Code Patterns for Agent-Oriented Programming

Peter Novák and Wojciech Jamroga

Department of Informatics
Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal-Zellerfeld, Germany

ABSTRACT

The mainstream approach to *design of BDI-inspired agent programming languages* is to choose a set of agent-oriented features with a particular semantics and their subsequent implementation in the programming language interpreter. The language designer's choices thus impose strong constraints on the architecture of the implemented agents as well as only a limited toolbox of high-level language constructs for encoding the agent program.

As an alternative, we propose a *purely syntactic* approach to designing an agent programming language. On the substrate of *Behavioural State Machines* (*BSM*), a generic modular programming language for hybrid agents, we show how an agent designer can implement high-level agent-oriented constructs in the form of *code patterns* (macros). To express the semantics of agent programs in the logic-agnostic programming language of *BSM*, we propose *LTL* program annotations and subsequently introduce *DCTL**, an extension of the *CTL** logic with features of dynamic logic, for reasoning about traces of *BSM* program executions. We show how *DCTL** specifications can be used to prove relevant properties of code patterns. Moreover, *DCTL** allows for natural verification of *BSM* agent programs.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*

General Terms

Focus: Agent programming languages; Inspiration source: Robotics, AI; Description level: Methodologies and Languages

Keywords

Agent-oriented programming, code patterns, temporal logic

Cite as: Code Patterns for Agent-Oriented Programming, Peter Novák and Wojciech Jamroga, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. XXX-XXX.

1. INTRODUCTION

Since the Shoham's seminal paper on agent-oriented programming [20], one of the high ambitions of the agents programming community is development of a *theoretically founded programming framework* enabling creation of cognitive agents, i.e. agents with mental states [20], also coined knowledge-intensive agents [12]. A programming language is an engineering tool in the first place and thus it has to provide a toolbox for development of practical systems. On the other hand, to support the design process, as well as to allow a deeper insight into system functionality, it is desirable to establish a tight relationship of the language with a formal framework for reasoning about programs written in it.

To bridge the gap between pragmatics of software engineering and theoretical foundations, BDI-inspired agent programming languages (such as *AgentSpeak(L)/Jason*, *3APL*, *GOAL*, etc., cf. e.g. [1, 2] for more details), provide a particular set of agent-oriented features and bind them to a rule-based computational model of reactive planning. The language designer's choices thus impose constraints on the resulting design of agent applications. The state-of-the-art languages enforce a fixed internal architecture of the agent, usually a fixed set of knowledge bases with a fixed knowledge representation (KR) technology, as well as a fixed implementation of language constructs for agent's beliefs and/or goals, and their mutual relationships. The main benefit from constraining agent designers in so many ways is usually a clear, theoretically sound Plotkin style [16] operational semantics of the language, traditionally provided in terms of computation runs (traces) in a transition system over the agent's mental states. However, a tight formal relationship with a reasoning framework for agent behaviours, such as that by Rao and Georgeff [18] or Cohen and Levesque [4], is only rarely established (see Section 6 for a discussion).

In this paper, we put forward an alternative approach to design of agent-oriented programming languages. On the level of a generic language for programming reactive systems, we propose development of a library of *code patterns* (macros, templates) whose semantics refers to various agent-oriented concepts, such as an *achievement goal* or a *maintenance goal*. The generic language of choice is the framework of *Behavioural State Machines* (*BSM*) [14] allowing for an application-specific architecture of an agent system in terms of knowledge representation modules exploiting the potential of heterogeneous KR technologies. In the *BSM* framework, an agent program is encoded in terms of *mental state transformers*, i.e., nested sub-programs connected by composition operators. The hierarchical structure of subprograms allows to define and instantiate macros which implement encapsulated and clearly defined agent-oriented concepts, such as a specific type of a goal. For proving that the execution of instances of such macros indeed satisfies properties of agent-specific concepts, we introduce *Dynamic CTL** (*DCTL**), a novel extension of the full branching time temporal logic *CTL** [9] with features of Harel's *Dynamic Logic* [10]. Finally, to bridge the gap between the flexible but logic-agnostic programming framework of *Behavioural State Machines* and *DCTL**, the logic for verification of *BSM* programs, we propose *program annotations* in the form of formulae of *Linear Time Temporal Logic* (*LTL*) [17]. Since *DCTL** is a proper extension of *LTL*, we in fact suggest that it is often convenient to reason about programs in a richer language than the one used for specification of program behaviour.

The contribution of this work is twofold: 1) we demonstrate an alternative approach to design of an agent-oriented programming language equipped with a library of high-level agent-oriented constructs, which an agent developer can further extend according to the specific application needs; and 2) to enable reasoning about programs in the *BSM* framework, we introduce a logic for their verification and proving their properties.

We begin with a brief description of the framework of *Behavioural State Machines* in Section 2. In Section 3, we discuss logics *LTL* and *DCTL** with an interpretation over the semantic structures of *BSM* programs. Program annotations, introduced in Section 4, bridge the gap between the logic-agnostic programs of the *BSM* framework and the temporal logics for reasoning about their executions. Section 5 introduces a number of code patterns implementing agent-oriented notions of achievement and maintenance goal similar to the concept of *persistent relativised goal* (P-R-GOAL) by Cohen and Levesque [4]. Section 6 concludes the paper with a discussion and a brief overview of the relevant work.

2. BEHAVIOURAL STATE MACHINES

In [14], we introduced the framework of *Behavioural State Machines* (*BSM*). Because of the generic and modular nature of its computational model, the framework is particularly suitable to study code patterns in programming languages. *BSM* draws a clear distinction between the *knowledge representation* and *behavioural* layers within an agent. It thus provides a programming framework that clearly separates the programming concerns of *how to represent an agent's knowledge* about, for example, its environment and *how to encode its behaviours.* Below, we briefly introduce the language of *BSM*'s without variables. For the complete formal description of the *BSM* framework, see [14].

2.1 Syntax

BSM agents are collections of one or more so-called *knowledge representation modules* (KR modules), typically denoted by $\mathcal{M}$, each representing a part of the agent's knowledge base. KR modules may be used to represent and maintain various mental attitudes of an agent, such as knowledge about its environment, or its goals, intentions, obligations, etc. Transitions between states of a *BSM* result from applying so-called *mental state transformers* (*mst*), typically denoted by τ. Various types of mst's determine the behaviour that an agent can generate. A *BSM agent* consists of a set of KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and a mental state transformer $\mathcal{P}$, i.e. $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$; the mst $\mathcal{P}$ is also called an *agent program.*

The notion of a KR module is an abstraction of a partial knowledge base of an agent. In turn, its states are to be treated as theories (i.e., sets of sentences) expressed in the KR language of the module. Formally, a KR module $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$ is characterized by a knowledge representation language $\mathcal{L}_i$, a set of states $\mathcal{S}_i \subseteq 2^{\mathcal{L}_i}$, a set of query operators $\mathcal{Q}_i$ and a set of update operators $\mathcal{U}_i$. A query operator $\models \in \mathcal{Q}_i$ is a mapping $\models : \mathcal{S}_i \times \mathcal{L}_i \to \{\top, \bot\}$. Similarly an update operator $\oplus \in \mathcal{U}_i$ is a mapping $\oplus : \mathcal{S}_i \times \mathcal{L}_i \to \mathcal{S}_i$.

Queries, typically denoted by φ, can be seen as operators of type $\models : \mathcal{S}_i \to \{\top, \bot\}$. A primitive query $\varphi = (\models\phi)$ consists of a query operator $\models \in \mathcal{Q}_i$ and a formula $\phi \in \mathcal{L}_i$ of the same KR module $\mathcal{M}_i$. We will use $\mathcal{Q}(\mathcal{A}) = \bigcup_{i=1}^{n} \mathcal{Q}_i \times \mathcal{L}_i$ to denote the set of primitive queries of *BSM* $\mathcal{A}$. Complex queries can be composed by means of conjunction $\wedge$, disjunction $\vee$ and negation $\neg$.

Mental state transformers enable transitions from one state to another. A primitive mst $\oslash\psi$, typically denoted by ρ and constructed from an update operator $\oslash \in \mathcal{U}_i$ and a formula $\psi \in \mathcal{L}_i$, refers to an update on the state of the corresponding KR module. We use $\mathcal{U}(\mathcal{A}) = \bigcup_{i=1}^{n} \mathcal{U}_i \times \mathcal{L}_i$ to denote the set of primitive mst's of $\mathcal{A}$. Conditional mst's are of the form $\varphi \longrightarrow \tau$, where φ is a query and τ is a mst. Such a conditional mst makes the application of τ depend on the evaluation of φ. Syntactic constructs for combining mst's are: non-deterministic choice | and sequence $\circ$.

Definition 1 (mental state transformer). *Let* $\mathcal{M}_1, \ldots, \mathcal{M}_n$ *be KR modules of the form* $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$. *The set of* mental state transformers *is defined as below:*

1. **skip** *is a* primitive *mst,*
2. *if* $\oslash \in \mathcal{U}_i$ *and* $\psi \in \mathcal{L}_i$, *then* $\oslash\psi$ *is a* primitive *mst,*
3. *if* φ *is a query, and* τ *is a mst, then* $\varphi \longrightarrow \tau$ *is a* conditional *mst,*
4. *if* τ *and* τ' *are mst's, then* $\tau|\tau'$ *and* $\tau \circ \tau'$ *are mst's (*choice, *and* sequence *respectively).*

2.2 Denotational Semantics

The *yields* calculus, summarised below after [14], specifies an update associated with executing a mental state transformer in a single step of the language interpreter. It formally defines the meaning of the state transformation induced by executing an mst in a state, i.e., a mental state transition.

Formally, a *mental state* σ of a *BSM* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ is a tuple $\sigma = (\sigma_1, \ldots, \sigma_n)$ of states $\sigma_1 \in \mathcal{S}_1, \ldots, \sigma_n \in \mathcal{S}_n$, corresponding to modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$, respectively. $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ denotes the space of all mental states over $\mathcal{A}$. A mental state can be modified by applying primitive mst's on it, and query formulae can be evaluated against it. The semantic notion of truth of a query is defined through the satisfaction relation $\models$. A primitive query $\models\phi$ holds in a mental state $\sigma = (\sigma_1, \ldots, \sigma_n)$, written $\sigma \models (\models\phi)$, iff $\models(\phi, \sigma_i)$; otherwise we have $\sigma \not\models (\models\phi)$. Given the usual meaning of Boolean operators, it is straightforward to extend the query evaluation to compound query formulae. Note that evaluation of a query does not change the mental state σ.

For an mst $\oslash\psi \in \mathcal{U}(\mathcal{A})$, we use $(\oslash, \psi)$ to denote its semantic counterpart, i.e., the corresponding *update* (state transformation). Sequential application of updates is denoted by

$\bullet$, i.e. $\rho_1 \bullet \rho_2$ is an update resulting from applying ρ_1 first and then applying ρ_2. The application of an update to a mental state is defined formally below.

Definition 2 (applying an update). *The result of applying an update $\rho = (\oslash, \psi)$ to a state $\sigma = (\sigma_1, \ldots, \sigma_n)$ of a* BSM *$\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$, denoted by $s \bigoplus \rho$, is a new state $\sigma' = (\sigma_1, \ldots, \sigma_i', \ldots, \sigma_n)$, where $\sigma_i' = \sigma_i \oslash \psi$ and σ_i, $\oslash$, and ψ correspond to one and the same $\mathcal{M}_i$ of $\mathcal{A}$. Applying the empty update* **skip** *on the state σ does not change the state, i.e. $\sigma \bigoplus \mathbf{skip} = \sigma$.*

Inductively, the result of applying a sequence of updates $\rho_1 \bullet \rho_2$ is a new state $\sigma'' = \sigma' \bigoplus \rho_2$, where $\sigma' = \sigma \bigoplus \rho_1$. $\sigma \overset{\rho_1 \bullet \rho_2}{\rightarrow} \sigma'' = \sigma \overset{\rho_1}{\rightarrow} \sigma' \overset{\rho_2}{\rightarrow} \sigma''$ denotes the corresponding compound transition.

The meaning of a mental state transformer in state σ, formally defined by the *yields* predicate below, is the update set it yields in that mental state.

Definition 3 (yields calculus). *A mental state transformer τ yields an* update *ρ in a state σ, iff $yields(\tau, \sigma, \rho)$ is derivable in the following calculus:*

$$\frac{\top}{yields(\mathbf{skip},\sigma,\mathbf{skip})} \qquad \frac{\top}{yields(\oslash\psi,\sigma,(\oslash,\psi))} \qquad \textit{(primitive)}$$

$$\frac{yields(\tau,\sigma,\rho),\ \sigma\models\phi}{yields(\phi\longrightarrow\tau,\sigma,\rho)} \qquad \frac{yields(\tau,\sigma,\theta,\rho),\ \sigma\not\models\phi}{yields(\phi\longrightarrow\tau,\sigma,\mathbf{skip})} \qquad \textit{(conditional)}$$

$$\frac{yields(\tau_1,\sigma,\rho_1),\ yields(\tau_2,\sigma,\rho_2)}{yields(\tau_1|\tau_2,\sigma,\rho_1),\ yields(\tau_1|\tau_2,\sigma,\rho_2)} \qquad \textit{(choice)}$$

$$\frac{yields(\tau_1,\sigma,\rho_1),\ yields(\tau_2,\sigma\bigoplus\rho_1,\rho_2)}{yields(\tau_1\circ\tau_2,\sigma,\rho_1\bullet\rho_2)} \qquad \textit{(sequence)}$$

We say that τ yields an update set *ν in a state σ iff $\nu = \{\rho | yields(\tau, \sigma, \rho)\}$.*

The mst **skip** yields the update **skip**. Similarly, a primitive update mst $\oslash\psi$ yields the corresponding update $(\oslash, \psi)$. In the case the condition of a conditional mst $\phi \longrightarrow \tau$ is satisfied in the current mental state, the calculus yields one of the updates corresponding to the right hand side mst τ, otherwise the no-operation **skip** update is yielded. A non-deterministic choice mst yields an update corresponding to either of its members and finally a sequential mst yields a sequence of updates corresponding to the first mst of the sequence and an update yielded by the second member of the sequence in a state resulting from application of the first update to the current mental state.

Definition 4 (BSM semantics). *A* BSM *$\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$ can make a step from state σ to a state σ', if the mst τ yields a non-empty update set ν in σ and $\sigma' = \sigma \bigoplus \rho$, where $\rho \in \nu$ is an update. We also say, that $\mathcal{A}$ induces a (possibly compound) transition $\sigma \overset{\rho}{\rightarrow} \sigma'$.*

2.3 Operational View

The underlying semantics of *BSM* can be seen in terms of traces within a labeled transition system over agent's mental states. Mental state transformers are interpreted as traces in a transition system over agent's mental states and transitions induced by updates. The notion of a *behavioural frame* formally captures the semantic structure induced by the set of KR modules of a *BSM* agent. In particular, it encapsulates the set of all mental states constructed from local states of the KR modules and applications of the corresponding update operators between them.

Definition 5 (behavioural frame). *Let $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ be a* BSM *over a set of KR modules $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$. The* behavioural frame *of $\mathcal{A}$ is a labeled transition system $LTS(\mathcal{A}) = (\mathcal{S}, \mathcal{R})$, where $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ is the set of mental states of $\mathcal{A}$, and the transition relation $\mathcal{R}$ is defined as follows:*

$$\mathcal{R} = \{\sigma \overset{\rho}{\rightarrow} \sigma' \in \mathcal{S} \times \mathcal{U}(\mathcal{A}) \times \mathcal{S} \mid \sigma' = \sigma \bigoplus \rho\} \cup$$
$$\{\sigma \overset{\mathbf{skip}}{\rightarrow} \sigma \in \mathcal{S} \times \mathcal{U}(\mathcal{A}) \times \mathcal{S}\}$$

A tuple of KR modules $\overline{\mathcal{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n)$, which is essential for constructing the behavioural frame is also called behavioural template. *We will sometimes write $LTS(\overline{\mathcal{A}})$ instead of $LTS(\mathcal{A})$ since the mst $\mathcal{P}$ in $\mathcal{A}$ plays no role in the construction of the corresponding frame.*

Note that $LTS(\mathcal{A})$ is finite (resp. enumerable) iff all the modules in $\mathcal{A}$ have finite (resp. enumerable) state spaces, languages, and repertoires of query and update operators.

The operational semantics of an agent is defined in terms of all possible computation runs induced by the iterated execution of the corresponding *BSM*. Let $\lambda = \sigma_0\sigma_1\sigma_2\ldots$ be a *trace* (finite or infinite). Then, $\lambda[i] = \sigma_i$ denotes the ith state on λ, and $\lambda[i..j] = \sigma_i \ldots \sigma_j$ denotes the "cutout" from λ from position i to j. The ith prefix and suffix of λ are defined by $\lambda[0..i]$ and $\lambda[i..\infty]$, respectively.

Definition 6 (traces and runs of a BSM). *Let $\mathcal{A} = (\overline{\mathcal{A}}, \tau)$ be a* BSM. *$\mathcal{T}(\mathcal{A})$ denotes the set of (complete) traces $\sigma_0\sigma_1\ldots\sigma_k$, such that $\sigma_0 \overset{\rho_1}{\rightarrow} \sigma_1 \overset{\rho_2}{\rightarrow} \ldots \overset{\rho_k}{\rightarrow} \sigma_k$, and τ yields $\rho = \rho_1 \bullet \ldots \bullet \rho_k$ in σ_0.*

A possibly infinite sequence of states λ is a run *of* BSM *$\mathcal{A}$ iff:*

1. *There is a sequence of positions $k_0 = 0, k_1, k_2, \ldots$ such that, for every $i = 0, 1, 2, \ldots$, we have that $\lambda[k_i..k_{i+1}] \in \mathcal{T}(A)$, and*
2. Weak fairness *(cf. [13]): if an update is enabled infinitely often, then it will be infinitely often selected for execution.*[1]

The semantics of an agent system characterized by BSM *$\mathcal{A} = (\overline{\mathcal{A}}, \tau)$ is the set of all runs of $\mathcal{A}$, denoted $\mathcal{T}(\overline{\mathcal{A}}, \tau^*)$.*

3. LOGICS FOR BSM

To enable reasoning about executions of *BSM* we need a logic tightly bound to the same semantic model as that of the *BSM* framework. This section presents the standard *Linear Time Temporal Logic* (*LTL*), and subsequently introduces *Dynamic CTL** (*DCTL**), an extension of the branching time temporal logic *CTL** [9] with features of Harel's *Dynamic Logic* [10]. While *LTL* provides a tool for relating mental state transformers to formulae of temporal logic, *DCTL** allows for expressing and reasoning about properties of executions of agent programs.

3.1 LTL

A mental state transformer, or a *BSM* program, specifies a set of traces in the corresponding behavioural frame

[1]This condition rules out traces where, for some nondeterministic choice $\tau_1|\tau_2$ in program τ, always the same option is selected during the iterated execution of τ – and the other option is neglected.

(i.e., a labeled transition system over the space of mental states of the corresponding behavioural template). Mental states of a *BSM* are composed of theories in KR languages of the corresponding KR modules. What is important, we do not assume any relationship between these languages and mathematical logic (that is why we call the languages *logic-agnostic*). For example, the interface of one module can be based on Java, another on Prolog, while queries and mst's of yet another module can be given in the assembly language. This section shows how a relationship between such KR modules and logical formulae can be obtained by means of *LTL* annotations. But first, we will extend behavioural frames with an interpretation of basic logical statements.

Definition 7 (behavioural model). *Let $LTS(\mathcal{S},\mathcal{R})$ be the behavioural frame of a behavioural template $\overline{\mathcal{A}} = (\mathcal{M}_1, \dots, \mathcal{M}_n)$. The* behavioural model *of $\overline{\mathcal{A}}$ is defined as $\overline{LTS}(\overline{\mathcal{A}}) = (\mathcal{S},\mathcal{R},\Pi,\pi)$, where $LTS(\overline{\mathcal{A}}) = (\mathcal{S},\mathcal{R})$ is the behavioral frame of $\overline{\mathcal{A}}$, $\Pi = \{\mathsf{p}_\phi \mid \phi \in \mathcal{Q}(\mathcal{A})\}$ is the set of atomic propositions, and $\pi : \Pi \rightarrow 2^{\mathcal{S}}$ defines their valuations so that they correspond to primitive queries: $\pi(\mathsf{p}_\phi) = \{\sigma \in \mathcal{S} \mid \sigma \models \phi\}$.*

Behavioural model of a BSM is defined as the behavioural model of the underlying behavioural template: $\overline{LTS}(\overline{\mathcal{A}},\mathcal{P}) = \overline{LTS}(\overline{\mathcal{A}})$.

LTL [17] enables reasoning about properties of execution traces by means of temporal operators $\bigcirc$ (in the next moment) and $\mathcal{U}$ (until). Additional operators $\Diamond$ (sometime in the future) and $\Box$ (always in the future) can be defined as $\Diamond\varphi \equiv \top\,\mathcal{U}\,\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. We will use a version of *LTL* that includes the "chop" operator $\mathcal{C}$ [19] because of the nature of sequential composition of mental state transformers: when constructing an aggregate annotation for $\tau_1 \circ \tau_2$, we need a way of enforcing that the part that refers to τ_1 is fulfilled *before* the execution of τ_2 begins. Formally, our version of *LTL* is given by the following grammar:

$$\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi\,\mathcal{C}\,\varphi.$$

Other Boolean operators (disjunction $\vee$, material implication $\rightarrow$, etc.) are defined in the usual way. The semantics is defined through the clauses below (where $\mathcal{A}$ is a *BSM* and λ is a trace in $\mathcal{T}(\mathcal{A})$):

$\mathcal{A},\lambda \models \mathsf{p}$ iff $\lambda[0] \in \pi(\mathsf{p})$ in $\overline{LTS}(\mathcal{A})$;

$\mathcal{A},\lambda \models \neg\varphi$ iff $\mathcal{A},\lambda \not\models \varphi$;

$\mathcal{A},\lambda \models \varphi_1 \wedge \varphi_2$ iff $\mathcal{A},\lambda \models \varphi_1$ and $\mathcal{A},\lambda \models \varphi_2$;

$\mathcal{A},\lambda \models \bigcirc\varphi$ iff $\mathcal{A},\lambda[1..\infty] \models \varphi$;

$\mathcal{A},\lambda \models \varphi_1\,\mathcal{U}\,\varphi_2$ iff there exists $i \geq 0$, such that $\mathcal{A},\lambda[i..\infty] \models \varphi_2$, and $\mathcal{A},\lambda[j..\infty] \models \varphi_1$ for every $0 \leq j < i$;

$\mathcal{A},\lambda \models \varphi_1\,\mathcal{C}\,\varphi_2$ iff there exists $i \geq 0$, such that $\mathcal{A},\lambda[0..i] \models \varphi_1$ and $\mathcal{A},\lambda[i..\infty] \models \varphi_2$.

LTL formula φ is *valid in* $\mathcal{A}$ (written $\mathcal{A} \models \varphi$) iff φ holds on every trace $\lambda \in \mathcal{T}(\mathcal{A})$.

3.2 DCTL*

Since each annotation is assigned to a particular mst, there is no point in referring to the mst in the object language. However, we will need a richer logic for *reasoning about programs* and their relationships: namely one which allows to address a particular program explicitly. To this end, we propose an extension of the branching-time logic *CTL** [9] with explicit quantification over program executions. In the extension, $[\tau]$ stands for "*for all executions of* τ"; "*there is an execution of* τ" can be defined as $\langle\tau\rangle\varphi \equiv \neg[\tau]\neg\varphi$. As the agenda of the logic resembles that of "dynamic *LTL*" from [11], we will call our logic "*Dynamic CTL**", *DCTL** in short. The syntax of *DCTL** is defined as an extension of *LTL* by the following grammar:

$$\begin{aligned}\theta &::= \mathsf{p} \mid \neg\theta \mid \theta \wedge \theta \mid [\tau]\varphi\\ \varphi &::= \theta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi\,\mathcal{C}\,\varphi\end{aligned}$$

where τ is a program or an iterated program. The semantics extends that of *LTL* by the clauses below:

$\mathcal{A},\lambda \models \theta$ iff $\mathcal{A},\lambda[0] \models \theta$;

$\mathcal{A},\sigma \models \mathsf{p}$ iff $\sigma \in \pi(\mathsf{p})$;

$\mathcal{A},\sigma \models \neg\theta$ iff $\mathcal{A},\sigma \not\models \theta$;

$\mathcal{A},\sigma \models \theta_1 \wedge \theta_2$ iff $\mathcal{A},\sigma \models \theta_1$ and $\mathcal{A},\sigma \models \theta_2$;

$(\overline{\mathcal{A}},\mathcal{P}),\sigma \models [\tau]\varphi$ iff for every $\lambda \in \mathcal{T}(\overline{\mathcal{A}},\tau)$, s.t. $\lambda[0] = \sigma$, we have that $(\overline{\mathcal{A}},\tau),\lambda \models \varphi$.

*DCTL** formula θ is *valid in* $\mathcal{A}$ (written $\mathcal{A} \models \theta$) iff $\mathcal{A},\sigma \models \theta$ for every state σ of $\mathcal{A}$.

The following proposition shows the relationship between *LTL* and *DCTL** (the proof is straightforward):

Proposition 1. *For every BSM $\mathcal{A} = (\mathcal{M}_1,\dots,\mathcal{M}_n,\tau)$ and* LTL *formula φ, we have: $\mathcal{A} \models_{\text{LTL}} \varphi$ iff $\mathcal{A} \models_{\text{DCTL*}} [\tau]\varphi$.*

When reasoning about code patterns in Section 5, we will use the following notion of *semantic consequence*.

Definition 8 (semantic consequence). *Formula ψ is a semantic consequence of φ (written: $\varphi \Rightarrow \psi$) iff, for every* BSM *$\mathcal{A}$, $\mathcal{A} \models \varphi$ implies $\mathcal{A} \models \psi$.*

Usually, we will use the notion to state that $[\tau]\varphi \Rightarrow [\tau^*]\psi$, that is, if formula φ correctly describes possible executions of program τ, then ψ holds for all possible iterated executions of the program.

4. TEMPORAL ANNOTATIONS FOR BSM

The *BSM* framework allows us to encode agent programs in terms of compound mst's interpreted in a behavioural model over a behavioural template (a set of KR modules). Our idea is to use *LTL* and *DCTL** for reasoning about execution traces in such models. To bridge the gap between the mental states of a *BSM* and interpreted states of behavioural models, we introduce *Annotated Behavioural State Machines*: *BSM* enriched with *LTL* annotations of primitive queries and updates occurring in the corresponding agent program. The basic methodological assumption behind our proposal is as follows: a KR module supplies a set of primitive queries and updates, i.e., a repository of basic tests and procedures for agent programming. Annotations provide an interpretation of these from logic-agnostic programming KR languages into a single language for reasoning about properties of agent programs. The interpretation of compound programs can be derived from the basic annotations by using a predefined scheme.

Definition 9 (Annotated BSM). Annotated BSM *is a tuple $\mathcal{A}^{\mathfrak{A}} = (\mathcal{M}_1,\dots,\mathcal{M}_n,\mathcal{P},\mathfrak{A})$, where $\mathcal{A} = (\mathcal{M}_1,\dots,\mathcal{M}_n, \mathcal{P})$ is a* BSM *and $\mathfrak{A} : (\mathcal{U}(\mathcal{A}) \cup \mathcal{Q}(\mathcal{A})) \rightarrow LTL$ is an* annotation function *assigning an* LTL *annotation to each primitive query and update occurring in $\mathcal{A}$.*

Technically, it suffices to annotate only the queries and mst's that occur in the program $\mathcal{P}$ of the *BSM* that we implement or plan to reason about. Annotations of primitive queries and mst's are provided by agent developer(s), according to their insight and expertise.

Given a complex mst τ, its annotation is determined by combining the annotations of its subprograms w.r.t. the outermost operator in τ.

DEFINITION 10 (AGGREGATION OF ANNOTATIONS). *Let $\mathcal{A}^{\mathfrak{A}}$ be an annotated* BSM. *We extend the function $\mathfrak{A}$ to provide also* LTL *annotations for compound queries and mst's recursively as follows:*

- *let ϕ, ϕ' be queries, then $\mathfrak{A}(\neg\phi) = \neg\mathfrak{A}(\phi)$, $\mathfrak{A}(\phi \wedge \phi') = \mathfrak{A}(\phi) \wedge \mathfrak{A}(\phi')$, and $\mathfrak{A}(\phi \vee \phi') = \mathfrak{A}(\phi) \vee \mathfrak{A}(\phi')$,*
- *let ϕ be a query and τ be an mst, then $\mathfrak{A}(\phi \longrightarrow \tau) = \mathfrak{A}(\phi) \rightarrow \mathfrak{A}(\tau)$,*
- *let τ_1, τ_2 be mst's, then $\mathfrak{A}(\tau_1|\tau_2) = \mathfrak{A}(\tau_1) \vee \mathfrak{A}(\tau)$ and $\mathfrak{A}(\tau_1 \circ \tau_2) = \mathfrak{A}(\tau_1)\, \mathcal{C}\, \mathfrak{A}(\tau)$.*

Annotations are not intended to be just arbitrary logical formulae; they should capture the relevant aspects of the queries and programs that they are assigned to. To this end, we require that the annotations in $\mathcal{A}^{\mathfrak{A}}$ are *sound* in the following sense:

DEFINITION 11 (SOUNDNESS OF ANNOTATIONS). *Let $\mathcal{A}^{\mathfrak{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \mathfrak{A})$ be an annotated* BSM.

1. *$\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ w.r.t. a query φ iff $\mathfrak{A}(\varphi)$ holds in exactly the same mental states of $\overline{LTS}(\mathcal{A})$ as φ;*
2. *$\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ w.r.t. a program τ iff $\mathfrak{A}(\tau)$ holds for all traces from $\mathcal{T}(\mathcal{M}_1, \ldots, \mathcal{M}_n, \tau)$.*

$\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ iff it is sound w.r.t. program $\mathcal{P}$ in $\mathcal{A}^{\mathfrak{A}}$. Note that $\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$ iff $\mathcal{A} \models [\mathcal{P}]\mathfrak{A}(\mathcal{P})$.

PROPOSITION 2. *If $\mathfrak{A}$ is sound for every primitive query and update in $\mathcal{A}^{\mathfrak{A}}$, then $\mathfrak{A}$ is sound in $\mathcal{A}^{\mathfrak{A}}$.*

PROOF. For every mst τ (resp. query φ), the soundness of $\mathfrak{A}$ w.r.t. τ (resp. φ) follows by induction on the structure of τ (resp. φ): it is sufficient to show that the aggregation rules in Definition 10 preserve soundness. $\square$

Provided an *LTL* specification and an annotated *Behavioural State Machine*, we are usually interested whether the runs generated by the machine satisfy the specification.

DEFINITION 12 (BSM VERIFICATION). *Let $\mathcal{A}^{\mathfrak{A}} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \mathfrak{A})$ be an annotated* BSM *and $\varphi \in LTL$ be a specification. We say that the iterated execution of $\mathcal{A}^{\mathfrak{A}}$ satisfies the specification φ iff $\mathcal{A} \models [\mathcal{P}^*]\varphi$.*

The following proposition turns out to be helpful in verification of *BSM* consisting of a non-deterministic choice of mst's:

PROPOSITION 3. *Let $\overline{\mathcal{A}}$ be a behavioural template, $\mathfrak{A}$ be a sound annotation function w.r.t. $\overline{\mathcal{A}}$ and τ_1, τ_2 be mst's. Then, $[(\tau_1|\tau_2)^*]\Box\big(\Diamond(\mathfrak{A}(\tau_1)\, \mathcal{C}\, \top) \wedge \Diamond(\mathfrak{A}(\tau_2)\, \mathcal{C}\, \top)\big)$.*

PROOF. Follows immediately from the weak fairness condition. $\square$

The formula in Proposition 3 is a DCTL* correspondent of the weak fairness condition. It also describes how characteristics of finite programs "propagate" to their iterations. This is why it includes the "chop" operators after the annotations of $\mathfrak{A}(\tau_1)$ and $\mathfrak{A}(\tau_2)$: we want the annotations of the component mst's to be evaluated against *finite* chunks of the iterated executions, while the executions themselves can be as well infinite.

5. CODE PATTERNS

The relationship between the programming framework of *Behavioural State Machines* and *DCTL** allows us to finally consider several code patterns useful in agent-oriented programming. They allow encoding agent's functionality in terms of a web of inter-dependencies between agent's beliefs, goals and behaviours without being bound to a specific KR language of the underlying KR modules. The main result of this section is introduction of code templates formalizing the notions of *achievement* and *maintenance goals*. A running example accompanies the introduced code templates. We do not provide detailed proofs of properties of the introduced code patterns; however, their validity can be easily derived from Proposition 3.

5.1 Agent System Architecture

We consider embodied BDI-inspired agent systems. Since the *BSM* framework does not impose constraints on the number of KR modules of an agent, we assume an architecture including the following: *belief base* $\mathcal{B}$, *goal base* $\mathcal{G}$ and an *interface to an environment* $\mathcal{E}$ in which the agent acts. Additionally, we assume the basic interfaces for the belief and goal bases: query operators $\models_{\mathcal{B}}$, $\models_{\mathcal{G}}$ and update operators $\oplus_{\mathcal{B}}$, $\oplus_{\mathcal{G}}$, $\ominus_{\mathcal{B}}$, $\ominus_{\mathcal{G}}$ for a KR language formula assertion/retraction respectively. We also assume the following code annotation function $\mathfrak{A}$: $\mathfrak{A}(\oplus_i\varphi) \equiv \bigcirc\varphi$, $\mathfrak{A}(\ominus_i\varphi) \equiv \bigcirc\neg\varphi$ and $\mathfrak{A}(\models_i\varphi) \equiv \varphi$ for $i \in \{\mathcal{B}, \mathcal{G}\}$, where $\varphi \in \mathcal{L}_i$. We emphasize that the annotations of basic programs refer only to KR sentences from $\mathcal{L}_i$, and not to structural properties of execution traces (like finiteness/termination of a trace). This allows us to omit the "chop" operator in the following examples, and renders our exposition easier to read. Soundness of query and update annotations follows immediately from the above definition of function $\mathfrak{A}$.

To improve readability, we use the mixed mathematical notation of *Jazzyk* [14], a programming language implementation of the *BSM* framework.[2] In *Jazzyk*, `when ... then ...` encodes a conditional mst, while `;` and `,` stand for the non-deterministic choice ($|$) and sequence ($\circ$) operators respectively. Mst's can be grouped into blocks enclosed in curly braces $\{\ldots\}$. Additionally, macros are defined by the keyword '`define`' followed by a macro identifier, an ordered list of named arguments and the macro body ended by the keyword '`end`'.

EXAMPLE 1 (RUNNING EXAMPLE). *Consider a robot acting in a physical environment.*[3] *It consists of a KR module $\mathcal{E}$ interfacing to its physical body, a belief base $\mathcal{B}$ and a goal base $\mathcal{G}$, for simplicity both implemented in a* Prolog-*like programming language. The robot's main task in the environment is to find and pick up an item located somewhere in the*

[2]See also `http://jazzyk.sourceforge.net/`.
[3]Example adapted from the *Jazzbot* case study [15].

environment, while at the same time appropriately reacting to threats occurring in the environment, such as for example an unfriendly agent in its vicinity.

Capabilities. An agent acts in its environment by executing primitive updates of the KR module representing the interface to the environment. Its capabilities are therefore determined by the range of well formed formulae in the corresponding KR language, together with the set of update operators of the module. *Capabilities* are a set of mst's constructed from this universe, primitive or compound, that encapsulate standalone, meaningful and reusable behaviours of the agent. Moreover, we assume that the meaning of these capabilities is well described by program annotations capturing their *intended* effects.

The problem of designing a *BSM* agent performing a specified range of behaviours, at least partially captured by some formal specification φ, can be seen as the problem of managing activation, deactivation and interleaving of the capabilities. In each step of its execution, the agent performs a *selection of a reactive behaviour* to execute. In the remainder of this section, we argue that the notion of *goal* provides a technical basis for management and programming of this selection.

Example 2 (running example cont.). *Our robot must be able to manage interactions of two concurring behaviours: one for searching and picking up an item called 'item42', and the second for handling a potential interruption in the form of a threat. Two compound mst's encoded as macros* FIND *and* RUN_AWAY *provide the corresponding capabilities of the bot. While we do not discuss their detailed implementations, we assume the following about their annotations:*

$$[\mathsf{FIND}]\mathfrak{A}(\mathsf{FIND}) \Rightarrow [\mathsf{FIND}^*]\Diamond \mathit{holds}(\mathit{item42})$$
$$[\mathsf{RUN\_AWAY}]\mathfrak{A}(\mathsf{RUN\_AWAY}) \Rightarrow [\mathsf{RUN\_AWAY}^*]\Diamond \mathit{safe}$$

That is, FIND*, when iterated, eventually brings about a state in which the robot found and picked up the item42; similarly, repeated* RUN_AWAY *eventually brings about a state in which the agent is safe. Additionally, we assume that the capabilities already implement parts of the agent's perception relevant to their specified functionality.*

5.2 Goal-oriented behaviours

To allow for an explicit management (activating/deactivating) of agent's capabilities, each capability mst should be triggered only when appropriate. The behaviour activation then becomes purposeful: a behaviour is triggered because an agent has a goal and it is supposed to take the agent closer to the goal achievement. The explicit representation of a goal is a formula derived from the agent's goal base. The code pattern for an agent's capability τ triggered by derivation of a goal formula $\varphi_{\mathbf{G}}$ from agent's goal base looks as follows:

```
define TRIGGER($\varphi_{\mathbf{G}}$, $\tau$)
  when $\models_{\mathcal{G}} \varphi_{\mathbf{G}}$ then $\tau$
end
```

The code pattern allows for conditional activating and deactivating of the capability τ, depending on the derivability of the condition $\varphi_{\mathbf{G}}$ w.r.t. the agent's goal base. In the following, we assume that execution of τ does not change validity of the associated goal formula $\varphi_{\mathbf{G}}$, i.e., the following independence condition holds:

$$[\tau]\mathfrak{A}(\tau) \Rightarrow (\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}}) \rightarrow [\tau^*]\Box\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}})).$$

It can be shown that when the agent has a goal $\varphi_{\mathbf{G}}$, then iterated execution of TRIGGER always eventually leads to satisfaction of the annotation of τ. Formally, if τ is a capability and $\varphi_{\mathbf{G}}$ a goal formula, then we have:

$$[\tau]\mathfrak{A}(\tau) \Rightarrow (\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}}) \rightarrow [\mathsf{TRIGGER}(\varphi_{\mathbf{G}},\tau)^*]\Diamond\mathfrak{A}(\tau)).$$

Example 3 (running example cont.). *The agent should search for the 'item42' only when it has a goal to get it. Similarly, it runs away, only when it's goal is to maintain its own safety. The following instances of* TRIGGER *reformulate the macros* FIND *and* RUN_AWAY *as goal oriented behaviours:*

$$\mathsf{TRIGGER}(\mathit{has}(\mathit{item42}),\ \mathsf{FIND})$$
$$\mathsf{TRIGGER}(\mathit{keep\_safe},\ \mathsf{RUN\_AWAY})$$

Goal commitment strategies. Explicit goal representation allows for conditional activation or deactivation of behaviours. However, to directly manage when such a goal formula should be derivable, a programmer must choose an explicit algorithm: *a goal commitment strategy.*

A goal commitment strategy explicitly encodes the *reasons* for a goal adoption and dropping of it. When an agent believes it can adopt a goal, it should also add the explicit representation of it (a goal formula) to its goal base. Similarly when it believes the goal can be dropped, it should also remove the corresponding goal formula from its goal base. The following two code patterns ADOPT and DROP provide a toolbox for encoding an appropriate commitment strategy for a given goal formula.

```
define ADOPT($\varphi_{\mathbf{G}}$, $\psi_{\oplus}$)
  when $\models_{\mathcal{B}}\psi_{\oplus}$ and not $\models_{\mathcal{G}}\varphi_{\mathbf{G}}$ then $\oplus_{\mathcal{G}}\varphi_{\mathbf{G}}$
end
define DROP($\varphi_{\mathbf{G}}$, $\psi_{\ominus}$)
  when $\models_{\mathcal{B}}\psi_{\ominus}$ and $\models_{\mathcal{G}}\varphi_{\mathbf{G}}$ then $\ominus_{\mathcal{G}}\varphi_{\mathbf{G}}$
end
```

Provided $\varphi_{\mathbf{G}}$ is a goal formula and $\psi_{\oplus}$, $\psi_{\ominus}$ are adopt/drop conditions on agent's beliefs, we can formulate the following property of the ADOPT and DROP macros (that is, the following formula is valid in every annotated *BSM*):

$$\mathfrak{A}(\models_{\mathcal{B}}\psi_{\oplus}) \rightarrow [\mathsf{ADOPT}(\varphi_{\mathbf{G}},\psi_{\oplus})^*]\Diamond\mathfrak{A}(\models_{\mathcal{G}}\psi_{\mathbf{G}}) \quad \wedge$$
$$\mathfrak{A}(\models_{\mathcal{B}}\psi_{\ominus}) \rightarrow [\mathsf{DROP}(\varphi_{\mathbf{G}},\psi_{\ominus})^*]\Diamond\neg\mathfrak{A}(\models_{\mathcal{G}}\psi_{\mathbf{G}}).$$

Goal oriented behaviour, together with an associated commitment strategy forms a particular goal. The goal formula thus becomes an explicit representation (a placeholder) for the goal. In the following, we discuss two code patterns for particularly useful goal types.

Achievement goals. The notion of an *achievement goal* is one of the central constructs of agent-oriented programming. Provided an agent does not believe that a goal satisfaction condition $\varphi_{\mathbf{B}}$ is true in a given point of time, an achievement goal $\varphi_{\mathbf{G}}$ specifies that the agent desires to eventually make it true. After having satisfied the goal, it can be dropped. Moreover, if the agent believes the goal is unachievable ($\varphi_{\ominus}$), it can retract its commitment to it. ACHIEVE_GOAL code template formalizes this type of goal:

```
define ACHIEVE_GOAL($\varphi_{\mathbf{G}}$, $\varphi_{\mathbf{B}}$, $\psi_{\oplus}$, $\psi_{\ominus}$, $\tau$)
  TRIGGER($\varphi_{\mathbf{G}}$, $\tau$) ;
```

```
    ADOPT($\varphi_{\mathbf{G}}$, $\psi_\oplus$) ;
    DROP($\varphi_{\mathbf{G}}$, $\varphi_{\mathbf{B}}$) ;
    DROP($\varphi_{\mathbf{G}}$, $\psi_\ominus$)
  end
```

The code template for an achievement goal combines a goal oriented behaviour with a corresponding commitment strategy, so that in a single execution step, either the agent handles the goal commitment, or it enables the behaviour for its achievement. Provided $\mathsf{TRIGGER}(\varphi_{\mathbf{G}}, \tau)$ is a goal oriented behaviour, $\psi_\oplus$, $\psi_\ominus$ are goal adoption and drop conditions respectively, and assuming that during iterated execution of τ an agent will eventually believe that the goal is satisfied, ACHIEVE_GOAL makes the agent stick to the goal until it is satisfied:

$$([\tau]\mathfrak{A}(\tau) \wedge [\tau^*]\Diamond\varphi_{\mathbf{B}}) \Rightarrow [\mathsf{ACHIEVE\_GOAL}(\varphi_{\mathbf{G}}, \varphi_{\mathbf{B}}, \psi_\oplus, \psi_\ominus, \tau)^*] \mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}})\,\mathcal{U}\,\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}} \vee \models_{\mathcal{B}}\varphi_\ominus).$$

The provided implementation of the achievement goal implements a commitment strategy similar to that of the specification of *persistent relativized goal* P-R-GOAL, defined by Cohen and Levesque in [4]. P-R-GOAL is to be dropped also when the agent believes the goal cannot be achieved. In our case, however, this constraint must be implemented by a programmer and formulated as an explicit goal drop condition $\varphi_\ominus$. In the basic setup of an agent architecture discussed here, we do not assume introspective capabilities of an agent.

Example 4 (running example cont.). *The robot should start searching for the 'item42', only when it believes it needs it. Otherwise, when either it does not need it anymore, or already holds it, or the item does not exist anymore, it should drop the goal and thus deactivate the searching behaviour. The following code pattern reformulates the goal oriented behaviour for the FIND capability as an instance of the achievement goal:*

```
  ACHIEVE_GOAL(
    $has(item42)$,
    $holds(item42)$,
    $needs(item42)$,
    $\neg needs(item42) \vee \neg exists(item42)$,
    FIND)
```

Maintenance goals. A particularly useful commitment strategy for a goal is *persistent maintenance* of a certain condition of the agent's beliefs. Provided that a belief condition $\varphi_{\mathbf{B}}$ is an intended consequence of a capability τ (i.e. $[\tau]\mathfrak{A}(\tau) \Rightarrow [\tau^*]\Diamond\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}})$), its violation should trigger the behaviour τ supposed to maintain its validity. Moreover, in our implementation of the maintenance goal it should never be dropped, i.e., in the case it cannot be derived from the goal base, it should be re-instantiated. The following code pattern implements the notion of a maintenance goal:

```
  define MAINTAIN_GOAL($\varphi_{\mathbf{G}}$, $\varphi_{\mathbf{B}}$, $\tau$)
    when not $\models_{\mathcal{B}}\varphi_{\mathbf{B}}$ then TRIGGER($\varphi_{\mathbf{G}}$,$\tau$) ;
    ADOPT($\varphi_{\mathbf{G}}$, $\top$)
  end
```

Let $\mathsf{TRIGGER}(\varphi_{\mathbf{G}}, \tau)$ be a goal oriented behaviour and $\varphi_{\mathbf{B}}$ be a maintenance condition. The code pattern MAINTAIN_GOAL satisfies the following property:

$$([\tau]\mathfrak{A}(\tau) \wedge [\tau^*]\Diamond\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}})) \Rightarrow (\mathfrak{A}(\models_{\mathcal{G}}\varphi_{\mathbf{G}}) \rightarrow [\mathsf{MAINTAIN\_GOAL}(\varphi_{\mathbf{G}}, \varphi_{\mathbf{B}}\tau)^*]\Box (\neg\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}}) \rightarrow \Diamond\mathfrak{A}(\models_{\mathcal{B}}\varphi_{\mathbf{B}}))).$$

Example 5 (running example cont.). *The robot desires to maintain its safety. If it feels threatened, it attempts to restore the feeling of safety by running away from the dangerous situation. The following mst formulates the corresponding maintenance goal:*

```
  MAINTAIN_GOAL($keep\_safe$, $safe$, RUN_AWAY)
```

We finish this section by completing the running example of the robot with a sketch of a final agent program implementing its specified behaviour.

Example 6 (running example finish). *The original intended behaviour of the agent system was to promptly react to interruptions it faces (potential threats), while still maintaining the contexts of the other active goals (searching for an item). The following program combines the previously developed code chunks:*

```
  PERCEIVE ,
  { MAINTAIN_GOAL($keep\_safe$, $safe$, RUN_AWAY) ;
    ACHIEVE_GOAL(
      $has(item42)$,
      $holds(item42)$,
      $needs(item42)$,
      $\neg needs(item42) \vee \neg exists(item42)$,
      FIND)
  }
```

To realistically model a real robot, we assume PERCEIVE, a macro implementing querying the environment and updating the agent's belief base $\mathcal{B}$ accordingly. The robot first extracts perceptory input from its interface to the environment and subsequently selects an appropriate reactive behaviour to be performed.

The *DCTL** characterizations of macros ACHIEVE_GOAL and MAINTAIN_GOAL, presented earlier in this section, ensure that the program in Example 6 indeed implements the right behaviour.

6. DISCUSSION & CONCLUSIONS

Source code modularity and reusability are one of the principal concerns of pragmatic software engineering. To support reusability, especially in teams of programmers, the code must provide clear interfaces and crisp semantic characterization of its functionality. The maxim of such semantic characterization is a non-ambiguous formal language. Therefore a tight relationship between a programming framework and a logic for reasoning about programs created in it is vital for a formal study of engineering non-trivial agent systems.

In this paper we introduce *DCTL**, a temporal branching time logic with features of propositional dynamic logic, as a formal vehicle for reasoning about *Behavioural State Machines.* Logic agnostic *BSM* programs are linked to the formal logic by means of annotations provided by the code designer. Capturing and extracting semantic characterizations of *BSM* mental state transformers allows us to describe two orthogonal aspects of code modularity. First, given a KR module interfacing with an environment, programmers can provide and subsequently annotate compound behaviours, *capabilities*, that an agent can perform. These can be reused in various agent systems which employ the same type of a physical body of an agent, i.e., the KR module interfacing with the agent's environment. Second, we demonstrate development of *application domain-independent code* *patterns* for implementing a modular, BDI-inspired agent system architecture. The patterns we provide allow for encoding agent's internal functionality in terms of a web of interdependent mental attitudes and capabilities. The central piece of the architectural sketch is the notion of a goal formula stored in an agent's goal base. Moreover, the provided patterns are completely decoupled from actual KR technologies that the programmer has chosen for implementing agent's individual mental categories, like beliefs or goals.

Mainstream programming languages for BDI-inspired cognitive agents with formal semantics, such as *AgentSpeak(L)/ Jason* [3], *3APL* [5], and *GOAL* [7], choose a set of agent-oriented concepts and provide their particular implementation. In consequence, the concrete choices of a feature set and its implementation constrain extensibility of the language itself: with each newly introduced feature, the language interpreter has to be modified accordingly. The presented approach provides an *alternative paradigm* for design of agent-oriented programming languages. Instead of a choice of features provided by a language designer, we present *a generic programming language for reactive systems integrating heterogeneous KR technologies* that enables extensions by *code patterns* implementing user defined agent-oriented concepts.

Additionally, providing a temporal logic tightly associated with the programming framework allows us to tackle the two sides of the software engineering process. First, given a formal specification of an agent system in terms of goals and temporal properties, the creative process of refining and decomposing the specification to sub-problems (divide-and-conquer) and designing an implemented system satisfying the specification can be supported by a library of semantically well characterized agent-oriented code patterns. On the other hand, provided an implementation of an agent system, we can relate it to the original system specification by extracting its semantic characterization.

Except for some recent efforts for the language *Gwendolen* by Dennis et al. [8] and for *GOAL* by de Boer et al. [7], a tight relationship of an agent programming language with a formal logic enabling verification of programs remains largely unexplored. Both these languages are yet to be applied in a non-trivial application domain. Moreover, their close relationship with a logic-based knowledge representation presents a serious limitation for implementation of domain specific cognitive agents requiring also non-logic based forms of reasoning, e.g., mobile robots which need to reason about topology of their environment. Finally, Bordini et al. in [3] made an attempt to propose several design patterns in the programming language *Jason*. However, they introduce code patterns only informally as an auxiliary tool, potentially useful in recurring pattern in agent programs development, rather than a basis for further extending of the language.

7. REFERENCES

[1] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge J. Gomez-Sanz, João Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30:33–44, 2006.

[2] 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, 2005.

[3] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. Wiley-Blackwell, 2007.

[4] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[5] 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* [2], 2005.

[6] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. A grounded specification language for agent programs. In *Proceedings of AAMAS'07*, pages 578–585, 2007.

[7] 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.

[8] Louise A. Dennis and Berndt Farwer. Gwendolen: A BDI language for verifiable agents. *Logic and the Simulation of Interaction and Reasoning*, 2008. AISB'08 Workshop.

[9] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. 1990.

[10] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.

[11] Jesper G. Henriksen and P. S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied Logic*, 96(1-3):187–207, 1999.

[12] Randolph M. Jones and Robert E. Wray III. Comparative analysis of frameworks for knowledge-intensive intelligent agents. *AI Magazine*, 27(2):45–56, 2006.

[13] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[14] Peter Novák. Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations. In *Proceedings of ProMAS'08*, pages 72–87. LNAI 5442, Springer, 2008.

[15] Peter Novák and Michael Köster. Designing goal-oriented reactive behaviours. In *Proceedings of CogRob'08*, pages 25–31, 2008.

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

[17] A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS*, pages 46–57, 1977.

[18] Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In *Proceedings of KR'92*, pages 439–449, 1992.

[19] Roni Rosner and Amir Pnueli. A choppy logic. In *Proceedings of LICS'86*, pages 306–313. IEEE Computer Society, 1986.

[20] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.