))$ to this module from the *collectRock* module, can only be executed if the subgoal of the previous module call, i.e., $goTo(loc(rock))$, has been achieved. If $pickUp(have(rock))$ is executed, we thus know for sure that the agent believes to be at the rock location. The module *pickUp* illustrates that the agent may have multiple plans, i.e., $pickUp1$ and $pickUp2$ in this case, for achieving the same goal. If, e.g., $pickUp1$ fails, the agent can try $pickUp2$.

4. FUTURE RESEARCH

As directions of future research, we mention the following. In the current proposal there is no notion of importing a module into another module. Some notion of importation could be added in a straightforward way, but it will have to be investigated what exactly the semantics of importation should be. Should the rules of the imported module just be "added" to the other module, or should there be some kind of prioritization among rules of the module itself and the imported module?

Further, goal-oriented modularity as presented here provides a high degree of information hiding. That is, when calling a module, only the name of the module has to be known. In principle, any part of a module can be adapted without having to adapt the call to the module. We thus have no notion of an interface of a module, i.e., those parts of the module which are known outside the module. Nevertheless, it might be worthwile to investigate whether the framework can be extended with some notion of interface, such as the goals for which a module can be called, etc. This might be a useful tool to help a programmer.

Also, we envisage that a mechanism similar to the mechanism of dispatching a goal to a module, could be used in a multi-agent team work setting (see, e.g., [22]) to delegate a goal to an agent. Using a uniform mechanism both for

[10]Non-executable basic actions are often termed *abstract plans* in the 3APL literature.

calling modules and delegating goals to agents could potentially yield more transparent systems. In case of delegation, a plan would have to contain a request message to an agent, rather than a call to a module. A way of "returning" from the request, just as one can return from a module, would have to be defined. An agent would for example have to report back to the requesting agent, either with a message expressing that the goal has been achieved, or that he has failed and stopped trying. Moreover, it can be interesting to investigate a construct for calling modules in parallel. This can also be interesting in the case of multi-agent teamwork as also discussed in [22], since an agent could then dispatch several goals to different agents in parallel.

Finally, we remark that it could be useful to be able to store information during execution within a module, which would not need to be kept after returning from the module. This could be realized by adding actions to the specification of a module, which would then update the module's internal information store, rather than the global belief base. This could be considered as a compromise between the way beliefs are handled in capabilities, and the way we handle beliefs in the presented framework.

Concluding, we have presented the idea of goal-oriented modularity, which takes the goals of an agent as the basis for modularization. Since we view goals as being an essential ingredient of cognitive agents, we argue that this approach to modularity is suited for cognitive agent programming languages. Further, we have shown how goal-oriented modularity might be incorporated in a cognitive agent programming language, by presenting a formalization of goal-oriented modularity in 3APL.

5. REFERENCES

[1] F. Bellifemine, A. Poggi, G. Rimassa, and P. Turci. An object oriented framework to realize agent systems. In *Proc. of WOA 2000 Workshop*, pages 52–57. 2000.

[2] J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372, 1990.

[3] R. H. Bordini and A. F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. *Electronic Notes in Theoretical Computer Science*, 70(5), 2002.

[4] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the capability concept for flexible BDI agent modularization. In *Proc. of ProMAS'05*, 2005.

[5] P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI agents in functional clusters. In *ATAL '99: 6th Int. Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages*, pages 277–289, 2000. Springer-Verlag.

[6] M. Dastani, M. B. van Riemsdijk, F. Dignum, and J.-J. Ch. Meyer. A programming language for cognitive agents: goal directed 3APL. In *Programming multiagent systems, first int. workshop (ProMAS'03)*, *LNAI*, pages 111–130. Springer, Berlin, 2004.

[7] M. Dastani, M. B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. Ch. Meyer. Enacting and deacting roles in agent programming. *Agent-Oriented Software Engineering V*, *LNCS*, pages 189–204. Springer-Verlag, 2005.

[8] M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.

[9] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice-Hall International, London, 1991.

[10] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[11] B. Meyer. *Object-oriented software construction*. Series in Computer Science. Prentice-Hall International, London, 1988.

[12] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[13] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: a BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.

[14] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. van der Velde and J. Perram, editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.

[15] M. B. van Riemsdijk, M. Dastani, F. Dignum, and J.-J. Ch. Meyer. Dynamics of declarative goals in agent programming. *Proc. of the second int. workshop on Declarative agent languages and technologies (DALT'04)*, *LNCS*, pages 1–18. Springer-Verlag, 2005.

[16] M. B. van Riemsdijk, M. Dastani, and J.-J. Ch. Meyer. Semantics of declarative goals in agent programming. In *Proc. of AAMAS'05*, 2005.

[17] M. B. van Riemsdijk, M. Dastani, and J.-J. Ch. Meyer. Subgoal semantics in agent programming. *Progress in Artifical Intelligence: 12th Portuguese Conference on Artificial Intelligence (EPIA'05)*, *LNCS*, pages 548–559. Springer-Verlag, 2005.

[18] M. B. van Riemsdijk, J.-J. Ch. Meyer, and F. S. de Boer. Semantics of plan revision in intelligent agents. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Theoretical Computer Science*, pages 240–257. 2006. Special issue of Algebraic Methodology and Software Technology (AMAST'04).

[19] M. B. van Riemsdijk, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in Dribble: from beliefs to goals using plans. In *Proc. of AAMAS'03*, 2003.

[20] M. Winikoff. JACK™ intelligent agents: an industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.

[21] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proc. of the eighth int. conf. on principles of knowledge respresentation and reasoning (KR2002)*, 2002.

[22] K. Yoshimura, R. Rönnquist, and L. Sonenberg. An approach to specifying coordinated agent behaviour. In *PRIMA'00*, *LNAI*, pages 115–127. Springer, 2000.