Modules as Policy-Based Intentions: Modular Agent Programming in GOAL

Koen Hindriks

EEMCS, Delft University of Technology, Delft, The Netherlands
k.v.hindriks@tudelft.nl

Abstract. Modular programming has the usual benefits associated with structured programming, information hiding and reusability, but also has additional benefits to offer when applied in agent programming. We argue that modules can be viewed as structures similar to that of policy-based intentions [2]. Modules perceived in this way are components within an agent that are triggered in a particular situation and combine the knowledge and skills to adequately pursue the goals of the agent in that situation. The context that triggers the activation of a module defines the interface of the module, which can be specified declaratively, in contrast to the usual functional interpretations of such interfaces. A feature that differentiates our notion of a module from plans is that modules provide an agent with a means to *focus* its attention on the relevant resources it needs to handle a situation. As a result, modules can be used to control or reduce the underspecification and inherent non-determinism that is typical of agent programs. In the paper, the proposed module concept is incorporated into the agent language GOAL and illustrated by means of a simple example.

1 Introduction

It has been argued by several authors that besides being able to decompose a complex system into multiple agents it is also important to be able to decompose single agents into structured units in agent programming languages. For various reasons, it is not always appropriate to provide this additional structure by decomposing a single agent into a group of yet smaller agents. An agent-based decomposition introduces additional communication overhead and requires duplication of knowledge and goals in those agents. This has motivated the introduction of *modularization* as a decomposition technique into various agent programming frameworks (cf. [3, 4, 16]).

Apart from the traditional motivations for modularization, we argue that there are also reasons more specifically related to rational agents for incorporating modules into agents. As in other programming paradigms, modularization provides the usual benefits associated with *structured programming*, *information hiding*, and *reusuability*.

In agent programming, modules support the *encapsulation* of domain knowledge, basic actions and plans that are logically related and relevant for handling

particular situations. From a software engineering point of view, modules allow a programmer to focus on those skills that are required to handle a situation. As components of an agent program, modules can be viewed as specialized, dedicated units of control to realize particular goals of the agent. Modules in agent programming are also called *capabilities* sometimes in the literature (cf. [3, 4]).

As will be discussed below in more detail, modular agent programming also provides additional benefits which are not traditionally recognized or simply do not apply to other programming paradigms. One of the most important of these is the fact that *modules provide additional structure to control the inherent nondeterminism of agent programs.* Agent programs typically do not specify for each situation that the agent may encounter a unique course of action that the agent should execute. In particular, often actions in parallel plans are interleaved nondeterministically, and various goal adoption rules and plan selection rules may be selected for execution at any time. As a result, agent programs in general *underspecify* the course of action that an agent takes.

This underspecification present in agent programs may result in suboptimal or even irrational behavior of an agent. Since an agent is supposed to "do the right thing", various proposals have been made to provide an agent with additional means to control the choices left open by the agent program. One particular strand of research has focused on defining control structures to achieve this objective. In the context of agent programming these are also called *deliberation cycles* (cf. [5, 14]). Another interesting proposal has been to use decision-theoretic techniques (e.g. [1]). The proposal discussed in this paper is to use modules to provide *focus* in the selection of actions of an agent. It is argued that the concept of a module provides for a particularly flexible programming technique to reduce the underspecification typically present in agent programs.

A further advantage of introducing modules in agent programming is that the interface of a module can be provided with a natural and moreover completely declarative definition. This is a distinguishing feature of the module concept presented in this paper. Typically, the *interface* of a module that implements the information hiding is based on an explicit *importing* and *exporting* mechanism which is *not* declarative. As a result, such module interfaces do not provide a declarative specification of what they can be used for but instead only specify an accessibility mechanism that determines what is "visible" to the environment of the module. A declarative concept of module interfaces as proposed here, however, allows a programmer to read of the module's intended use from its interface without any additional inspection of the implementation details inside a module. The idea is that a declarative interface specifies in what circumstances a module can usefully be activated.

The declarative nature of module interfaces differentiates our proposal from those that are inspired by Prolog and Object Oriented concepts of modules such as [3, 4] and is closer in spirit to the logic-based approach in [12]. In line with our conception of a module being specialized in handling specific situations it is natural to define a module interface as a condition that identifies that situation. The declarative interface of an agent module specifies *which situations* a module

can handle well because it is designed to do just that. The internal structure of a module specifies *how* the situation specified by the interface is to be handled: it encapsulates the basic actions, knowledge, and plans that the agent needs to handle the situation, given its current goals.

This view of modules in agent programming provides for a natural and intuitive separation of concerns. On the one hand, the encapsulation of basic actions, domain knowledge and plans in a module facilitates the programmer in combining all relevant knowledge and skills that are needed to handle a particular situation. On the other hand, the declarative specification of a module interface entails that it can be defined more or less independently from other parts of the program: A module only has to provide a kind of plan to handle the situation as specified by the interface.

This concept of a module that focuses the attention of an agent in order to handle the situation at hand is incorporated in this paper in the agent programming language GOAL. Due to the additional structure that modules provide, the incorporation of modules into GOAL can also be viewed as an extension that makes available a structure similar to a *policy* or *plan* in GOAL. The paper is organized as follows. First, a brief overview of the GOAL programming language is presented. In section 3 GOAL is extended with modules. The semantics of modules is informally motivated and formally specified by providing an operational semantics. Section 4 compares with related work and concludes the paper.

2 The GOAL Language

GOAL, for Goal-Oriented Agent Language, is an agent programming language that incorporates declarative notions of beliefs and goals, and a mechanism for action selection based on these notions. That is, GOAL agents derive their choice of action from their beliefs and goals. For a detailed overview and discussion of the language see [7, 10]. An example of an (incomplete) GOAL agent program that will be used throughout the paper for illustrative purposes is provided in Figure 1. This agent provides a specification for a delivery agent that delivers parcels to various clients. A GOAL agent program consists of four sections: (1) a set of initial beliefs, collectively called the (initial) *belief base* of the agent, (2) a set of initial goals, called the (initial) *goal base*, (3) a *program section* which consists of a set of conditional actions, and (4) an *action specification* that consists of a specification of the pre- and post-conditions of *basic actions* of the agent. To avoid confusion of the program section with the agent program itself, from now on, the agent program will simply be called *agent*. The term *agent* will be used both to refer to the program text itself as well as to the execution of such a program. It should be clear from the context which of the two senses is intended.

The program and action specification components of a GOAL agent are static and do not change at runtime. The agent's belief and goal bases are dynamic and may vary over time. They change because of actions that are performed

```
:main:deliveryAgent
{
   :beliefs{ home(a).
     loc(p1,a). loc(p2,a). loc(p3,a). loc(p4,a). loc(truck,a).
     loc(c1,b). loc(c2,c). order(c1,[p1,p2]). order(c2,[p3,p4]).
     ordered(C,P) :- order(C,Y), member(P,Y).
     loaded_order(C) :- order(C,O), loaded(O).
     delivered_order(C) :- order(C,O), loc(C,X), loc(O,X), loc(truck,a).
     packed :- setOf(P,in(P,truck),L), size(L,2).
     empty :- setOf(P,in(P,truck),[]).
   }
   :goals{ delivered_order(c1). delivered_order(c2). ... }
   :program{
     if goal(delivered_order(C)), bel(ordered(C,P)), ~bel(in(P,truck))
       then load(P).
     if goal(delivered_order(C)),
        bel(loc(truck,X), loaded_order(C), loc(C,Y)) then goto(Y).
     if goal(delivered_order(C)), bel(loc(truck,X), loc(C,X),
        in(P,truck), ordered(C,P)) then unload(P).
     if bel(loc(C, X), empty, home(Y)) then goto(Y).
     if bel(ordered(C,P), empty),~bel(in(P,a)) then adopt(in(P,a)).
     ...
   }
   :action-spec{
     load(P){
       :pre{~packed, loc(truck,X), loc(P,X)}
       :post{in(P,truck), ~loc(P,X)} }
     goto(Y){
       :pre{loc(truck,X), X≠Y}
       :post{loc(truck,Y), ~loc(truck,X)} }
     ...
   }
}
```

Fig. 1. Example of a GOAL agent program

by the agent, which, apart from changing the agent's environment, also update and modify the beliefs and, indirectly, the agent's goals. Belief bases are typically denoted by Σ and goal bases by Γ. Together, the belief and goal base pair $\langle \Sigma, \Gamma \rangle$ are called the *mental state* of the agent, typically denoted by s. The language GOAL does not fix the representation of beliefs nor goals, but here we assume they are sentences from a first-order language, which in practice are suitably restricted to allow for an efficient implementation. Mental states are required to satisfy the following rationality constraints ($\models$ denotes first-order entailment):

(i)	Belief bases are consistent:	$\Sigma \not\models \mathbf{false}$,
(ii)	Individual goals are consistent:	$\forall \gamma \in \Gamma$: $\not\models \neg\gamma$,
(iii)	Goals are not believed to be achieved:	$\forall \gamma \in \Gamma$: $\Sigma \not\models \gamma$.

The beliefs of the agent in Figure 1 consist of facts about the current situation, in this case about parcel locations and clients and their orders, and a number of rules that represent the logical relations between these facts. For example, the rule for `delivered_order(C)` states that an order is delivered if all ordered parcels have been delivered at the client's site location and the truck is (back) at its home base `a`. (Due to space limitations, definitions of the `loc` and `loaded` predicates for lists and the `unload` action specification are not included, but the intended meaning should be clear. `setOf` is a standard Prolog predicate that returns a list of items satisfying the condition of its second argument.) Note that the example agent does not believe that it delivered an order and thus initially satisfies the constraint that goals are not believed by the agent.

The conditional actions in the program section of a GOAL agent define a mapping from states to actions, together specifying a non-deterministic policy or course of action. The condition of a conditional action is called a *mental state condition*. It determines the states in which the conditional action may be executed. Mental state conditions are boolean combinations of basic formulas **bel**(ϕ) or **goal**(ϕ) with ϕ a first-order formula. A prolog-like notation is used in examples, as in Figure 1: Literals in a conjunction are separated by means of a comma, and negation is written as $\sim$. (In the main text, however, we also use $\neg$ and $\wedge$ to denote negation and conjunction.) These conditions allow an agent to inspect its beliefs and goals. For example, in the program section of the agent in Figure 1, the first conjunct of the first condition, **goal**`(delivered_order(C))`, inspects the goal base and verifies whether the agent has a goal to deliver for some client `C`, and the second conjunct **bel**`(ordered(C,P))` inspects the belief base and verifies whether client `C` ordered a parcel `P`. Free variables in mental state conditions are instantiated when the condition is evaluated at runtime.

Since mental state conditions need not be exclusive, multiple conditional actions may be simultaneously enabled. GOAL agents thus may underspecify the behavior of the agent resulting in a non-deterministic choice of action. In Figure 1, initially the first conditional action is enabled for each of the four parcels listed and the agent may load any of these parcels into the truck by executing a corresponding instantiation of the action `load(P)`. In such a case, the agent may non-deterministically choose any one of these actions for execution.

The fact that agents may be underspecified may provide benefits at design time, but it may also pose problems at runtime. In the example, the agent is supposed to deliver orders to various clients. In order to do so, the agent first needs to load the truck with the ordered items. However, since the agent has multiple orders to deal with and no priority on handling these orders has been specified, the agent may end up loading parcels into the truck that do not belong together. Since the load the truck can carry is also very limited, as a result, the agent may end up delivering no orders at all and end up in a deadlock situation. (Of course, a slightly smarter agent would start unloading parcels again, but this would not guarantee resolution of the problem. Other ways to resolve the problem in a principled way seem to require significant modification of the agent.) Note that in case the agent would have had only a single delivery goal to deal

with, there would have been no problem, indicating that the example delivery agent is not an incorrect implementation per se. With the appropriate focus, the agent would have been able to deliver successfully. To provide agents with such focus is one of our motivations for introducing modules. Modules are introduced to provide a means to control the non-determinism inherent in agents and to provide agents with a focus of attention on some of their goals among the many others that they may have.

3 Modules As Policy-Based Intentions

In this section the use of modules conceived as policy-based intentions is illustrated using the example introduced in the previous section and the informal discussion of modules is complemented with a precise definition of the operational semantics of modules by means of a transition system (cf. [13]).

The concept of a module that is introduced here is inspired by the the concept of a *policy-based intention* in [2] and motivated by the fact that modules so viewed can be identified with plans or policies that guide the agent's action. Policy-based intentions are general policies and concern potentially recurring circumstances in the agent's life. Such policies shape an agent's plans in ways that may help achieve a range of different and potentially conflicting goals. They do so by providing a partial solution to the problems posed by the limited resources for deliberation by making a previously successfully tested strategy readily available to the agent. Policy-based intentions may be particular to an agent, coding the specific ways in which that agent typically handles a recurring circumstance.

Our notion of module incorporates the main ideas of such policy-based intentions. In particular, it incorporates the notion of a *circumstance-triggered* intention and a notion of *commitment* to executing the intention. A module viewed as a policy-based intention specifies these circumstances as a condition for activating the module. A module, additionally, can be used to structure and combine the relevant knowledge and skills needed to handle such circumstances in ways that help achieve the agent's goals. Modules do not only describe the capabilities that an agent has, but specify a policy or plan that an agent applies in particular situations to handle that situation.

The GOAL language allows for an elegant definition of modules that are circumstance-triggered, general policies for acting. Syntactically, GOAL modules are just GOAL agents with an additional *context section*. A module also has a name, which serves as a bookkeeping device and facilitates easy reference. An example GOAL agent with two modules named `deliverOrder` and `stockMngt` is provided in Figure 2. The example is a modified version of the agent in Figure 1 in which most of the program text has been placed inside the modules except for the facts in the belief base and the initial goals in the goal base (the `...` refer to missing parts, which can be filled in partially by copying text from Figure 1). The context and program section of a module must be non-empty, but the belief and goal sections may be empty. Empty module sections can simply be left out, e.g. in our example the goals section of the first module might have been left out.

For ease of reference, below we write *m.section* to refer to each of the different sections of a module named m. For example, `deliverOrder.context` refers to the context section of module `deliverOrder`.

Module Activation: Intuitively, a module is specialized in handling the situations that are specified by the context section. The context section of a module, which may be any mental state condition, determines its activation condition: A module may be *activated* when its context section is true. For example, the context section in Figure 2 of the module `deliverOrder` specifies that the module is specialized in achieving an instance of the goal `deliver_order(C)`. Context sections may also include conditions on the beliefs of an agent, to indicate in which circumstances a module may be used to achieve a goal, for example, a precondition for activating the `deliverOrder` module is that the ordered items are in stock. Note that context sections thus allow to define preconditions for executing composed activities, i.e. plans or polices, that are specified in the program section of a module. A module with a context section that consists of belief conditions only such as the `stockMngt` module is called a *reactive module*.

In order to present the formal operational semantics below a precise definition of the semantics of mental state conditions is required. A mental state condition is evaluated in a mental state $\langle \Sigma, \Gamma \rangle$. We overload the first-order entailment $\models$ and also use it to denote the truth conditions of mental state conditions.

Definition 1. (Mental State Condition Semantics)
The truth conditions for (closed) mental state formula, relative to a mental state $s = \langle \Sigma, \Gamma \rangle$, are defined by the following four clauses:

$$
\begin{array}{lll}
s \models \mathbf{bel}(\phi) & \textit{iff} & \Sigma \models \phi, \\
s \models \mathbf{goal}(\phi) & \textit{iff} & \textit{there is a } \gamma \in \Gamma \textit{ s.t. } \gamma \models \phi \textit{ and } \Sigma \not\models \phi, \\
s \models \neg\varphi & \textit{iff} & s \not\models \varphi, \\
s \models \varphi_1 \wedge \varphi_2 & \textit{iff} & s \models \varphi_1 \textit{ and } s \models \varphi_2.
\end{array}
$$

Like any mental state condition, a context φ may have free variables, denoted by $free(\varphi)$. Any free variables that occur in other sections of a module, with the exception of the program section, should also occur in the context section. (In the example agents, rules are assumed to be implicitly universally quantified.) Variables are instantiated at runtime when a module is activated. A module is instantiated when all its free variables have been instantiated. Formally, an instantiation of a module with free variables var in its context section is a substitution ρ such that $dom(\rho) =$ var and the range of ρ is a set of constants or closed terms. The application of a substitution ρ to a formula φ is written as usual as $\varphi\rho$. The composition of two substitutions ρ_1 and ρ_2 is written as $\rho_1 \circ \rho_2$.

The activation of a module requires that various items are recorded to facilitate a proper definition of the operational semantics, e.g. that module m has been activated, with what values ρ the free variables in the context of module m have been instantiated, and which goals are actively pursued. To this end, the notion of a *configuration* which extends a mental state with these items is introduced. Note that the static parts of a module, i.e. the domain knowledge,

```
:main:deliveryAgent
{
   :beliefs{ home(a).
     loc(p1,a). loc(p2,a). loc(p3,a). loc(p4,a). loc(truck,a).
     loc(c1,b). loc(c2,c). order(c1,[p1,p2]). order(c2,[p3,p4]).
   }
   :goals{ delivered_order(c1). delivered_order(c2). ... }
   :program{ ... }
   :action-spec{ ... }
   :module:deliverOrder{
    :context{ bel(order(C,O), in(O,a)), goal(delivered_order(C)) }
    :beliefs{
      ordered(C,P) :- order(C,Y), member(P,Y).
      ...
    }
    :goals{ }
    :program{
      if bel(ordered(C, P)), ~bel(in(P, truck)) then load(P).
      if bel(loc(truck, X), loaded_order(C), loc(C, Y)) then goto(Y).
      if bel(loc(truck,X), loc(C,X), in(P,truck), ordered(C,P))
        then unload(P).
      if bel(loc(C, X), empty, home(Y)) then goto(Y).
    }
    :action-spec{ ... }
   }
   :module:stockMngt{
    :context{ bel(ordered(C,P), empty), ~bel(in(P,a)) }
    :goals{ in(P,a) }
    :program{ ... }
    :action-spec{ ... }
   }
...
}
```

Fig. 2. Example of GOAL Agent with Modules

conditional actions in the program section, and the action specification need not be explicitly represented in a configuration. They can be retrieved when needed from the module and instantiated appropriately by applying the substitution which is recorded in a configuration. We write $\langle m, \rho, \Gamma_m, \langle \Sigma, \Gamma \rangle \rangle$ to represent a configuration in which a module m has been activated in a mental state $\langle \Sigma, \Gamma \rangle$.

A module may be activated while another module has been activated. A configuration thus may consist of a stack of modules and in that case a more recently activated module is executed within the context of a previously activated module. For example, it may be that the former is activated because of additional subgoals or domain knowledge introduced by the latter module. This need not be the case, however, since e.g. the `stockMngt` module may be activated because of changes of the beliefs in the agent's belief base. To allow for activation of other modules in the case that other modules have been activated and not yet termi-

nated, we also write $\langle m_1, \rho_{m_1}, \Gamma_{m_1}, \ldots, \langle m_n, \rho_{m_n}, \Gamma_{m_n}, \langle \Sigma, \Gamma \rangle \rangle \ldots \rangle$ to indicate that module m_1 has been activated after modules m_n to m_2 (in that order) have been activated. We say that a *ρ-instantiation of module m has been activated* if the module name together with the substitution ρ occurs in the configuration $\langle m_1, \rho_{m_1}, \Gamma_{m_1}, \ldots, \langle m_n, \rho_{m_n}, \Gamma_{m_n}, \langle \Sigma, \Gamma \rangle \rangle \ldots \rangle$. Moreover, only the most recently activated module is called *active*.

The belief section of a module can be used to specify relevant domain knowledge for handling a situation. There is no restriction on the formulas allowed in the belief section of a module; it may contain both facts as well as rules, just like the belief base of the agent. When more than one module has been activated, we need to ensure that the domain knowledge of each activated module can be accessed by the agent. Each module is activated within a particular context and the information about this context, represented by the domain knowledge of previously activated modules, should still be available to the agent. To this end, we define the notion of *accessible beliefs in configuration v* as the set $\Sigma_{accessible}$ of the beliefs Σ combined with the domain knowledge $(m_i.beliefs)\rho_i$ of each activated module m_i, where any free variables have been instantiated by applying the substitution ρ_i. The accessible domain knowledge of all activated, properly instantiated modules is also denoted by Σ_{domain}. Note that $\Sigma_{domain} \subseteq \Sigma_{accessible}$.

In the goal section of a module additional subgoals may be introduced for structuring the problem that needs to be dealt with, as is done e.g. in the `stockMngt` module. These goals are goals local to the module and are only pursued while the module is activated. Subgoals introduced by a module may trigger other modules again to achieve these subgoals.

Modules provide more focus by restricting the set of goals that the agent actively pursues. The notion of *active goals in configuration v* is defined as the set of goals associated with the most recently activated module, i.e. $\Gamma_{active} = \Gamma_{m_1}$ where m_1 denotes that module. (Note that in this case no substitutions need to be applied since the goals in Γ_{m_1} have already been instantiated.) All other goals are called *passive* and the set of these goals can be defined as $\Gamma_{passive} = \Gamma \cup \Gamma_{m_2} \cup \ldots \cup \Gamma_{m_n}$ where Γ denotes the top-level goals of the agent and the Γ_{m_i} denote the goals introduced and processed by previously activated modules m_i. Finally, $\Gamma_{all} = \Gamma_{active} \cup \Gamma_{passive}$, i.e. Γ_{all} is the set of all goals currently adopted by the agent.

When a module is activated, a filter is applied to the then active set of goals to select only those that triggered the activation. As a result, the agent will focus on those goals that entail the context of the module given the currently accessible beliefs. Informally, only those goals that the agent has adopted and make the context condition of the module true are considered after activation of that module in combination with those introduced by the module's goal section. The set of active goals is computed from the context section and the goal section. A context condition φ can be converted into disjunctive normal form, taking formulas of the form $\mathbf{bel}(\phi)$ and $\mathbf{goal}(\phi)$ as atoms. An occurrence of an atom of the form $\mathbf{bel}(\phi)$ or $\mathbf{goal}(\phi)$ is called a *positive literal* if it occurs unnegated in the normal form, otherwise it is called a *negative literal*. Assuming that a context

condition φ is in disjunctive normal form, the function $filter(\varphi)$ extracts all positive literals of the form $\mathbf{goal}(\phi)$ from φ and removes the goal operator **goal**. For example, if $\varphi = [\mathbf{bel}(p) \wedge \mathbf{goal}(q)] \vee [\mathbf{goal}(r) \wedge \neg\mathbf{goal}(p)]$, then $filter(\varphi) = \{q, r\}$. The focus of attention then is defined as those filtered atoms that are also currently active goals of the agent by the function $focus(\varphi, s) = \{\phi \in filter(\varphi) \mid s \models \mathbf{goal}(\phi)\}$, where s is a state defined by the accessible beliefs and active goals.

Definition 2. (Module Activation Rule)
Let $v = \langle m_1, \rho_{m_1}, \Gamma_{m_1}, \ldots \langle \Sigma, \Gamma \rangle \ldots \rangle$ be a configuration (possibly without module instantiations, i.e. $v = \langle \Sigma, \Gamma \rangle$), m be a module, and ρ be a substitution such that $dom(\rho) = free(m.context)$. Then the activation *of module m is defined by:*

$$\frac{\langle \Sigma_{accessible}, \Gamma_{active} \rangle \models (m.context)\rho \quad \textit{no } \rho\textit{-instantiation of module } m \textit{ has been activated yet}}{v \longrightarrow \langle m, \rho, focus((m.context)\rho, \langle \Sigma_{accessible}, \Gamma_{active} \rangle) \cup (m.goals)\rho, v \rangle}$$

The second condition in the rule avoids that the same instantiation of a module is activated twice. The activation of a module does not change the beliefs or goals in the mental state of the initial configuration v. Implicitly, however, the set of accessible beliefs $\Sigma_{accessible}$ is extended with the instantiated domain knowledge in the belief section (if any) of the module. Module activation also changes the set of active goals Γ_{active} to the set of goals that result from filtering the context of the module and computing the corresponding focus of attention combined with goals that result from instantiating the goal section of the module. As a consequence, other modules are only activated when they are relevant for achieving an active goal (with the exception possibly of reactive modules).

In our simple example agent, the module `deliverOrder` serves to provide a focus of attention on one of the delivery goals of the agent and to temporarily disregard any other goals that the agent may have. Upon activation of that module, the context is instantiated with either client `c1` or client `c2`. Assuming that `c1` is used, the set of active goals is restricted to the goal of delivering an order for `c1` and the goal of delivering for `c2` becomes passive. This provides the agent with the relevant focus to complete a delivery for a single client. The actions in the program section of the module thus will only be directed at achieving that goal and potential conflicts due to other goals are avoided.

Action Execution: Once a module is activated execution is restricted to actions from the module's program section. This is a second way to structure and focus the behavior of an agent. For example, by excluding the first conditional action for adopting a goal to replenish stock from the module in Figure 2, any potential interference of actions to achieve this goal with the actions for delivering the ordered parcels is prevented. All actions that are relevant for delivering an order are combined in the module which in this way facilitates the specification of a general policy for achieving this goal. In the `deliverOrder` module the program section specifies that the context of the module is handled by loading the truck

with the items the client ordered, going to the client site, unloading the ordered items, and returning to home base.

In order to define the semantics of action execution, a transition function $\mathcal{T}$ is assumed to be given that captures the semantics of basic actions `a` and is consistent with all the action specification sections (including those within modules). Action specifications, moreover, are required to be consistent with the domain knowledge stored in a module. That is, since it is assumed that such knowledge does not change during the lifetime of an agent, an action is not allowed to update this knowledge to avoid inconsistencies when a module is activated. While in principle such beliefs can be added to the belief base, encapsulating such knowledge in a module facilitates information hiding and efficient execution. In the example agent of Figure 2 the rule for the predicate `ordered` is used to represent fixed domain knowledge about the relation between individual ordered items and the order of a client. Formally, this means that the transition function should be defined in such a way that actions never update knowledge stored in a module.

Constraint 1. (Domain Knowledge Not Updated)
If a belief section in a module consists of a set of formulas D, then for any belief base Σ and action `a` *(including* **skip***, i.e. the action without any effects) it must be the case that $\mathcal{T}(\mathtt{a}, \Sigma \cup D) = \Sigma' \cup D$, such that $\Sigma' \cup D$ is consistent.*

Since the **skip** action does not change the configuration of an agent, it follows that $\mathcal{T}(\textbf{skip}, \Sigma \cup D) = \Sigma \cup D$ must be consistent, which implies in particular that the domain knowledge present in the various modules of an agent also should be consistent with the initial beliefs of an agent.

An important aspect of modules concerns the encapsulation of the effects of belief updates and goal updates. It is argued here that the beliefs in the mental state of an agent (in contrast with the domain knowledge stored in modules), and any updates on these beliefs, should *not* be encapsulated in a module. This would make these beliefs "invisible" to other modules. The effects on the agent's environment caused by executing a module, should be available for later reference and therefore incorporated into the agent's belief base. For example, the updated locations of parcels need to be stored in the belief base of the example agent.

It has already been argued that goals of an agent should be local to a module in order to provide an agent with a focus on the goals relevant in a particular situation. But even though the agent's focus is on achieving the active goals by selecting appropriate actions, whenever either an active or passive goal has been achieved such a goal is updated and removed from the set of all adopted goals. It is considered irrational for an agent to invest any more time and resources in a goal that has already been achieved.

The previous discussion is formally captured in the action execution rule for modules. The rule restricts the choice of action to those that are available within the program section of the most recently activated module. Modules thus may create focus and prevent unexpected or undesirable interference effects of other actions.

Definition 3. (Action Execution Rule: Modules)
Let $v = \langle m_1, \rho_{m_1}, \Gamma_{m_1}, \ldots \langle \Sigma, \Gamma \rangle \ldots \rangle$ be a configuration, c be a conditional action of the form **if** φ **then** $\mathtt{a}(\boldsymbol{t})$ *in the program section of module m_1, ρ a substitution with $dom(\rho) = free(\varphi\rho_{m_1})$, and $\sigma = \rho_{m_1} \circ \rho$. Then the execution of the conditional action c is defined by:*

$$\frac{\langle \Sigma_{accessible}, \Gamma_{active} \rangle \models \varphi\sigma \text{ and } \Gamma_{active} \neq \emptyset}{v \longrightarrow \langle m_1, \rho_{m_1}, \Gamma'_{m_1}, \ldots \langle \Sigma', \Gamma' \rangle \ldots \rangle}$$

where:

- $\Sigma' = \mathcal{T}(\mathtt{a}(\boldsymbol{t})\sigma, \Sigma_{accessible}) \setminus \Sigma_{domain}$,
- $\Gamma'_{(i)} = \Gamma_{(i)} \setminus \{\psi \in \Gamma_{(i)} \mid \mathcal{T}(\mathtt{a}(\boldsymbol{t})\sigma, \Sigma_{accessible}) \models \psi\}$, *where $\Gamma_{(i)}$ denotes any of the sets Γ_i or the top-level goal base Γ.*

Note that only the top-level beliefs in the belief base Σ of the agent are updated when an action is performed. The accessible domain knowledge stored in modules is not updated (and excluded from the result of applying the transition function to the set of all accessible beliefs). The definition of the updated beliefs Σ' based on the transition function $\mathcal{T}$ is correct provided that the constraint on updating knowledge stored in modules holds (cf. constraint 1).

Execution at the top-level (i.e. when no modules have been activated) is defined exactly the same as that for GOAL without modules (cf. [7]). In fact, it is a special case of the action execution rule for modules below since a GOAL agent can be viewed as a module without a context section.

Goal Update Actions: The action execution rule is not applicable to the goal update actions but only to basic actions $\mathtt{a}(\boldsymbol{t})$. Different rules are needed for the goal update actions **drop** and **adopt**. Only the rule for **adopt**(ϕ) is provided here. The rule for **drop** can be derived from the rule provided in [7] and the action execution rule above. A **drop**(ϕ) action does not have any effect on the beliefs of an agent and simply removes all goals from the total set of goals which imply that ϕ is a (sub)goal of the agent, i.e. all active as well as passive goals that imply that ϕ is a (sub)goal of the agent are removed from the goal sets Γ_m in a configuration v, and from the top-level goal base Γ.

The rule for executing an **adopt**(ϕ) action requires the agent to check whether it is reasonable to add the goal ϕ, properly instantiated, to the set of adopted goals within the current context. A weak condition is used to verify whether adopting ϕ is reasonable: ϕ may be adopted if it is consistent and is not currently implied by any of the accessible beliefs of the agent. It is not required that ϕ is consistent with the domain knowledge of the agent, in order to avoid unnecessary complications. It is left to the programmer to verify that such consistency will always be maintained. The goal ϕ that is adopted is added to the set of active goals associated with the most recently activated module. The motivation for this is that newly adopted goals are only valid within the context of that module.

Definition 4. (Execution Rule for **adopt**: Modules)
Let $v = \langle m, \rho_m, \Gamma_m, \ldots \langle \Sigma, \Gamma \rangle \ldots \rangle$ be a configuration, c be a conditional action of the form **if** φ **then adopt**(ψ)*, ρ be a substitution with $dom(\rho) = free(\varphi\rho_m)$, and $\sigma = \rho_m \circ \rho$. Then the adoption of a goal ϕ is defined by:*

$$\frac{\langle \Sigma_{accessible}, \Gamma_{active} \rangle \models \varphi\sigma \text{ and } \Gamma_{active} \neq \emptyset, \quad \Sigma_{accessible} \not\models \psi\sigma, \not\models \neg\psi\sigma}{v \longrightarrow \langle m, \rho_m, \Gamma_m \cup \{\psi\sigma\}, \ldots \langle \Sigma, \Gamma \rangle \ldots \rangle}$$

Module Termination: Intuitively, a module is terminated when its associated active goals are achieved. Upon module termination the module's name is removed from the stack along with the associated substitution and the (in this case empty) set of active goals related to the module. A module thus implements a commitment to the goals introduced by the module which can only be overridden by dropping goals using a **drop** action that is available within the module's program section.

Incidentally, this commitment of a module to achieving the associated goals also explains why the goal condition **goal**`(delivered_order(C))` in the mental state conditions of the conditional actions that were present in Figure 1 can be removed when they are placed inside the module: It may be assumed that this condition holds when the module is active, since the module itself does not introduce any new goals and because the appropriate instantiations for the variable `C` are used while the module is being executed (cf. also Definition 2 which introduces a substitution ρ to record variable instantiations).

The module `deliverOrder` of the example agent of Figure 2 is terminated after loading the truck with ordered items for a specific client, going to the client's site, unloading the ordered items at that location, and returning to home base. This achieves the goal condition of the context `goal(delivered_order(C))` since in that case the agent will believe that it delivered the order and a goal that is believed to be achieved is removed from the agent's goal base (cf. also Definition 2). At that moment, the context condition of the module no longer holds and the module is automatically terminated.

Definition 5. (Module Termination Rule)
Let $v = \langle m, \rho_m, \Gamma_m, \langle m_1, \rho_{m_1}, \Gamma_{m_1}, \ldots \langle \Sigma, \Gamma \rangle \ldots \rangle \rangle$ be a configuration (which contains at least one module instantiation). Then the rule for termination of the most recently activated module m is defined by:

$$\frac{\Gamma_{active} = \emptyset}{v \longrightarrow \langle m_1, \rho_{m_1}, \Gamma_{m_1}, \ldots \langle \Sigma, \Gamma \rangle \ldots \rangle}$$

After terminating the `deliverOrder` module, our example agent will resume execution at the top-level and may reenter the module to process another delivery order.

4 Conclusion

There are similarities between GOAL modules and plans in other agent programming languages (e.g. [6, 8, 15]). The plans referred to are typically part of a plan library that an agent is provided with during design. Both modules as well as such plans specify a condition called the *context* of the module or plan. Such context conditions specify the situation in which the module or plan can be put to good use. This context may in both cases also be used to bind variables through a substitution mechanism that instantiates variables in a module or plan body.

However, a 3APL [6] or AgentSpeak agent [15] that would implement our example agent, it seems, would have to face the same problem of dealing with the multiple goals for delivering orders. Typically, such agent programs trigger plans for achieving declarative goals and in the example discussed multiple plans would be introduced into the agent's plan base. As a result, similar interference effects are to be expected. The difference between GOAL modules and plans resides in the execution of a module and of a plan taken from a plan library. Once activated, a module becomes the *focus of execution* whereas a plan instead is added to the plan base of an agent and just is one of the current, "active" plans that an agent tries to complete.

The approach to incorporate modularization presented in [16] introduces an operator $m(\phi)$ that is applied to goals ϕ that is also motivated to control non-determinism. This operator may also be used to resolve the problem of our example agent. In contrast with the context sections of GOAL modules, however, this operator introduces a *non-declarative* mechanism for activating modules. Also, whereas the termination condition of GOAL modules is based on a commitment strategy that pursues goals until achieved, the termination of the modules in [16] is based on a strategy to try various plans once and in case of failure to quit. Moreover, in order to activate more than one module, calls to such modules need to be explicitly incorporated as steps in a plan.

In this paper, several benefits of using modules in agent programming have been identified. In particular it has been argued that modules provide an elegant solution to *focus the execution* of an agent. Modules restrict both the goals that an agent takes into consideration in a given context as well as the conditional actions that the agent needs to choose from. Modules therefore can be used to reduce the inherent non-determinism present in agent programs. They also provide a tool to structure the flow of control in an agent.

Our module concept is related in various ways to other concepts presented in e.g. [3, 4, 11, 12, 16]. It shares with [12] the idea of relating modules to particular contexts. The idea of a "capability filter" in [11] to constrain adoption of goals is somewhat similar to our notion of a module acting as a context that filters out the active goals. Finally, it shares with [3, 4, 16] the idea of combining the relevant knowledge and skills (actions, plans) to achieve the agent's goals into a module. One of the differences, however, is that modules are not *called* by other modules nor by the agent's plans during run-time, but are *circumstance-triggered* and *focus the attention of the agent* on the situation for which the

module provides a policy. In this sense, our notion of a module is similar to the notion of a policy-based intention in [2].

An interesting idea is to investigate how decision-theoretic notions can be combined with the module concept introduced here. Although GOAL modules provide a tool to qualitatively focus attention on a goal, they do not allow for the consideration of quantitative utilities that indicate preference over such goals. Adding utilities would not provide an agent with a focus on its goals per se, but adding utility-based preferences to GOAL modules may be useful in order to allow preference-based activation of modules as well as to allow an agent to compute optimal ways to execute the plan or policy coded in a module (cf. [1]).

There are several other ideas for future research related to the proposal to view modules as policy-based intentions. In particular, one of the characteristics of such intentions is their *defeasibility* (cf. also [9]). In certain circumstances, the formation of such intentions (in the terminology introduced here, an active module) derived from generic policies (coded as modules here) are *blocked*. Another interesting aspect of policy-based intentions is time-related. The modules proposed and incorporated into GOAL do not allow to distinguish between the *time of adoption* of such an intention and the *time of execution* nor for a notion of *deadline*. It remains for future work to investigate how such extensions can be integrated into GOAL.

References

1. Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-Theoretic, High-level Agent Programming in the Situation Calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 355–362, 2000.
2. Michael E. Bratman. *Intentions, Plans, and Practical Reasoning*. Harvard University Press, 1987.
3. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. In *The 3rd International Workshop on Programming Multiagent Systems (PROMAS-2005)*, pages 139–155, 2005.
4. P. Busetta, N. Howden, R. Ronnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In N. Jennings and Y. Lesperance, editors, *Intelligent Agents VI: Theories, Architectures and Languages*, pages 277–289, 2000.
5. Mehdi Dastani, Frank de Boer, Frank Dignum, and John-Jules Ch. Meyer. Programming Agent Deliberation: An Approach Illustrated Using the 3APL Language. In *Proceedings of The Second Conference on Autonomous Agents and Multi-agent Systems (AAMAS'03)*, pages 97–104, 2003.
6. M.M. Dastani, M.B. van Riemsdijk, F.P.M. Dignum, and J-J.Ch. Meyer. A Programming Language for Cognitive Agents: Goal-Directed 3APL. In M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems (Proc. ProMAS 2003)*, pages 111–130, 2004.
7. F.S. de Boer, K.V. Hindriks, W. van der Hoek, and J.-J.Ch. Meyer. A Verification Framework for Agent Programming with Declarative Goals. *Journal of Applied Logic*, 2006. In Press.

8. M. P. Georgeff and A.L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682. MIT Press, 1987.
9. Guido Governatori and Vineet Padmanabhan. A defeasible logic of policy-based intention. In *Proc. of the 16th Australian Conference on Artificial Intelligence*, LNCS 2903, pages 414–425, 2003.
10. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming with Declarative Goals. In *Proceedings of ATAL00*, volume 1986 of *LNCS*, pages 228–243, 2000.
11. Lin Padgham and Patrick Lambrix. Formalisations of Capabilities for BDI-Agents. *Autonomous Agents and Multi-Agent Systems*, 10:249–271, 2005.
12. Simon Parsons, Nicholas R. Jennings, Jordi Sabater, and Carles Sierra. Agent Specification Using Multi-Context Systems. In *Foundation and Applications of Multi-Agent Systems*, pages 205–226. Springer-Verlag, 2002.
13. G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
14. Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. A Goal Deliberation Strategy for BDI Agent Systems. In T. Eymann et all, editor, *Third German Conference on Multi-Agent Technologies and Systems (MATES 2005)*. Springer, 2005.
15. Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. van der Velde and J.W. Perram, editors, *Agents Breaking Away*, pages 42–55. Springer-Verlag, 1996.
16. M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. Goal-Oriented Modularity in Agent Programming. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1271–1278, 2006.