Code Patterns for Agent-Oriented Programming[1]

Peter Novák[a] Wojciech Jamroga[a b]

[a] *Department of Informatics, Clausthal University of Technology, Germany*
[b] *Individual and Collective Reasoning Group, University of Luxembourg*
peter.novak@in.tu-clausthal.de, wojtek.jamroga@uni.lu

1 Introduction

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. 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, it is desirable to establish a tight relationship of the language with a formal framework for reasoning about programs written in it. BDI-inspired agent programming languages (such as *AgentSpeak(L)/Jason*, *3APL*, *GOAL*, etc.) 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, as well as a fixed implementation of language constructs for agent's beliefs and/or goals, and their mutual relationships.

In this paper, we put forward an alternative approach. 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*) [2] allowing for an application-specific architecture of an agent system employing heterogeneous KR technologies. The hierarchical structure of subprograms allows to define and instantiate macros which implement 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** with features of dynamic logic. Finally, to bridge the gap between the flexible but logic-agnostic programming framework and *DCTL**, we propose *program annotations* in the form of temporal formulae.

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 agent programs, we introduce a logic for their verification and proving their properties.

2 Temporal Annotations for BSM

The *BSM* framework allows us to encode agent programs in terms of compound mental state transformers (mst's). Our idea is to use temporal-dynamic logic 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

[1]The full version of the paper appeared in: Peter Novak and Wojciech Jamroga, Code Patterns for Agent Oriented Programming, *Proceedings of AAMAS'09*, pp. 105–112, 2009.

proposal is as follows: a 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 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.

2.1 The Logics: LTL and DCTL*

LTL 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 use a version of *LTL* that includes the "chop" operator $\mathcal{C}$ because of the nature of sequential composition of mental state transformers. Since each annotation is assigned to a particular mst, there is no point in referring to the mst when we *write* the annotations. However, a richer logic is needed 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** 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 [1], we call our logic "*Dynamic CTL**", *DCTL** in short.

2.2 Annotated BSM's

An *annotated BSM* is a *BSM* enhanced by an *annotation function* $\mathfrak{A}$ assigning an *LTL* annotation to each primitive query and update occurring in it. 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 with respect to the outermost operator in τ.

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.

3 Code Patterns

The relationship between the programming framework of *BSM* and *DCTL** allows us to finally consider several code patterns useful in agent-oriented programming. The logic can be then used to prove that the code patterns indeed implement the concept they are supposed to capture. As an example, consider the following code pattern:

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

that triggers capability τ whenever goal formula $\varphi_{\mathbf{G}}$ can be derived from the agent's goal base. It can be shown that when the agent has a goal $\varphi_{\mathbf{G}}$, then iterated execution of TRIGGER eventually leads to a temporal pattern that displays the characteristics of τ. Formally:

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

For our analysis of more sophisticated code patterns implementing various types of goals, we refer to the full version of the paper.

References

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

[2] Peter Novák. Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations. In *Proceedings of ProMAS*, 2008.