Adding structure to agent programming languages

Peter Novák and Jürgen Dix

Department of Informatics, Clausthal University of Technology, Germany
{novak,dix}@in.tu-clausthal.de

Abstract. There is a huge gap between agent programming languages used for industrial applications and those developed in academia. While the former are mostly extensions of mainstream programming languages (e.g. Java), the latter are often very specialized languages, based on reactive rules. These specialized languages enjoy clear semantics and come with a number of knowledge representation features, but lack important aspects such as *code re-use*, *modularity*, *encapsulation* etc.
We present a method to extend the syntax of existing specialized agent oriented programming languages to allow more efficient *hierarchical structuring of agent programs*. We illustrate our method through a simple language based on reactive rules. We then gradually extend the core language by several higher level syntactic constructs, thus improving the support for source code modularity and readability.

1 Introduction

While providing a clear and robust theoretical semantics and easy integration of powerful knowledge representation and reasoning techniques, an ideal specialized programming language for agents with mental states *must* also take into account *engineering aspects* of software development as equally important issues. We believe that an *easily readable* syntax of a programming language, allowing conceptual *encapsulation* on the source code level, and support for a program *modularization*, are crucial issues in design of programming languages for cognitive agents. *Such abstraction has to be both (1) a practical means for modular structuring of an agent program source code, as well as (2) a methodological tool guiding translation from an analytical model to a real source code implemented in a rule based programming language.*

In this paper, we demonstrate how the syntax of specialized agent oriented programming languages, based on reactive rules, can be carefully designed to approach the requirements of modern programmers. We understand this work as a work in progress towards development of high level abstract concepts for development of agents with mental states, rather than a proposal for an ultimate solution of this problem.

Our approach follows the tiered approach to programming language design [8] and focuses on introducing *purely syntactical* constructs, rather than making the semantics of the language more complicated. As a basis for further enrichment, we first propose a simple abstract programming language based on reactive rules (Section 2) together with an *interpreter* for it. The main focus of this paper is on gradually extending the core language by several higher level syntactic constructs (Section 3), thus improving the support for source code modularity and readability.

One of the main results in this paper is the introduction of a *mental state transformer (mst)*, an extension based on the functional view on an agent program. We propose a *compiler* transforming a program using the extended syntax into the core language. Discussion on related and future work (sections 4 and 5) conclude the paper.

2 Core programming language

The architecture of specialized programming languages for agents with mental states can be naturally decomposed in two parts. Firstly, the language has to provide means for *modeling the internal structure* of an agent's mental state. Secondly, it has to feature *control structures for encoding transitions* between these states. We are convinced, that these aspects of an agent oriented programming language should be studied separately. In this work, we focus on dynamic aspects of an agent programming language.

The starting point for the design of the core language is the approach applied in the *Modular BDI architecture* [11] and IMPACT [13], where authors abstract from the internal structure of agent's mental state. The structure of a mental state reduces to a black box providing only a query and update interface, while the programming language itself facilitates the control over mental state transitions.

An agent program in the core language consists of a set of reactive rules. Given a query, when a reactive rule evaluates to true in the current mental state, then an update operation is performed on this state. The semantics of a query and update operation is provided by abstract operators, specific to the internal structure of agent's mental states. We also abstract from the interface to agent's environment. It can be handled by queries (sensor interface) and updates (effector interface) integrated in the implementation of the mental state as well (see *Modular BDI architecture* presented in [11]).

The semantics of an agent system is then provided in terms of a transition system over a set of mental states. We give both operational and denotational views on the semantics of the core language. The operational semantics shows the execution of a single primitive construct of the language, while the denotational viewpoint provides a *functional* view on the semantics of an agent program. This view will later turn out to be crucial for modularizing the language. We conclude by detailing the interpreting algorithm and proposing a concrete syntax for the core programming language.

2.1 Abstract syntax

We abstract from the internal structure of mental states by defining them as a theories in a given language $\mathcal{L}$. This abstraction keeps the concept of a mental state modular and allows to examine and update it by means of abstract *query* and *update* operations.

Definition 1. ***(language, formula, query, update)*** *Let $\mathcal{L}$ be a language of mental states. Then a mental state σ is a theory in this language and a formula $\varphi \in \sigma$ is called a mental state formula. Query and update languages $\mathcal{L}_Q$ and $\mathcal{L}_U$ are defined as follows:*

- *if $\varphi \in \mathcal{L}$, then $Q(\varphi) \in \mathcal{L}_Q$ and $U(\varphi) \in \mathcal{L}_U$,*
- *if $\phi_1, \phi_2 \in \mathcal{L}_Q$ then $\phi_1 \wedge \phi_2 \in \mathcal{L}_Q$, $\phi_1 \vee \phi_2 \in \mathcal{L}_Q$ and $\neg\phi_1 \in \mathcal{L}_Q$,*
- *$\top \in \mathcal{L}_Q$, $\bot \in \mathcal{L}_Q$, $nop \in \mathcal{L}_U$*

While update formulas are quite simple (a single update operation at a time) query formulas can be more complex. They can involve conjunctions, disjunctions and negations. Primitive constructs of the language are composed of query and update formulas.

Definition 2. ***(transition rule, agent program)*** *Let $\mathcal{L}_Q$ and $\mathcal{L}_U$ be query and update languages and $\phi \in \mathcal{L}_Q$, $\psi \in \mathcal{L}_U$ be formulas. We say that a rule of the form $\phi \longrightarrow \psi$ is a transition rule. An agent program is a set of rules*

$$\mathcal{P} = \{\phi \longrightarrow \psi |\ \phi \in \mathcal{L}_Q \text{ and } \psi \in \mathcal{L}_U\}$$

composed of query and update formulas from the corresponding query and update languages $\mathcal{L}_Q$ and $\mathcal{L}_U$. We also say that $\mathcal{P}$ is an agent program in $\mathcal{L}$.

2.2 Semantics

Given an agent program, we show first how transition rules are interpreted as single transitions. Then, we also provide denotational semantics for a program to show a functional view of its meaning. The semantics of transition rules is defined in terms of abstract query and update operators. This makes the semantics of the core language modular and independent on the internal structure of agent's mental states.

Definition 3. ***(abstract query/update operators)*** *Let $\mathcal{L}$ be a language of mental states and $\sigma \subseteq \mathcal{L}$ be a mental state (theory) in that language. Let also $\varphi \in \mathcal{L}$ be a formula. An operator $Query_{\mathcal{L}}$ is a mapping*

$$Query_{\mathcal{L}} : \mathcal{L} \times 2^{\mathcal{L}} \rightarrow \{true, false\}; \langle \varphi, \sigma \rangle \mapsto Query_{\mathcal{L}}(\varphi, \sigma)$$

The corresponding update operator $Update_{\mathcal{L}}$ is a mapping

$$Update_{\mathcal{L}} : \mathcal{L} \times 2^{\mathcal{L}} \rightarrow 2^{\mathcal{L}}; \langle \varphi, \sigma \rangle \mapsto Update_{\mathcal{L}}(\varphi, \sigma)$$

We assume $Query_{\mathcal{L}}(\top, \sigma) = true$, $Query_{\mathcal{L}}(\bot, \sigma) = false$, $Update_{\mathcal{L}}(nop, \sigma) = \sigma$.

In the case of a query, the result is the truth value of φ w.r.t. σ. The result of an update operator application is a new mental state $\sigma' \subseteq \mathcal{L}$, which reflects an update φ on σ.

For practical purposes, query and update operators $Query_{\mathcal{L}}$ and $Update_{\mathcal{L}}$ should be computable procedures evaluating formula $\varphi \in \mathcal{L}$ against a theory $\sigma \subseteq \mathcal{L}$.

The semantics of transition rules is defined in terms of query and update operator evaluations. We relate formulas of the form $Q(\varphi)$ and $U(\varphi)$ to applications of corresponding abstract operators $Query_{\mathcal{L}}$ and $Update_{\mathcal{L}}$ to the actual mental state.

Definition 4. ***(semantics of queries and updates)*** *Let $\mathcal{L}$ be a mental state language and $\mathcal{L}_Q$ and $\mathcal{L}_U$ be query and update languages over $\mathcal{L}$. Let also σ be a mental state. Application of a query operator $Query_{\mathcal{L}}$ is denoted by $\models$ and $\oplus$ denotes an application of an update operator $Update_{\mathcal{L}}$ on a mental state. The semantics of a ground query formula $\phi \in \mathcal{L}_Q$ is defined as follows*

- *if $\phi = Q(\varphi)$ and $\varphi \in \mathcal{L}$, then $\sigma \models \phi$ iff $Query_{\mathcal{L}}(\varphi, \sigma) = true$, otherwise $\sigma \not\models \phi$ (i.e. $Query_{\mathcal{L}}(\varphi, \sigma) = false$),*

- *if* $\phi = \neg\phi'$ *and* $\phi' \in \mathcal{L}_Q$*, then* $\sigma \models \phi$ *iff* $\sigma \not\models \phi'$*,*
- *if* $\phi = \phi_1 \wedge \phi_2$ *and* $\phi_1, \phi_2 \in \mathcal{L}_Q$*, then* $\sigma \models \phi$ *iff* $\sigma \models \phi_1$ *and* $\sigma \models \phi_2$*,*
- *if* $\phi = \phi_1 \vee \phi_2$ *and* $\phi_1, \phi_2 \in \mathcal{L}_Q$*, then* $\sigma \models \phi$ *iff* $\sigma \models \phi_1$ *or* $\sigma \models \phi_2$

and for an update formula $\psi \in \mathcal{L}_U$*:*

- *if* $\psi = U(\varphi)$ *and* $\varphi \in \mathcal{L}$*, then* $\sigma \oplus \psi = \sigma'$ *iff* $Update_{\mathcal{L}}(\varphi, \sigma) = \sigma'$.

Note, that we do not define a more powerful notion of negation in query formulas (e.g. default negation): This would require a deeper insight in the structure of mental state. The interpreter of the language only needs to know whether a query is satisfied or not, regardless of the kind of specialized reasoning hidden behind the internal semantics of the query operator.

Definition 5. ***(agent system transition)*** *An agent system moves from* σ *to* σ'*:*

$$\frac{\sigma \models \phi, \sigma \oplus \psi = \sigma'}{\sigma \longrightarrow \sigma'},$$

when $\phi \longrightarrow \psi$ *is an applicable transition rule (i.e.* $\sigma \models \phi$*).*

Finally, we specify a semantics of an agent program in terms of possible evolutions of within the transition system.

Definition 6. ***(agent system: operational view)*** *A computation run* $Comp(\sigma_0)$ *of an agent system over a language of mental states* $\mathcal{L}$*, described by an agent program* $\mathcal{P}$*, is a possibly infinite sequence* $\sigma_0, \ldots, \sigma_n, \ldots$ *of mental states over* $\mathcal{L}$ *(*$\forall i : \sigma_i \subseteq \mathcal{L}$*), so that* $\forall i \geq 0 : \sigma_i \longrightarrow \sigma_{i+1}$ *is an agent system transition induced by some rule* $r_i \in \mathcal{P}$*.*

The agent system is then characterized by the set of all possible computation runs induced by the program $\mathcal{P}$*.*

The operational semantics offers a procedural view on the agent program as a specification of a problem subspace in terms of allowed computation runs. It gives a rather localized perspective on the meaning of a single transition rule w.r.t. to a given agent system evolution. But we can also see a single rule $\phi \longrightarrow \psi$ as a prescription of a transition between classes of mental states. The query part ϕ divides the space of mental states in two classes, according to the truth value of formula ϕ. When the system happens to be in one of the states in which ϕ evaluates to true, the update formula ψ specifies a *direction* in which it should move in the next step. The rule then prescribes a transition from the class of states in which ϕ holds to the set of states resulting from application of update formula ψ to it. This view inspires the alternative semantics of an agent system: *A set of transition rules is a partial function over mental states*.

Definition 7. ***(agent system: denotational view)*** *Let* $\mathcal{L}$ *be a language of mental states and* $\mathcal{P}$ *an agent program in* $\mathcal{L}$*. The program* $\mathcal{P}$ *is characterized by a partial function*

$$\mathcal{F}_{\mathcal{P}} : 2^{\mathcal{L}} \times \mathcal{L}_U \longrightarrow 2^{\mathcal{L}}; \langle \sigma, \psi \rangle \longmapsto \sigma'$$

where $\sigma, \sigma' \subseteq \mathcal{L}$ *are mental states and* $\psi \in \mathcal{L}_U$ *is an update formula.* $\mathcal{F}_{\mathcal{P}}(\sigma, \psi) = \sigma'$ *iff* $\exists(\phi \longrightarrow \psi) \in \mathcal{P}$*, such that* $\sigma \longrightarrow \sigma'$ *is a transition induced by this rule. We say that* $\mathcal{P}$ *is characterized by* $\mathcal{F}_{\mathcal{P}}$*. We also say that the set of states* $\Sigma_{\mathcal{F}_{\mathcal{P}}}$ *over which the partial function* $\mathcal{F}_{\mathcal{P}}$ *is defined is the application domain of* $\mathcal{F}_{\mathcal{P}}$*.*

The function $\mathcal{F}_{\mathcal{P}}$ is a partial function defined only for those mental states in which some rule $r \in \mathcal{P}$ can be applied. I.e. those, in which a query formula of some rule from the program $\mathcal{P}$ is satisfied. It completely characterizes the agent system described by $\mathcal{P}$.

The operational semantics provides a view on an agent program as an explicit characterization of a problem subspace in terms of possible agent system evolutions. Complementary to that, the denotational semantics suggest a specification of the same problem space in terms of a specification of all the considered sets of mental states and all the allowed transitions between them. We stress here, that in essence both provided semantics allow formalization of a same system. They just reflect two different views on its specification. While the first shows how an existing agent program is interpreted, the second offers more methodological insight on how to analyze, create and organize such programs.

2.3 Concrete syntax and interpreter

To complete our core programming system definition we provide a concrete syntax and an interpreter algorithm for it. Our syntax proposal of the core language is straightforward. The EBNF of the core programming language is as follows (white space and string definitions are omitted):

```
<program> := <rules>
<rules>   := <rule> | <rule> <rules>
<rule>    := "when" <queries> "then" <update> ";"
<queries> := <query> | "not" <queries> |
             "(" <queries> "and" <queries> ")" |
             "(" <queries> "or" <queries> ")"
<query>   := "true" | "false" | "query" "[{" <qformula> "}]"
<update>  := "nop" | "update" "[{" <uformula> "}]"
```

`<qformula>` and `<uformula>` are well formed formulas from $\mathcal{L}_Q$ and $\mathcal{L}_U$ respectively. Query and update non-terminals are defined using quite complex bracket delimiters. This is because the syntax of `<string>` non-terminal can be arbitrarily complex and might involve various kinds of character combinations possibly including characters like `{`, or `}`[1].

We finally propose an extremely simple and straightforward interpreting algorithm following the generic scheme applied in other languages like e.g. 3APL. Algorithm 1 lists the detail pseudocode of the core language interpreter.

Given a current mental state, the interpreter first selects all the rules applicable in that state, then non-deterministically chooses one of them and applies its update formula to the actual mental state. The result of this update operation is a new mental state which becomes the current one in the subsequent interpreter iteration. In the case no rule is currently applicable, the interpreter loops and waits until the mental state changes by means of external events. The details of the interpreter algorithm can be found in [10].

[1] In practice, probably additional handling of special character classes would be necessary.

Algorithm 1 run($\sigma_0, \mathcal{P}$)

$\sigma = \sigma_0$
loop
  $\rho = \{(\phi \longrightarrow \psi) \in \mathcal{P} | \sigma \models \phi\}$
  if $\rho \neq \emptyset$ **then**
    non-deterministically choose $(\phi \longrightarrow \psi) \in \rho$
    $\sigma = \sigma \oplus \psi$
  end if
end loop

We assume here, that the selection function which chooses the rule to be applied from the set of all applicable rules conforms to a *fairness* condition (inspired by a similar weak-fairness condition in GOAL [5]):

Condition 1 ***(fairness condition)*** *It is not the case that for a given computation run* $Comp(\sigma_0)$ *a rule* $r \in \mathcal{P}$ *is always applicable from some point in time on and never selected for the execution.*

An interpreter as described above is inherently non-deterministic. It is desirable to allow a programmer to secure a higher degree of determinism in the rule selection, when needed. To this end, we introduce in Subsection 3.1 a simple syntactic extension of the core language facilitating a finer grained control of the rule selection mechanism.

Note also, that our core language is intentionally oversimplified. For the sake of clarity, we did not introduce variables in transition rules and we also define only atomic updates without chaining of update formulas. Although these trivial extensions semantically enhance the language and in practice would be crucial for a practical use, they are not important within the scope of this paper. For introducing such language features we refer to [11], where we introduced them to a language similar to the one discussed here. We also discuss the problems following from introducing variables to the language in Section 5.

Example 1. **(stock exchange agent)** Consider an agent managing its user's stock portfolio. Given a mental state implementation in a Prolog-like language, a simplified program for buying the title MSFT might look like the following:

when *[{ wants(MSFT) }]* **and** *[{ price(MSFT)<avg(MSFT,12h) }]*
then *[{ act(issue_order(buy(MSFT,10))) }]* ;

when *[{ price(MSFT)<max(MSFT,180d) }]* **and** *[{ price(MSFT)<avg(MSFT,7d) }]*
then *[{ introduce_goal(wants(MSFT)) }]* ;

The agent buys the stock MSFT when it knows it wants it and its price falls under the last 12 hours average. Similarly, when the price of the stock is low, according to the agent's analysis, it introduces a desire to buy the stock.

3 Extensions

The core programming language as it is defined in the previous section is still quite rigid. It is hardly imaginable for a programmer to easily manipulate an unstructured

and possibly huge set of reactive rules. Therefore, in this section we propose several extensions of the core language enhancing the flexibility in structuring the source code of an agent program.

To introduce various programming language extensions we follow the tiered approach to design a programming language [8]. This leads to a layered language processing structure: a *core language interpreter* and a *compiler* with integrated *macro preprocessor*. The compiler translates programs written in an extended language, using high level language features, into an equivalent program in the core language.

The tiered structure helps to maintain the simplicity and clarity of the programming language semantics and at the same time allows further extending of the language. The integration of a powerful macro language preprocessor allows a limited support for custom-made language extensions.

Firstly, we introduce the abstraction of *mental state transformer (mst)* inspired by the denotational view on an agent program. Then the *"when-then-else*" construct, extending the mental state transformer syntax, is defined. It facilitates structuring applicable and not applicable rules. For both of these extensions we provide a detailed translational semantics into the core programming language, which should serve as a basis for the language compiler implementation.

Secondly, we propose several extensions based on a *macro expansion mechanism*. The most important one is the construction of a *named mental state transformer*, which utilizes the previously introduced plain mental state transformer extension. Named mental state transformers provide a powerful means for agent program decomposition and modularization. To define the precise meaning of this construct, we also provide a translation to denotational semantics of standard mst's.

Finally, we mention several other, rather trivial, extensions based on macro expansion, which show the way how to further enrich and simplify the programming language syntax. More detailed description can be found in the extended version [10].

3.1 Core language extensions

Mental state transformers (mst) Denotational semantics of agent programs provides a functional view on sets of transition rules: Any set of rules can be considered an agent program of its own. Using this idea, we define a structural decomposition of an agent program in subunits and provide means for composing them into compound structures. According to Definition 7, a set of transition rules is a partial function, transforming a class of mental states to another class of mental states by means of performing updates on them. We call such a set of rules a *mental state transformer (mst)*[2].

Obviously, by unification of two mst's we obtain a new mst, which is defined over a larger class of mental states than the two original ones. Similarly, by specialization of all the query formulas of a set of transition rules, we again obtain a new mst defined on a subclass of mental states of the original one. Hence the agent program source code can be hierarchically structured in terms of compound structures, mst's, which are combined by means of generalization and specialization.

[2] The name *mental state transformer* is inspired by a feature of the language GOAL [5]: however, the semantics is different.

Following the tiered specification approach, we first provide a modified syntax of agent program and subsequently define a translational semantics into the core language syntax, introduced in Subsection 2.3. For clarity, we also provide a constructive denotational semantics (using EBNF).

```
<program>     := <transformer>
<transformer> := <update> ";" |
         "{" <transformer>* "}" |
         "when" <queries> "then" <transformer> ";"
```

We get the extended programming language syntax by replacing the original definition of <program> and adding the definition of <transformer> to the syntax definition from Subsection 2.3. Obviously, the new syntax subsumes the old one. Definitions of <rules> and <rule> are obsolete and replaced by <transformer>.

Abstraction of mental state transformer provides a means for hierarchical nesting of transition rules of the core language. A primitive mst specifies a single update operation ψ. It is a shortcut for the fully expanded transition rule $\top \longrightarrow \psi$, however it helps translating the original syntax of transition rules to that of mst's.

Definition 8. ***(mst: translational semantics)*** *Let τ be a mental state transformer. Then τ is said to be an agent program with mental state transformers and the corresponding core language program $\mathcal{P}$ is constructed as follows*[3]*:*

1. *iff $r \in \tau$ is* <update>*, then "*`when true then` r*" $\in \mathcal{P}$,*
2. *iff $r \in \tau$ is a plain transition rule of the form "*`when` Q `then` U*", where Q and U are* <queries> *and* <update>*, then also $r \in \tau'$,*
3. *iff $r \in \tau$ is "*`when` Q `then` τ'*" where Q is* <queries> *and τ' is a plain set of transition rules, then for each rule $r \in \tau'$ of the form "*`when` Q' `then` U'*" a rule "*`when` Q `and` Q' `then` U'*"$\in \mathcal{P}$. Q and U are* <queries> *and* <update> *respectively.*

For multiply nested mst's, the transformation, specified by Item 3 should be performed bottom-up from the innermost nesting, which contains either a simple update, or a set of plain transition rules. A corresponding denotational semantics for mst's shows how the original notion of agent program is reconditioned.

Definition 9. ***(mst: denotational semantics)*** *Let $\mathcal{L}$ be a language of mental states. A mental state transformer τ is then characterized by a partial function over mental states $\mathcal{F}$ as follows*

1. *primitive mst $\tau = \{\phi \longrightarrow \psi\}$ is characterized by $\mathcal{F}(\sigma, \psi) = Update_{\mathcal{L}}(\psi, \sigma)$, where $\psi \in \mathcal{L}_U$, $\phi \in \mathcal{L}_Q$ and the rule $\phi \longrightarrow \psi$ is applicable in $\sigma \subseteq \mathcal{L}$. The application domain of $\mathcal{F}$ is $\Sigma_{\mathcal{F}} = \{\sigma | \sigma \models \phi\}$.*
2. *if mst τ' is characterized by $\mathcal{F}'$ and $\phi \in \mathcal{L}_Q$ is a query formula, then mst $\tau = \{\phi \longrightarrow \tau'\}$ is characterized by partial function $\mathcal{F}(\sigma, \psi) = \mathcal{F}'(\sigma, \psi)$ with corresponding application domain $\Sigma_{\mathcal{F}} = \{\sigma | \sigma \in \Sigma_{\mathcal{F}'} \wedge \sigma \models \phi\}$.*

[3] For better readability, we omit the syntactic sugar w.r.t. the language EBNF.

3. *if mst's $\tau'$and τ'' are characterized by $\mathcal{F}'$ and $\mathcal{F}''$ correspondingly, then mst $\tau = \tau' \cup \tau''$ is characterized by* $\mathcal{F}(\sigma,\psi) = \begin{cases} \mathcal{F}'(\sigma,\psi) & \text{if } \sigma \in \Sigma_{\mathcal{F}'} \\ \mathcal{F}''(\sigma,\psi) & \text{if } \sigma \in \Sigma_{\mathcal{F}''} \end{cases}$ *with the corresponding application domain* $\Sigma_{\mathcal{F}} = \Sigma_{\mathcal{F}'} \cup \Sigma_{\mathcal{F}''}$.

A simple rule is a *primitive* mst (Item 1). Primitive elements can be combined to *compound* mst's by means of *generalization* (Item 3) and *specialisation* (Item 2). Note, that according to Definition 8, Item 1, a simple update formula ψ serves as a shortcut for a trivial rule $\top \longrightarrow \psi$: a plain update formula is the most primitive mst.

An agent program is also a mst. The concept of mental state transformer provides a functional view on an agent program as composed of conceptually encapsulated subunits, which are again composed of lower level subunits (mst's) of the same type.

When-then-else When in a given mental state several transition rules are applicable, the interpreter is supposed to non-deterministically choose one of them. A developer might need to restrict and narrow the choice of the interpreter's selection function. In the core language, this can be done by writing complex queries, so that the number of applicable rules in a certain mental state is minimized and in turn, the number of states in which a rule is applicable is minimized as well.

The abstraction of the mental state transformer introduced nested rules, allowing a developer to restrict the scope of applicability of the inner rule by the query of the outer one (*"when-then"* construct). As we already said above, according to its validity, a query divides a set of mental states to two classes. By a trivial extension of the *"when-then"* construct to handle also the *"-else"* branch, the programmer gets a means to specify mental state transformers for both of them. This helps to narrow the interpreter's choice using a compact syntax.

```
<transformer> := "when" <queries>
          "then" <transformer>
          "else" <transformer>
```

Definition 10. ***(mst: translational semantics cont.)***

4. *iff $r \in \tau$ is "*`when` Q `then` τ' `else` τ'' *", where Q is* `<queries>` *and τ', τ'' are plain sets of transition rules, then for each rule $r \in \tau'$ of the form "*`when` Q' `then` U'*" a rule "*`when` Q `and` Q' `then` U'*"$\in \mathcal{P}$. Similarly for each rule $r \in \tau''$ a rule "*`when` $\neg Q$ `and` Q' `then` U'*"$\in \mathcal{P}$. Q', U' are* `<queries>` *and* `<update>` *respectively.*

By introducing sequences of nested rules of the form "`when` Q_1 `then` τ_1 `else when` Q_2 `then...` `else when` Q_n `then` τ_n`;`" a programmer gradually restricts the choice of the interpreter using a compact syntax without annoying repetitions. An interesting consequence of using *"when-then-else*" construct is that the program can be read in a sequential way, although it is not sequential in nature.

Example 2. **(stock exchange cont.)** We modify the agent program from Example 1 to drop the goal to buy stock when there is a market turmoil going on. Otherwise it should behave as in Example 1.

when *[{ news('overtake')>2 }]* **and** *[{ avg(DOW,5h)<0.70∗avg(DOW,2d) }]*
then *[{ drop_goal(wants(MSFT)) }]*
else { *%% Example 1 code %%* } ;

Market turmoil is defined as a state, when at least two news about a company overtake arise and market index average in the last five hours falls more than 30% under the last two days average. Note, that in the core language, the equivalent program would require three separate transition rules.

3.2 Macro extensions

As we already indicated at the beginning of this section, we propose integrating a macro language into the compiler. In the following, we introduce several extensions exploiting a macro expansion. In practice we have in mind employing a robust macro preprocessor like e.g. GNU M4[4].

Named mental state transformers The concept of a plain mst introduced modularity into an agent program, however it does not allow an easy re-use of already defined mst's in different contexts of the agent program. The extension to a *named mental state transformer* provides a means to re-use previously defined mst's in different contexts of an agent program. A label (handle) of a named mst serves as a placeholder for it. It is expanded into a full-fledged code by a macro preprocessor.

Again, following the tiered approach, we first provide a syntactical specification followed by a detailed translation of named mst into the denotational semantics of plain mst. The syntax of the programming language is extended by the following EBNF:

```
<program>     := <trans_def>* <transformer>
<trans_def>   := "define" <identifier> <transformer>
<transformer> := <identifier>
```

`<program>` is again redefined, and the rest of the definition extends the previous ones. `<identifier>` should be a unique label, distinct from the already introduced keywords like `query`, `update` etc. A straightforward denotational semantics of the extended definition of agent program in terms of simple mst's follows.

Definition 11. ***(named mst: denotational semantics)*** *A modified mst construct is defined by adding the following to Definition 9: Let τ be a mst and $label$ is a unique identifier (`<identifier>`), then $(label, \tau)$ is a named mst definition.*

5. *If $(label, \tau')$ is a named mst definition and τ' is characterized by a partial function $\mathcal{F}'$, then mst $\tau = \{(label)\}$ is characterized by $\mathcal{F}(\sigma, \psi) = \mathcal{F}'(\sigma, \psi)$ with the application domain $\Sigma_{\mathcal{F}} = \Sigma_{\mathcal{F}'}$.*

Note, that because the Definition 11 is an extension of the Definition 9 also mst of the form $\phi \longrightarrow (label)$ is a well formed mst defined as a specialisation of mst $(label)$ by the query formula ϕ.

[4] `http://www.gnu.org/software/m4/`

Now we provide a translation of the extended mst construct to the plain mst as defined in Definition 9.

Definition 12. ***(expanded mst)*** *Let $\mathcal{L}$ be a language of mental states, Γ be a set of named mst definitions in $\mathcal{L}$ and τ be a mst. We define $Exp(\tau)$, the expansion operator:*

$$Exp(\tau) = \begin{cases} \tau & \text{if } \tau = \{\phi \longrightarrow \psi\}, \text{ where } \phi \in \mathcal{L}_Q, \psi \in \mathcal{L}_U \\ \{\phi \longrightarrow Exp(\tau')\} & \text{if } \tau = \{\phi \longrightarrow \tau'\}, \tau' \text{ is not primitive and } \phi \in \mathcal{L}_Q \\ \bigcup_{\tau' \in \tau} Exp(\tau') & \text{if } \tau \text{ is a union of several mst's} \\ Exp(\tau') & \text{if } \tau = (label) \text{ and } (label, \tau') \in \Gamma \end{cases}$$

Now the expansion fixed point is as usual: $Exp^0(\tau) = \tau$, and $Exp^{i+1}(\tau) = Exp(Exp^i(\tau))$. The expanded mst τ_e, corresponding to τ, is a fixed point of the Exp operator. I.e. such a mst τ_e, for which $\exists i \geq 0$, so that $\tau_e = Exp^i(\tau) = Exp^{i+1}(\tau)$.

The semantics of an agent program $\mathcal{P} = (\Gamma, \tau)$ with a set of named mst definitions Γ is that of the expanded mst τ_e corresponding to τ w.r.t. Γ.

The expansion operator *Exp* simply replaces all the labels by their corresponding content according to their definitions. For an agent program to expand correctly, each label, used as a placeholder for a mst, has to be previously defined in the agent program $\mathcal{P}$ as well. Recursive schemata of mst "calls" do not correctly expand into a simple program without labels, because the fixed point w.r.t. *Exp* operator, does not exist for them. Recursive applications of named mst's in the agent program, if allowed, would also lead to infinite query evaluation[5].

Note also, that due to uniqueness of labels of named mst's, there is *at most one* fixed point of *Exp* operator for any program $\mathcal{P} = (\Gamma, \tau)$. Obviously not all agent programs with syntax extended to named mst's have a semantics accroding to *Exp* expansion operator. This might happen when the program uses recursive application of a named mst, or when it uses a previously undefined mst w.r.t. given Γ.

Named queries, code templates and more To conclude our tour through gradual extensions of the core programming language, we finally sketch several simple extensions, which further enrich the language and stand as an inspiration for implementation of the language compiler.

As we already mentioned several times above, queries in transition rules can be seen as mental state classifiers. It might be practical to re-use these classifiers in different contexts and specialize them in different parts of an agent program. For that, we can again use the macro expansion facility of the language compiler. A *named query* can be viewed as an abbreviation for a complex query formula:

```
<query_def> := "defineq" <identifier> "{" <queries> "}"
<query>     := <identifier>
```

[5] Except for external events, which we abstract from, a mental state cannot be changed unless an update is performed. In turn, this cannot happen either, because the interpreter cannot properly perform a query on it.

This definition again extends the definition of the core language syntax introduced in Subsection 2.3. We assume unique query abbreviation identifiers. A formal definition of named query expansion is similar to that of named mst: our comments w.r.t. recursive application and the existence of fixed points of the expansion operator apply as well.

Many more trivial extensions based on macro expansion can be introduced. We only briefly list some of those, which we believe contribute to improving the coding experience using a reactive rule based programming language, such as the one defined here. Named updates, or definition of re-usable modules, consisting of several named mst's, with features resembling name spaces, will further enhance modularization of an agent program. Parametrized macro definitions and their further extensions to syntactical constructs resembling lambda-calculus of Lisp will lead to implementation of re-usable code templates, similar to those of C++.

We conclude this section with an example sketching a part of an agent program using some of the features above.

Example 3. **(stock exchange cont.)** Parametrized mst definitions allow us to reformulate and modify the code from Example 2 to implement specific strategy w.r.t. certain stock title. Different variants of strategies for different stocks can be used in different situations. Use of a named query definition further improves the code readability as well.

```
define careful_strategy(TITLE) {
    when [{ wants(TITLE) }] then [{ drop_goal(wants(TITLE)) }] ;
}
define opportunistic_strategy(TITLE) {
    %% Adapted code from Example 1 %%
}
defineq market_turmoil {
    [{ news('overtake')>2 }] and [{ avg(DOW,5h)<0.70*avg(DOW,2d) }]
}
...
when market_turmoil then {
    careful_strategy(APPL);
    careful_strategy(MSFT);
} else {
    opportunistic_strategy(APPL);
    opportunistic_strategy(MSFT);
}
```

The last rule clearly summarizes the meaning of the whole program in a very compact and easily readable statement.

4 Discussion and related work

Reactive planning is an important paradigm that led to implementations using reactive rules in languages like AgentSpeak(L) [12,2], 3APL [6,4], GOAL [5], or the one introduced here.

Interpreters of these languages in every step select a rule and then execute it w.r.t. semantics of the particular language. In general, in each cycle the interpreter considers a set of rules independently of the previously selected and executed rules (doing some bookkeeping within the internal structure of agent's mental state). The resulting agent system is then able to flexibly react to events and exceptional situations without reconsidering its previous actions.

However, designers of an agent oriented programming language face a very difficult problem: Such a reactive architecture of an agent program clashes with the traditional sequential and imperative view on the program code.

We argue, that the functional view on an agent program

1. can be appropriately represented by an abstraction called *mental state transformer*,
2. has a potential to become a basis for a powerful abstraction useful for conceptual decomposition of an agent program into functionally encapsulated subunits (higher level units "call" lower level ones, which allows structuring the agent program into several conceptually separated layers), and
3. is particularly appropriate in the context of programming languages for agents with mental states.

Instead of considering an agent program as a specification of all the paths along which the agent system is allowed to evolve within its transition system, this abstraction shifts the programming style to consider different contexts in which the agent might be in. Each such context forms a subspace of the agent's transition system and it might again consist of a number of smaller subspaces, in each of which the agent performs a different behaviour, i.e. different mental state update.

The concept of mental state transformer favours this subspace-nesting view on the specification of an agent system by nesting queries of transition rules, finally resulting in a mental state update. In the previous sections, we tried to demonstrate how this view can be used to conceptually decompose an agent program into functionally encapsulated subunits. We stress, that this work should be perceived more as a basis for further development of strong abstractions for agent oriented programming, rather then an ultimate solution of this problem.

Most probably, 3APL [6,4] is the rule based language which received the most attention w.r.t. agent program modularity. Recent works by Dastani et. al. [3] and by van Riemsdijk et. al. [14] introduce a semantically oriented modularity to 3APL. In [3] the authors formalize a notion of *role*, grouping together beliefs, goals, plans and reasoning rules of a BDI agent. A role can be enacted, or deacted at run-time. The whole process is handled by 3APL's deliberation cycle.

In [14], the authors introduce a concept of *goal oriented modularity* for 3APL. It is based on decomposition of a set of practical reasoning rules of a BDI agent into modules, according to goals they help to achieve. A module can be called within a rule to achieve a subgoal in the context of a plan. When the subgoal is achieved, the control returns back to the context from which the module was called. This resembles a stack of routine calls in procedural languages. Implementation of both of these 3APL extensions requires modification of 3APL's semantics and the language interpreter.

Both role and goal oriented approaches to modularize an agent program are based on particularities of the internal structure of agent's mental state, namely BDI architecture. As our approach introduces a functional modularity, supported by *purely syntactic extensions* of the language, they can be seen as orthogonal to ours and, we believe, can be combined. A combination of modularization of practical reasoning rules, based on the abstraction of mst's, within the role, or a goal oriented module, can lead to a finer grained structuring of agent programs.

In [7] plan patterns for programming declarative goals in AgentSpeak(L) ([2]) are introduced. While their approach is similar to ours in that it exploits a macro preprocessor as well, in [7] the authors describe use of this mechanism only for implementation of code templates, similar to those we discuss in Subsection 3.2, for handling various types of goals. In this paper, we propose a *functional view of an agent program*, embodied in the concept of mental state transformer, which has an ambition to become a basis for further development of code templates implementing also agent's behaviours, or roles.

Finally, according to our personal communication with Koen V. Hindriks, there's an ongoing work on policy based modularization of GOAL [5].

IMPACT [13] and *Modular BDI architecture* [11] introduce a vertical modularity to agent programming. The programming language in which a developer encodes how an agent system should move from one mental state to another using updates, is in these approaches *independent of the internal structure* of agent's mental states. A programmer is free to choose a knowledge representation technique to employ and develop the agent's mental state representation in it. She can also agentize 3rd party legacy code like e.g. mainstream database systems. These two approaches inspired the design of our core programming language. It allowed us to study modularization of an agent program independently of intricacies of the architecture of an agent's mental state.

This paper is an attempt to engineer a practical syntax for the *Modular BDI architecture* introduced previously in [11]. A more thorough discussion on applicability of our approach to other agent oriented programming frameworks is given in [10].

5 Conclusion and future work

The contribution of this paper is an attempt to give an answer to the following question:

> *Given an (unstructured) agent language based mainly on reactive rules, how can the syntax be extended so that important features allowing code re-use, modularization and the like are available?*

To this end we introduced a *novel* abstraction: the *mental state transformer*.

We did not yet touch the important issue of *variables* in our language. Using variables broadens our approach significantly and enhances its applicability. The problem with allowing variables in the rules, is that the implementation of the notion of named mst using macros is not sufficient any more. In such an extended language the *name scope of variables* has to be considered, i.e. variables used in named mst, should be *local to that mst*. A *customized macro preprocessor*, which handles local variables has to be used in such a case. We are currently developing such preprocessor.

While we discussed in this paper only the theoretical basis, we are currently working on an implementation of an interpreter-compiler stack for the programming language similar to the one proposed here. We hope to refine some of the extensions of the language using macro expansion and to experiment with the resulting language, in order to put the abstraction of mst's to a test. As the structural decomposition, introduced in this paper, leads to a *new programming style*, it is necessary to prove the usefulness of the presented language in practice by developing a non-trivial agent system application.

Finally we would like to thank several anonymous referees for their careful reading. Their comments helped to improve this paper a lot.

References

1. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2005.
2. Rafael H. Bordini, Jomi F. Hübner, and Renata Vieira. *Jason and the Golden Fleece of Agent-Oriented Programming*, chapter 1, pages 3–37. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [1], 2005.
3. Mehdi Dastani, Birna van Riemsdijk, Joris Hulstijn, Frank Dignum, and John-Jules Ch. Meyer. Enacting and deacting roles in agent programming. In James Odell, Paolo Giorgini, and Jörg P. Müller, editors, *AOSE*, volume 3382 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2004.
4. Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Meyer. *Programming Multi-Agent Systems in 3APL*, chapter 2, pages 39–68. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [1], 2005.
5. Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. *CoRR*, cs.AI/0207008, 2002.
6. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
7. Jomi Fred Hübner, Rafael H. Bordini, and Michael Wooldridge. Programming declarative goals using plan patterns. In Matteo Baldoni and Ulle Endriss, editors, *DALT*, volume 4327 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.
8. Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990.
9. Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors. *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*. ACM, 2006.
10. Peter Novák and Jürgen Dix. Adding structure to agent programming languages. Technical Report IfI-06-12, Clausthal University of Technology, 2006.
11. Peter Novák and Jürgen Dix. Modular BDI architecture. In Nakashima et al. [9], pages 1009–1015.
12. Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In Walter Van de Velde and John W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.
13. V. S. Subrahmanian, Piero A. Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogenous Active Agents*. MIT Press, 2000.
14. M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. Goal-oriented modularity in agent programming. In Nakashima et al. [9], pages 1271–1278.