

# Coordinating Agents in OO

Frank S. de Boer<sup>1,2,3</sup>, Cees Pierik<sup>1</sup>,  
Rogier M. van Eijk<sup>1</sup>, and John-Jules Ch. Meyer<sup>1</sup>

<sup>1</sup> Institute of Information and Computing Sciences, Utrecht University,  
The Netherlands

{frankb,cees,rogier,jj}@cs.uu.nl

<sup>2</sup> CWI, Amsterdam, The Netherlands

<sup>3</sup> LIACS, Leiden, The Netherlands

**Abstract.** In this paper we introduce an object-oriented coordination language for multi-agents systems. The beliefs and reasoning capabilities of an agent are specified in terms of a corresponding abstract data type. Agents interact via an extension of the usual object-oriented message passing mechanism. This extension provides the autonomy that is required of agents but which objects in most object-oriented languages do not have. It consists of an explicit answer statement by means of which an agent can specify that it is willing to accept some specified messages. For our coordination language we also present a formal method for proving correctness. The method extends and generalizes existing assertional proof methods for object-oriented languages.

## 1 Introduction

Software agents and multi-agent systems have become a major area of research and development activity in computing science, with a huge potential of applications in many different domains. One may even speak of an emerging new paradigm of *agent-oriented programming* in which the concept of an agent as a software entity displaying autonomous behavior based on a private mental state and communicating and co-operating with other agents in a common environment plays a central role.

However, it is also clear that the very concept of agenthood is a complex one, and it is imperative that programming agents be amenable to precise and formal specification and verification, at least for some critical applications. This is recognized by (potential) applicers of agent technology such as NASA, which organizes specialized workshops on the subject of formal specification and verification of agents [1, 2]. Since agents comprise a new computational concept, tools for verification are not immediately available, and it is not clear *a priori* how these can be obtained. (There *are* special (modal) logics for describing agents but generally it is not clear how these are to be related to concrete agent programs. This is an instance of what is generally called the gap between agent theories and agent implementation [3, 4].)

A possibly viable way to go is to consider the relation of agents with the much more established paradigm of object-oriented programming. Sometimes agents

are compared to objects as being special kinds (classes) of objects where the main distinction is the autonomous aspect of their behavior. Typically, agents cannot be just commanded to execute some action (method), but they should be requested to perform some action on behalf of another agent. An agent that is requested to do such will consider whether it is in line with its own mental state (beliefs, desires, goals, intentions etc.) to do so (and may refuse) [3].

If a relationship with OO programming can be established this also opens up a way to use specification and verification methods and techniques from the realm of OO. In the present paper we investigate this line of research by considering a simple agent programming language (in the style of [5]) that is an ‘agentified’ adaptation of an object-oriented language for which a correctness logic (and associated tools) is known. Of course, special attention has to be given to the agent-oriented features such as agent communication and coordination, which results in a novel Hoare-style correctness logic.

This novel Hoare logic constitutes a promising first step towards realizing C.A.R. Hoare’s Grand Challenge for computer science in the context of agent-oriented programming. In general, information technology industry is still in need for more reliable techniques that guarantee the quality of software. This need is recognized by Hoare as a Grand Challenge in his proposal of the verifying compiler [6].

A verifying compiler resembles a type checker in the sense that it *statically* checks properties of a program. The properties that should be checked have to be stated by the programmer in terms of *assertions* in the code. The first sketch of such a tool is given by Floyd [7]. In recent years, several programming languages (e.g., Eiffel [8]) have been extended to support the inclusion of assertions in the code. These assertions can be used for testing but they can also be used as a basis for a *proof outline* of a program. Hoare logic [9] can be seen as a systematic way of generating the verification conditions which ensure that an annotated program indeed constitutes a proof-outline. A verifying compiler then consists of a front-end tool to a theorem prover which checks the automatically generated verification conditions (see, for example, [10]).

A fundamental idea underlying Hoare logics as introduced by C.A.R. Hoare is that it allows the specification and verification of a program at the same level of abstraction as the programming language itself. This idea has some important consequences for the specification of properties of agent-oriented programs that involve dynamically allocated variables for dynamic referencing of agents. Firstly, this means that, in general, there is no explicit reference in the assertion language to the heap (using object-oriented jargon) which stores the dynamically allocated variables at run-time. Moreover, it is only possible to refer to variables in the heap that do exist. Variables that do not (yet) exist never play a role. In general, these restrictions imply that first-order logic is too weak to express some interesting properties of dynamically allocated heap structures.

The main contribution of this paper is a Hoare logic for reasoning about an object-oriented programming language for the coordination of intelligent agents. The Hoare logic is based on an assertion language which allows the specification

of run-time properties at an abstraction level that coincides with that of the programming language. The coordination language itself supports an abstract data type for the beliefs and reasoning capabilities of an agent. This abstract data type allows the integration of heterogenous agents which are defined by different implementations of their beliefs and reasoning capabilities. Agents can be created dynamically and to ensure autonomy, agents can interact only via method calls, that is, they cannot access directly each other beliefs (which are thus encapsulated). Agents however can communicate their beliefs via the parameter passing mechanism of method calls. The autonomy of agents is additionally supported by means of answer statements which specify the methods an agent is willing to answer.

This paper is organized in a straightforward way. In Sect. 2 we introduce the programming language. In the following section we describe the assertion language that is used to annotate programs. Section 4 contains the Hoare logic. This paper ends with conclusions and some hints for future work.

## 2 The Programming Language

An agent system consists of a dynamically evolving set of independently operating agents that interact with each other through requests. Each agent belongs to an agent class that specifies the requests that agents of the class can be asked to answer. Agents have explicit control over the invocations of its methods: there is a special language construct that is used to specify at which point in its execution an agent is willing to answer which requests (although not from which agent).

Furthermore, an agent has a belief base and a program that governs its behavior. This program is built from traditional programming statements like sequential composition, choice and while-loops and contains atomic operations for handling question invocations, for querying and maintaining the belief base and an operation for integrating new agents into the system. Each agent has a private set of instance variables to keep track of and refer to the agents in the system that it knows (and thus can ask questions to).

Agents maintain their own subjective belief base. We will assume that a belief base is an element of an abstract data type `Belief`. In this paper  $p, q, \dots$  will denote typical values of this type. Beliefs can appear in several ways in the program. For example, an agent can pass beliefs to other agents by sending them as arguments in requests. The receiving agents store the incoming beliefs in the formal parameters of the corresponding methods. A parameter that refers to a formula will have type `Belief`.

An agent can perform three operations on beliefs besides passing beliefs to other agents. It can add a belief to its belief base by means of the statement `assert( $p$ )` and retract it from its beliefs by the command `retract( $p$ )`. Finally, it can test whether a belief  $p$  follows from its belief base by the command `query( $p$ )`. The result of a query is always a boolean value that can be assigned to a (boolean) variable. We do not allow assignments of beliefs to instance variables because

the beliefs of an agent are stored in a separate belief base and we do not want to mix up the belief base of an agent and the values of its internal state.

We will now formally define the expressions, statements, method declarations, classes and programs of the programming language. We start with the expressions. The expressions in the language are built from local variables and instance variables and the keywords `self` and `null` that are typical for the object-oriented paradigm. In the following definition,  $u$  denotes an arbitrary local variable, and  $x$  is an instance variable of an agent. Agents only have direct access to their own belief base and their own instance and local variables.

**Definition 1 (Expressions).**

$$e \in \text{Expr} ::= \text{null} \mid \text{self} \mid u \mid \text{self}.x \mid e_1 = e_2 \mid p$$

The keyword `self` always refers to the agent that is actively executing the method. The keyword `null` denotes the null reference.

We assume a set of agent class names  $\mathcal{A}$  with typical element  $a$ . The set of types  $\mathcal{T}$  in the program is the set  $\mathcal{A} \cup \{\text{bool}, \text{Belief}\}$ . The language is strongly-typed. Only formal parameters of a method can have type `Belief`. We will silently assume that all expressions and statements are well-typed.

An important feature of the proposed language is that each agent has its own internal activity that is executed in parallel with the activities of the other agents. The internal activity is described by a (sequential) statement  $S$ . We allow the following statements.

**Definition 2 (Statements).**

$$\begin{aligned} S \in \text{Stat} ::= & \text{skip} \mid u := e \mid \text{self}.x := e \mid S_1 ; S_2 \mid \text{if } e \text{ then } S_1 \text{ else } S_2 \mid \\ & \mid \text{while } e \text{ do } S \text{ od} \mid u := \text{new}(a) \mid u := e_0.m(e_1, \dots, e_n) \\ & \mid \text{answer}(m_1, \dots, m_n) \mid \text{assert}(e) \mid \text{retract}(e) \mid u := \text{query}(e) \end{aligned}$$

Observe that an agent can only alter the value of its own instance variables (by means of assignments of the form `self.x := e`). Thus each agent has control over its own instance variables. A statement like  $u := \text{new}(a)$  denotes the creation of a new agent, and the storage of a reference to this agent in the local variable  $u$ . Creation of an agent also implies that the internal activity of an agent is activated.

A statement  $u := e_0.m(e_1, \dots, e_n)$  involves a request to agent  $e_0$  to execute method  $m$  with actual parameters  $e_1, \dots, e_n$ . Such a request will only be fulfilled if the receiving agent  $e_0$  indicates by means of a statement `answer( $m_1, \dots, m_n$ )` that it is willing to answer a request by executing a message out of the list of methods  $m_1, \dots, m_n$ . If  $m \in \{m_1, \dots, m_n\}$  then  $m$  might be selected for execution. Execution of the requesting agent is suspended until its request is answered. When executing a statement `answer( $m_1, \dots, m_n$ )`, an agent will be suspended until it receives a request to execute one of the methods  $m_1, \dots, m_n$ . If some requests are already pending at execution of `answer( $m_1, \dots, m_n$ )` then one of these is chosen non-deterministically. The answering agent executes the method and returns a value. After that, both agents resume their own internal activity.

A method declaration specifies the name of the method  $m$ , a sequence of formal parameters  $\bar{u} = u_1, \dots, u_n$ , a body  $S$  and a return value  $e$ .

**Definition 3 (Method declarations).**

$$\text{md} \in \text{Meth} ::= m(u_1, \dots, u_n) \{ S \text{ return } e \}$$

Formal parameters can have type **Belief**. In that case, the actual parameter should be a belief. We do not allow assignments to formal parameters. For technical convenience, we also assume that the body  $S$  of a method contains no answer statements. Thus an agent always answers one request at a time. However, it may delegate (parts of) its task to other agents by sending requests in the body of a method.

An agent is an instance of an agent class  $A$ .

**Definition 4 (Agent classes).**

$$A \in \text{Agent} ::= (a, X, M, S)$$

Each agent class consists of a unique name  $a$ , a set of typed instance variables  $X$ , a set of method declarations  $M$  and a statement  $S$ . The statement  $S$  specifies the internal activity of the agent. The belief base of an agent is initially empty (i.e., equivalent to **true**).

**Definition 5 (Multi-agent programs).** *A multi-agent program  $\pi$  is a tuple  $(S, A_1, \dots, A_k)$  of an initial statement  $S$  and agent classes  $A_1$  to  $A_k$ .*

The initial statement  $S$  starts the execution of the program. It typically creates some agents and then terminates. The keyword **self** is illegal in  $S$  because the command is not linked to a particular agent. For the same reason, it can neither contain answer statements nor operations on the belief base.

In order to formalize the semantics of a program  $\pi$  we introduce for each agent name  $a$  of a class defined in  $\pi$  an infinite set of agent identities  $\text{Id}_a$ . A global state  $\sigma$  of  $\pi$  is a family of *partial* functions

$$\sigma_a \in \text{Id}_a \rightarrow F,$$

where  $F$  denotes the set of assignments of values to the instance variables of class  $a$ ,  $a$  being a name of a class defined by  $\pi$ . We assume an implicit auxiliary variable **belief** which represents the belief base, that is,  $\sigma_a(i)(\text{belief})$  represents the belief base of the agent  $i$ . It should be observed here that the auxiliary variable **belief** does not appear in programs and can be accessed only via the operations of the abstract data type of beliefs. An agent  $i \in \text{Id}_a$  *exists* in  $\sigma$  if  $\sigma_a(i)$  is defined, in which case  $\sigma_a(i)(x)$  gives the value of the instance variable  $x$  of  $i$ .

A local configuration is a pair  $\langle \tau, S \rangle$ , where  $S$  denotes the statement to be executed, and where  $\tau$  is an assignment of values to the local variables of  $\pi$ . Additionally we assume that **self** is a local variable, that is,  $\tau(\text{self})$  denotes the agent that is executing the statement  $S$ .

In order to model the (recursive) execution of answer statements we introduce a stack of method invocations, denoted by  $\gamma$ , which consists of a finite sequence of local configurations  $\langle \tau_1, S_1 \rangle \cdots \langle \tau_n, S_n \rangle$ , where the executing agent is given by  $\tau_i(\mathbf{self})$ , i.e.,  $\tau_i(\mathbf{self}) = \tau_j(\mathbf{self})$ , for  $1 \leq i, j \leq n$ . Moreover, every local configuration  $\langle \tau_{i+1}, S_{i+1} \rangle$ ,  $1 \leq i < n$ , describes the execution of a method which is generated by the execution of a corresponding answer statement  $\mathbf{answer}(\dots, m, \dots); S_i$  in the preceding local configuration.

In the sequel, by  $\gamma \cdot \langle \tau, S \rangle$  we denote the result of appending the local configuration  $\langle \tau, S \rangle$  to the stack  $\gamma$ .

A global configuration of  $\pi$  is a pair  $(\sigma, T)$ , where  $T$  is a set of stacks (one for every existing agent in  $\sigma$ ). The operational semantics then can be defined in a fairly straightforward manner in terms of a transition relation on global configurations. The semantics is parametric in the interpretation of the operations of the abstract data type of beliefs. That is, we assume semantic functions

$$\llbracket \mathbf{assert}(e) \rrbracket(\sigma, \tau)$$

and

$$\llbracket \mathbf{retract}(e) \rrbracket(\sigma, \tau)$$

which specify the global state  $\sigma'$  resulting from a corresponding revision of the belief base  $\sigma(\tau(\mathbf{self}))(\mathbf{belief})$ . The semantic function  $\llbracket \mathbf{query}(e) \rrbracket(\sigma, \tau)$  does not generate any side-effect and encodes whether the belief denoted by  $e$  holds (in the belief base of  $\tau(\mathbf{self})$ ).

We have the following transition for the assert statement (the retract statement is described in a similar manner).

**Assert.** Let  $\mathbf{assert}(e)(\sigma, \tau) = \sigma'$  in the following transition.

$$(\sigma, T \cup \{\gamma \cdot \langle \tau, \mathbf{assert}(e); S \rangle\}) \rightarrow (\sigma', T \cup \{\gamma \cdot \langle \tau, S \rangle\}) .$$

This transition singles out one agent and its corresponding execution stack

$$\gamma \cdot \langle \tau, \mathbf{assert}(e); S \rangle$$

where the top statement starts with an assert statement. The execution of this statement simply consists of updating the global state as specified by the semantics of the assert statement and of a corresponding update of the statement still to be executed. Note that all the other agents in  $T$  remain the same, i.e., we describe concurrency by interleaving.

We have the following main cases describing the semantics of method calls.

**Method Invocation.** In the following transition the stack of the caller is given by

$$\gamma \cdot \langle \tau, u := e_0.m(\bar{e}); S \rangle.$$

The stack of the callee is given by

$$\gamma' \cdot \langle \tau', \mathbf{answer}(\dots, m, \dots); S' \rangle.$$

Furthermore, the agent  $\tau'(\mathbf{self})$  indeed is addressed by  $e_0$  in the call  $e_0.m(\bar{e})$ , i.e., the agent identity  $\tau'(\mathbf{self})$  equals the value of the callee denoted by the expression  $e_0$  evaluated in  $\sigma$  and  $\tau$ .

$$\begin{aligned} & (\sigma, T \cup \{\gamma \cdot \langle \tau, u := e_0.m(\bar{e}); S \rangle, \gamma' \cdot \langle \tau', \mathbf{answer}(\dots, m, \dots); S' \rangle\}) \rightarrow \\ & (\sigma, T \cup \{\gamma \cdot \langle \tau, \mathbf{wait}(u); S \rangle, \gamma' \cdot \langle \tau', S' \rangle \cdot \langle \tau'', S'' \mathbf{return } e \rangle\}) \end{aligned}$$

The stack of the callee is extended with a new local configuration

$$\langle \tau'', S'' \mathbf{return } e \rangle ,$$

where  $S'' \mathbf{return } e$  is the body of method  $m$ . The local state  $\tau''$  assigns to the formal parameters of  $m$  the values of the corresponding actual parameters  $\bar{e}$  (evaluated in  $\sigma$  and  $\tau$ ) and  $\tau''(\mathbf{self}) = \tau'(\mathbf{self})$ . In order to model the return we additionally assume that a local variable  $\mathbf{return}$  is set in  $\tau''$  to the caller  $\tau(\mathbf{self})$ . The waiting of the caller for the return value is modelled by the auxiliary statement  $\mathbf{wait}(u)$ . The semantics of this wait statement is described in the following transition which describes the return from a method call.

**Method Return.** In the following transition the stack of the suspended caller is given by

$$\gamma \cdot \langle \tau, \mathbf{wait}(u); S \rangle .$$

The stack of the terminated callee is given by

$$\gamma' \cdot \langle \tau', S' \rangle \cdot \langle \tau'', \mathbf{return } e \rangle .$$

Furthermore, we require that the callee indeed returns a call invoked by the agent  $\tau(\mathbf{self})$ , i.e.,  $\tau(\mathbf{self}) = \tau''(\mathbf{return})$ . Finally, in the following transition  $v$  is the value of return expression  $e$  (evaluated in  $\sigma$  and  $\tau''$ ) and the result of assigning  $v$  to  $u$  is denoted by  $\tau\{u := v\}$ .

$$\begin{aligned} & (\sigma, T \cup \{\gamma \cdot \langle \tau, \mathbf{wait}(u); S \rangle, \gamma' \cdot \langle \tau', S' \rangle \cdot \langle \tau'', \mathbf{return } e \rangle\}) \rightarrow \\ & (\sigma, T \cup \{\gamma \cdot \langle \tau\{u := v\}, S \rangle, \gamma' \cdot \langle \tau', S' \rangle\}) \end{aligned}$$

Note that after the return the stack of the callee is simply popped.

The transitions for the other statements are standard and therefore omitted.

### 3 The Assertion Language

The proof system that we introduce in the next section is tailored to a specific assertion language. We introduce the syntax and semantics of the assertion language in this section. Between the lines we also explain some of the design decisions behind the language.

An important design decision is to keep its abstraction level as close as possible to the programming language. In other words, we refrain as much as possible from introducing constructs that do not occur in the programming language. This should make it easier for programmers to annotate their programs.

The assertion language is strongly-typed similar to the programming language. By  $\llbracket l \rrbracket$  we denote the type of expression  $l$ . The type of `self` is determined by its context. We will silently assume that all expressions are well-typed.

The set of logical expressions is defined by the following grammar.

**Definition 6 (Logical expressions).**

$$l \in \text{LExpr} ::= \text{null} \mid \text{self} \mid u \mid z \mid l.x \mid l.\phi \mid l_1 = l_2 \\ \mid \text{if } l_1 \text{ then } l_2 \text{ else } l_3 \text{ fi} \mid l[l'] \mid l.\text{length}$$

Here  $\phi$  denotes either a belief  $p$ , a local variable  $u$  or a logical variable  $z$  (of type `Belief`).

The expressions in the programming language are the basis for the set of logical expressions. The variable  $z$  is a logical variable. Logical variables are auxiliary variables that do not occur in the program. Therefore we are always sure that the value of a logical variable can never be changed by a statement. Logical variables are used to express the constancy of certain expressions (for example in a proof rule for message passing below). Logical variables also serve as bound variables for quantifiers.

In a proof outline one can express assumptions about the values of instance variables of an agent (by expressions of the form  $l.x$ ) and about its beliefs (by means of  $l.\phi$ ). These two types of statements have a different meaning. An expression  $l.\phi$  is true if the belief denoted by  $\phi$  follows from agent  $l$ 's belief base, whereas  $l.x$  simply refers to the value of instance variable  $x$  of agent  $l$ .

*Example 1.* Let  $u$  be a formal parameter of a method, and let its type be `Belief`. Then the assertion `self.u = self.x` says that the boolean instance variable  $x$  of agent `self` signals whether belief  $u$  currently follows from its belief base. On the other hand, the assertion `u = self.x` is not well-typed because an instance variable cannot have type `Belief`.

We included logical expressions of the form `if  $l_1$  then  $l_2$  else  $l_3$  fi` to be able to reason about aliases (see Sect. 4.1). Logical variables can additionally have type  $t^*$ , for some  $t$  in the set of types  $\mathcal{T}$  extended with the type of integers. This means that its value is a *finite sequence* of elements from the domain of  $t$ . This addition adds expressive power to the language needed to express certain properties of dynamically changing multi-agent systems. We use  $l[l']$  to select the element at position  $l'$  in the sequence  $l$ . The length of a sequence  $l$  is denoted by  $l.\text{length}$ .

Formulas are built from expressions in the usual way.

**Definition 7 (Formulas).**

$$P, Q \in \text{Ass} ::= l_1 = l_2 \mid \neg P \mid P \wedge Q \mid \exists z : t(P)$$

Note that as atomic formulas we only allow equations. Furthermore we have the usual boolean operations of negation and conjunction (using the standard abbreviations for the other boolean operations). The type  $t$  in the (existential)

quantification  $\exists z : t(P)$  can be of any type. For example,  $t$  can be  $a$  or  $a^*$ , for some class  $a$ . In the first case, the formula means that  $P$  holds for an *existing* agent of class  $a$ . In the latter case,  $P$  should hold for a sequence of such agents. We sometimes omit the type information in  $\exists z : t(P)$  if it is clear from the context. The standard abbreviations like  $\forall z P$  for  $\neg \exists z \neg P$  are valid.

*Example 2 (Agreement).* Many interesting properties can be expressed in the assertion language. We can, for example, say that all agents of an agent class  $a$  share the opinion of a particular agent on the belief denoted by  $\phi$ . This is expressed by the formula  $\forall z : a(\text{self}.\phi = z.\phi)$ .

*Example 3 (Conflicting agents).* We also allow quantification over beliefs in the assertion language. Thus one can, for example, express that two agents  $u$  and  $v$  disagree about everything by the formula  $\forall z : \text{Belief}(u.z = \neg(v.z))$ .

Assertion languages for dynamically allocated variables inevitably contain expressions like  $l.x$  that are normally undefined if, for example,  $l$  is null. However, as an assertion their value should be defined. We solved this problem by giving such expression the same value as null. By only allowing the non-strict equality operator as a basic formula, we nevertheless ensure that formulas are two-valued. For example, the expression  $u.x = \text{true}$  indicates that the value of the (boolean) instance variable  $x$  of object  $u$  is true *and* that  $u$  does not equal null. On the other hand,  $u.x = \text{null}$  signals that either  $u$  equals null or that the value of field  $x$  of object  $u$  equals null. If we omit the equality operator then the value of a boolean logical expression  $l$  is implicitly compared to true.

*Example 4 (Finite sequence).* The following formula

$$\exists z : a^* \forall n (1 \leq n \wedge n < z.\text{length} \rightarrow \forall z' : \text{Belief}(z[n].z' \rightarrow z[n+1].z'))$$

expresses that there exists a finite sequence of agents of class  $a$  which is monotonically increasing with respect to their beliefs.

Formally the semantics of assertions is defined along the lines of [11] with respect to a local state which specifies the values of the local variables and the value of `self`, a global state which specifies the values of the instance variables and the belief bases of the existing agents, and, finally, a logical environment  $\omega$  which specifies the values of the logical variables. The semantics is parametric in the interpretation

$$\llbracket \text{query}(p) \rrbracket(\sigma, \tau)$$

of the query operation of the abstract data type `Belief`, which, as discussed already above, encodes if the belief  $p$  holds in the belief base  $\sigma(\tau(\text{self}))(\text{belief})$ . For example, we define the boolean value

$$\llbracket l.\phi \rrbracket(\omega, \sigma, \tau)$$

resulting from the evaluation of  $l.\phi$ , inductively by

$$\llbracket \text{query}(p') \rrbracket(\sigma, \tau \{\text{self} := \llbracket l \rrbracket(\omega, \sigma, \tau)\}),$$

where  $p' = \llbracket \phi \rrbracket(\omega, \sigma, \tau)$ . This latter semantic function is given by the following three clauses.

- $\llbracket p \rrbracket(\omega, \sigma, \tau) = p$
- $\llbracket u \rrbracket(\omega, \sigma, \tau) = \tau(u)$  ( $u$  is a local variable of type **Belief**)
- $\llbracket z \rrbracket(\omega, \sigma, \tau) = \omega(z)$ , ( $z$  is a logical variable of type **Belief**)

## 4 Proof Outlines

Proof outlines as a means for representing a correctness proof for shared-variable concurrency were introduced by S. Owicki and D. Gries [12]. A proof outline is a program annotated with formulas (assertions) that describe the program state when execution reaches that point. For simplicity, we will assume that every statement occurs between two assertions (its precondition and postcondition). The precondition of a statement describes the state upon arrival at the statement, whereas the postcondition describes the state on completion of the statement. Usually, the postcondition of a statement will be the precondition of the next statement.

Correctness of a proof outline is checked in three stages. The first stage analyzes the *local* correctness of the proof outline. Local correctness comprises checking the correctness of local statements that is, those statements which do *not* involve communication via method calls. Roughly, local correctness amounts to proving that the precondition implies the postcondition of the statement. For example, for a simple assignment to a local variable  $u := e$  with precondition  $P$  and postcondition  $Q$  this means checking if  $P \rightarrow Q[e/u]$  holds. By  $[e/u]$  we denote the substitution of  $u$  by  $e$ . It is defined by induction on the structure of assertions. The formula  $P \rightarrow Q[e/u]$  is called the verification condition of the statement. In the Sects. 4.1-4.2 we investigate the local correctness of the special commands in our language.

The second stage of the proof method consists of a variant of the cooperation test (see, e.g., [13]) which deals with the communication statements (i.e., method calls and answer statements). This test thus concerns the communication of the agents. The cooperation test inspects if all possible matches of a request and answer satisfy their specification. It is described in Sect. 4.3.

The final stage of the system is an interference freedom test. It tests if the validity of the proof outline in a particular agent class is not invalidated by the execution of commands by other agents (possibly of the same class). The interference freedom test that we propose is described in Sect. 4.4. It involves an adaptation of the interference freedom test in [12] to our object-oriented agent language.

### 4.1 Agent Creation and Instance Variables

The typical object-oriented programming statements can be proved locally correct by means of the Hoare logic described in [11]. (We also take subtyping, and inheritance into account in [11], but that does not invalidate the verification conditions. It only means they can be further simplified.)

From that paper, we use the substitutions  $[e'/e.x]$  and  $[\text{new}(a)/u]$ . The first substitution corresponds to the assignment  $e.x := e'$ , where  $x$  is an instance variable of agent  $e$ . The substitution  $[\text{new}(a)/u]$  corresponds to the statement  $u := \text{new}(a)$ . Despite their appearance these operations do not correspond to the usual notion of substitution. For example, the substitution  $[e'/e.x]$  has to take possible aliases of  $e.x$  into account, e.g., logical expressions of the form  $l.x$ , where  $l$  after the assignment  $e.x := e'$  denotes the same agent as  $e$ , and thus should also be replaced by  $e'$ . Since we cannot identify such aliases statically the substitution operation  $[e'/e.x]$  replaces  $l.x$  by the conditional expression  $if\ l[e'/e.x] = \text{self}\ \text{then}\ e'\ \text{else}\ l[e'/e.x].x\ \text{fi}$ .

Since we only allow assignments to instance variables of the form  $\text{self}.x := e$  we can instantiate the substitution  $[e'/e.x]$  to  $[e/\text{self}.x]$ . For an assignment  $\text{self}.x := e$  with precondition  $P$  and postcondition  $Q$  we obtain the verification condition  $P \rightarrow Q[e/\text{self}.x]$ .

Agent creation is slightly more complex because first of all in the state before its creation the new agent does not exist and so we cannot refer to it! This problem however can be solved by a static analysis of occurrences of  $u$  (which denotes the newly created agent after its creation) and observing that  $u$  can only be compared with other variables and can be dereferenced. These different contexts can be resolved statically, e.g.  $(u = v)[\text{new}(a)/u]$  results in **false** (for further details we again refer to [11]). Furthermore, the new agent will start executing its internal activity after its creation. We therefore also must ensure that the precondition of the internal activity of the new agent is satisfied by the state that results from its creation. Let  $P$  be the precondition of  $u := \text{new}(a)$  and let  $Q$  be its postcondition. Let  $P'$  be the precondition of the internal activity of agents of class  $a$ . The corresponding verification condition is the formula

$$P \rightarrow (Q[\text{new}(a)/u] \wedge P'[u/\text{self}][\text{new}(a)/u]) .$$

The substitution  $[u/\text{self}]$  simply replaces every occurrence of **self** in  $P'$  by  $u$ . This yields the desired result because the new agent is known in the context of its creation by the reference  $u$ .

## 4.2 Reasoning about Beliefs and Belief Updates

Our assertion language allows one to express assumptions about what follows from the belief base of a particular agent by means of expressions of the form  $l.\phi$ . Such an expression is true if  $\phi$  follows from the belief base of agent  $l$ . It is clear that belief revision commands influence the validity of such expressions. We therefore have to explore the logical foundations of such belief updates.

In the context of this paper we will restrict our attention to a simple setting where, given some signature of predicate and function symbols, beliefs are (ground) literals. In other words, we now instantiate the abstract data type **Belief** with a concrete data type. Reasoning about more advanced belief data types is left for future research.

We first consider the command  $\text{assert}(e)$ . After execution of this command we know that  $\text{self}.e$  holds. A first attempt to formulate a verification condition

would be to check whether the postcondition is implied by the precondition if we replace all occurrences of  $\text{self}.e$  in the postcondition by  $\text{true}$ . However, there might also be other expressions that denote the same agent as  $\text{self}$ . In other words, we are faced with the aliasing problem. Moreover, the belief  $e$  may also have aliases. However, we can solve this problem by introducing a conditional expression that takes this possibility into account. By  $[\text{true}/\text{self}.e]$  we denote the substitution of all aliases of  $\text{self}.e$  by  $\text{true}$ . It is defined as follows on a formula  $l.\phi$ .

$$l.\phi[\text{true}/\text{self}.e] \equiv \text{if } l = \text{self} \wedge \phi = e \text{ then true else } l.\phi \text{ fi}$$

On all other formulas the substitution corresponds to the usual notion of structural substitution. For a specification  $\{P\} \text{assert}(e) \{Q\}$  the verification condition is then given by  $P \rightarrow Q[\text{true}/\text{self}.e]$ .

*Example 5.* Let us consider the action  $\text{assert}(p)$ . Suppose we want to prove that after this command the postcondition  $\forall z : \text{Member}(\text{self}.p = z.p)$  holds. That is, we want to ensure that after adding  $p$  to our belief base all agents of sort  $\text{Member}$  share our belief of  $p$ . The weakest precondition of this command given the postcondition is

$$\forall z : \text{Member}(\text{self}.p = z.p)[\text{true}/\text{self}.p] ,$$

which amounts to checking that for every agent  $z$  in class  $\text{Member}$  we have

$$\begin{aligned} &(\text{if } \text{self} = \text{self} \wedge p = p \text{ then true else } \text{self}.p \text{ fi} \\ &= \\ &\text{if } z = \text{self} \wedge p = p \text{ then true else } z.p \text{ fi}) . \end{aligned}$$

The latter formula is equivalent to  $\forall z : \text{Member}(z \neq \text{self} \rightarrow z.p = \text{true})$ , which is indeed the weakest precondition we informally expect.

Retraction of beliefs is axiomatized similarly. A command  $\text{retract}(e)$  with precondition  $P$  and postcondition  $Q$  results in the verification condition

$$P \rightarrow Q[\text{false}/\text{self}.e] .$$

Finally, we consider assignments that involve a query of the belief base. That is, we investigate statements of the form  $u := \text{query}(e)$ . Note that  $u$  is a variable with type  $\text{bool}$  in this context. Observe that a query does not alter the belief base. Therefore we can test in the state before the assignment by means of the expression  $\text{self}.e$  what the value will be that is assigned to  $u$ . This observation leads to the following verification condition. If  $P$  is the precondition of this statement and  $Q$  is its postcondition, then the corresponding verification condition is

$$P \rightarrow Q[\text{self}.e/u] .$$

The substitution  $[\text{self}.x/u]$  completely corresponds to the usual notion of structural substitution. We have  $u[\text{self}.e/u] \equiv \text{self}.e$ , and  $v[\text{self}.e/u] \equiv v$  for every local variable  $v$  that syntactically differs from  $u$ .

### 4.3 Reasoning about Requests

In this section, we will explain in detail how reasoning about communication in our agent language takes place. We will develop a variant of the cooperation test described in [13]. The cooperation test describes the verification conditions that result from the annotation of request and answer statements.

Three statements are involved in a communication step. The first statement is a request of an agent of the form  $u := e_0.m(e_1, \dots, e_n)$ . The second statement is a corresponding answer statement `answer( $m_1, \dots, m_n$ )` such that  $m \in \{m_1, \dots, m_n\}$ . This statement must occur in the internal activity of agent class  $\llbracket e_0 \rrbracket$ . The third statement is the body  $S$  of the implementation of method  $m$  in class  $\llbracket e_0 \rrbracket$ . That is the statement that will be executed as a result of the request.

The correspondence between a request and the implementation of the method is statically given. We assume that  $m(u_1, \dots, u_n) \{ S \text{ return } e \}$  is the implementation of method  $m$  in some agent class  $A$ . Therefore it corresponds to the call  $u := e_0.m(e_1, \dots, e_n)$  if  $\llbracket e_0 \rrbracket$  is  $A$ .

However, we cannot (in general) statically determine to which answer statement a request corresponds (if any). Therefore we have to consider all possibilities. The cooperation test consists of two verification conditions for every *syntactically matching* request/answer pair. An answer statement `answer( $m_1, \dots, m_n$ )` is said to match syntactically with a request  $u := e_0.m(e_1, \dots, e_n)$  if:

- the answer statement occurs in class  $\llbracket e_0 \rrbracket$ ;
- $m \in \{m_1, \dots, m_n\}$ .

So let us consider a syntactically matching request/answer pair and a corresponding implementation  $m(u_1, \dots, u_n) \{ S \text{ return } e \}$ . We assume the following pre- and postconditions.

- $\{P[\bar{v}, \bar{z}]\} u := e_0.m(e_1, \dots, e_n) \{Q\}$
- $\{P'[\bar{v}', \bar{z}']\} \text{answer}(m_1, \dots, m_n) \{Q'\}$
- $\{P''\} S \{Q''\}$

The assertions  $P, P'$  and  $Q, Q'$  may be arbitrary assertions. The specification of the method body  $S$  should satisfy certain restrictions. The only local variables that the precondition  $P''$  can refer to are `self` and the formal parameters of the method. All other local variables are out of scope in  $P''$ . The postcondition  $Q''$  is not allowed to mention local variables at all.

The precondition of a request and an answer statement may also specify a list of local variables  $\bar{v}$  and a corresponding list of logical variables  $\bar{z}$  of equal length. The logical variables  $\bar{z}$  are used to temporarily replace the local variables  $\bar{v}$  in the context of the implementation. That is needed because the local variables of the request and answer statement are out of scope in the specification of the implementation. We provide an example of the use of this feature below.

Recall that the activity of the requesting agent is suspended during the execution of the request. This means that its local variables and its internal state cannot be altered during the request. The execution of the internal activity

of the answering agent is also suspended while it executes the corresponding method. This implies that the local variables of its internal activity also remain unchanged during the request. However, the execution of the method can result in changes to instance variables of the answering agent.

The state of the program at the start of the request is described by the conjunction of  $P$  and  $P'$ . Together, they should imply that the precondition of the implementation holds. This is checked by the following verification condition.

$$P \wedge (P'[e_0/\text{self}]) \wedge e_0 \neq \text{self} \rightarrow (P''[e_0, \bar{e}, \text{self}/\text{self}, \bar{u}, \text{caller}][\bar{v}, \bar{v}'/\bar{z}, \bar{z}']) \quad (1)$$

Note that (1) is the above mentioned implication augmented with several substitutions and the clause  $e_0 \neq \text{self}$ . This latter clause reflects the fact that an agent cannot communicate with itself. The verification condition (1) is the first half of the cooperation test.

The first substitution  $[e_0/\text{self}]$  resolves the possible clash between occurrences of  $\text{self}$  in both  $P$  and  $P'$ . These occurrences do not refer to the same object. Therefore we replace the keyword  $\text{self}$  in  $P'$  by the expression  $e_0$  since  $e_0$  is a reference to the answering agent from the perspective of the requesting agent.

Observe that it is also possible that both  $P$  and  $P'$  mention the same local variable. This should be resolved by replacing the occurrences of this local variable in  $P'$  by a fresh logical variable. A similar warning applies to the post-conditions  $Q$  and  $Q'$ .

The composite substitution  $[e_0, \bar{e}, \text{self}/\text{self}, \bar{u}, \text{caller}]$  denotes a simultaneous substitution. First of all, it involves the same substitution of  $\text{self}$  by  $e_0$  as mentioned above. Secondly, it contains a substitution of the formal parameters  $\bar{u} = u_1, \dots, u_n$  by the actual parameters  $\bar{e} = e_1, \dots, e_n$ . Finally, it specifies a substitution of the logical variable  $\text{caller}$  by  $\text{self}$ . The logical variable  $\text{caller}$  is an implicit argument of the call similar to the keyword  $\text{self}$ . It always receives the value of  $\text{self}$  from the perspective of the requesting agent. Thus we always have a reference to this agent in the context of the implementation. This reference may only be used in the annotation. We will provide an example of its use at the end of this section.

The substitution  $[\bar{v}, \bar{v}'/\bar{z}, \bar{z}']$  denotes the simultaneous replacement of the logical variables in  $\bar{z}$  and  $\bar{z}'$  by the corresponding local variables in  $\bar{v}$  and  $\bar{v}'$ . It captures the fact that local variables of the request and the answer statement are modelled in the context of the implementation by logical variables.

The state upon termination of the implementation is described by the post-condition  $Q''$ . It should imply that both  $Q$  and  $Q'$  hold. More precisely, they should be related in the following way.

$$(Q''[e_0, \bar{e}, \text{self}/\text{self}, \bar{u}, \text{caller}][\bar{v}, \bar{v}'/\bar{z}, \bar{z}']) \wedge e_0 \neq \text{self} \rightarrow (Q[\text{result}/u]) \wedge (Q'[e_0/\text{self}]) \quad (2)$$

The verification condition (2) is the final part of the cooperation test.

The substitutions in (2) are analogue versions of those in (1). The only new substitution  $[\text{result}/u]$  handles the passing of the result value. We assume that the result value  $e$  is implicitly assigned to the special-purpose logical variable

result upon termination of  $S$ . The substitution  $[\text{result}/u]$  in turn symbolically assigns the value of **result** to  $u$ .

The following example shows many features of the cooperation test. We assume that a particular agent class has the following method implementation.

$$\text{add}(u)\{ v := \text{query}(u); \text{assert}(u); \text{return } v; \}$$

The method `add` first checks if belief  $u$  already follows from its belief base. It returns this information to the caller after adding belief  $u$  to its belief base. This method has the following precondition (3) and postcondition (4).

$$\text{self}.u = z \wedge \text{self} = A \wedge \text{caller}.x = X \quad (3)$$

$$A.u \wedge \text{result} = z \wedge \text{caller}.x = X \quad (4)$$

Note that the specification also states that the instance variable  $x$  of the caller is invariant over the request. Here,  $x$  is an arbitrary instance variable of the requesting agent, and  $X$  is a logical variable of the same type.

The formulas (3) and (4) correspond to  $P''$  and  $Q''$  above. We assume in this example that the internal activity of the same class contains an answer statement that is annotated as follows.

$$\{\text{true}\} \text{answer}(\text{add}) \{\text{true}\}$$

Thus both  $P'$  and  $Q'$  in our verification conditions will be **true**. The rather weak annotation of this command suffices for our purpose. The specification of the corresponding call is more interesting.

$$\{z = a.p \wedge \text{self}.x = X [a, A]\} b := a.\text{add}(p) \{a.p \wedge b = z \wedge \text{self}.x = X\}$$

The precondition  $z = a.p \wedge \text{self}.x = X$  corresponds with  $P$  above, and  $a.p \wedge b = z \wedge \text{self}.x = X$  is the postcondition  $Q$ . This specification is a translation of the method specification to the context of the requesting agent. The singleton list  $a$  is an instantiation of the sequence  $\bar{v}$  in (1) and (2) above. Similarly,  $A$  corresponds to the sequence  $\bar{z}$ . This suffices because  $a$  is the only local variable in the specification of the request statement.

We now instantiate the verification conditions in (1) and (2) as given above with the pre- and postconditions of the example statements. This results in the following two verification conditions.

$$(z = a.p \wedge \text{self}.x = X) \wedge (\text{true}[a/\text{self}]) \wedge a \neq \text{self} \rightarrow ((\text{self}.u = z \wedge \text{self} = A \wedge \text{caller}.x = X)[a, p, \text{self}/\text{self}, u, \text{caller}][a/A]) \quad (5)$$

$$((A.u \wedge \text{result} = z \wedge \text{caller}.x = X)[a, p, \text{self}/\text{self}, u, \text{caller}][a/A]) \wedge a \neq \text{self} \rightarrow ((a.p \wedge b = z \wedge \text{self}.x = X)[\text{result}/b]) \wedge (\text{true}[a/\text{self}]) \quad (6)$$

After applying the substitutions we end up with the following implications.

$$(z = a.p \wedge \text{self}.x = X) \wedge \text{true} \wedge a \neq \text{self} \rightarrow (a.p = z \wedge a = a \wedge \text{self}.x = X)$$

$$a.p \wedge \text{result} = z \wedge \text{self}.x = X \wedge a \neq \text{self} \rightarrow (a.p \wedge \text{result} = z \wedge \text{self}.x = X) \wedge \text{true}$$

Both verification conditions are evidently valid.

#### 4.4 Interference Freedom

As explained above, the final component of the system is an interference freedom test. It tests if the validity of the proof outline in a particular agent class is not invalidated by the execution of commands by other agents (possibly of the same class). Such a test is needed because assertions in general describe properties of the global state. What we propose in this section is an adaptation of the interference freedom test in [12].

The interference freedom test ensures that every assignment  $S$  executed by an agent does not interfere with an assertion  $Q$  in another agent (but not necessarily in another class). In other words,  $Q$  should be invariant over execution of  $S$ . This test generates a verification condition for every pair of an assertion  $Q$  and a statement  $S$  that possibly interferes.

Inspection of the commands in our language learns us that there are four sorts of statements that might interfere. First of all, assignments to instance variables have to be considered. Secondly, creation of new agents might invalidate assertions. For example, because an assertion states that  $P$  holds for all existing agents of a sort  $a$ . But then  $P$  should also hold for the new agent  $u$  on completion of  $u := \text{new}(a)$ . Finally, we remark that both assertion and retraction of beliefs possibly influence the validity of assertions. Observe that assignments to local variables cannot interfere with the states of other agents.

The definition of the interference freedom tests for the different sorts of statements is straightforward given the substitutions that we defined in the previous sections. Let the assignment  $\text{self}.x := e$  have precondition  $P$ , then  $Q$  is invariant over  $\text{self}.x := e$  if:

$$(Q[z/\text{self}] \wedge P \wedge z \neq \text{self}) \rightarrow Q[z/\text{self}][e/\text{self}.x] .$$

To avoid clashes between the keywords `self` in  $Q$  and  $P$  we substitute `self` in  $Q$  by a fresh logical variable  $z$ . The statement can only interfere if it is executed in parallel by a different agent. Hence the additional assumption  $z \neq \text{self}$ .

Similarly, we should check that for any assignment  $u = \text{new}(a)$  with precondition  $P$  we have that  $Q$  is invariant. That is, we should check

$$(Q[z/\text{self}] \wedge P \wedge z \neq \text{self}) \rightarrow Q[z/\text{self}][\text{new}(a)/u] .$$

The test for a statement `assert( $e$ )` (7) and a command `retract( $e$ )` (8) are also similar.

$$(Q[z/\text{self}] \wedge P \wedge z \neq \text{self}) \rightarrow Q[z/\text{self}][\text{true}/\text{self}.e] \quad (7)$$

$$(Q[z/\text{self}] \wedge P \wedge z \neq \text{self}) \rightarrow Q[z/\text{self}][\text{false}/\text{self}.e] \quad (8)$$

The inference freedom test completes our proof method for the presented object-oriented agent language.

## 5 Conclusions

Reasoning about agent communication is a relatively new topic. In [14] the verification problem of agent communication is defined and examined at a rather

high level of abstraction. The different components of a verification framework for agent communication are outlined in [15]. An actual verification method for agent communication is developed in [16] that is based on the synchronous handshaking mechanism of Hoare's paradigm of CSP (Communicating Sequential Process) and the information-processing aspects of CCP (Concurrent Constraint Programming). In this framework, agent communication proceeds through passing information along bilateral channels rather than by requests like in the current paper.

In this paper we have described (to the best of our knowledge) a first proof method based on assertions for an object-oriented coordination language for multi-agent systems. To our opinion this work presents a promising first step towards a further integration of OO and agent technology (for example, 'agentifying' inheritance and subtyping) and the application of formal methods to complex agent systems. Additionally, this work also provides a promising basis for the application of concepts like compositionality developed in the semantics and proof theory of concurrent systems [17].

One of the main proof theoretical challenges for future developments concerns an integration of a new class of modal logics [18] for representing the beliefs of agents which capture the object-oriented mechanism of dynamic referencing of agents by means of corresponding parameterized modal operators. A perhaps even more ambitious future goal is to extend the proof method to agent-oriented programming languages like 3APL [19] which involve very expressive mechanisms for belief and goal revision.

Currently, we are working on soundness and completeness proofs. The soundness proof consists of a fairly straightforward induction on the length of the computations. The completeness proof consists of showing that the verification conditions hold for proof outlines which are defined in terms of so-called semantic reachability predicates (see [17]).

In the near future we are also planning to develop tools for the automatic generation of the verification conditions and the use of theorem provers along the lines of [10].

## References

1. Rash, J., Rouff, C., Truszkowski, W., Gordon, D., Hinchey, M., eds.: Formal Approaches to Agent-Based Systems (Proc. FAABS 2001). Volume 1871 of LNAI. Springer (2001)
2. Hinchey, M., Rash, J., Truszkowski, W., Rouff, C., Gordon-Spears, D., eds.: Formal Approaches to Agent-Based Systems (Proc. FAABS 2002). Volume 2699 of LNAI. Springer (2003)
3. Wooldridge, M., Ciancarini, P.: Agent-oriented software engineering: The state of the art. In Wooldridge, M., Ciancarini, P., eds.: Agent-Oriented Software Engineering. Volume 1957 of LNCS. (2001) 1–28
4. Meyer, J.J.C.: Tools and education towards formal methods practice. In Hinchey, M., Rash, J., Truszkowski, W., Rouff, C., Gordon-Spears, D., eds.: Formal Approaches to Agent-Based Systems (Proc. FAABS 2002). Volume 2699 of LNAI. (2003) 274–279

5. van Eijk, R., de Boer, F., van der Hoek, W., Meyer, J.J.C.: Generalised object-oriented concepts for inter-agent communication. In Castelfranchi, C., Lesprance, Y., eds.: *Intelligent Agents VII*. Volume 1986 of LNAI. (2001) 260–274
6. Hoare, T.: Assertions. In Broy, M., Pizka, M., eds.: *Models, Algebras and Logic of Engineering Software*. Volume 191 of NATO Science Series. IOS Press (2003) 291–316
7. Floyd, R.W.: Assigning meaning to programs. In: *Proc. Symposium on Applied Mathematics*. Volume 19., American Mathematical Society (1967) 19–32
8. Meyer, B.: *Eiffel: The Language*. Prentice-Hall (1992)
9. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* **12** (1969) 576–580
10. de Boer, F., Pierik, C.: Computer-aided specification and verification of annotated object-oriented programs. In Jacobs, B., Rensink, A., eds.: *FMOODS V*, Kluwer Academic Publishers (2002) 163–177
11. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. In Najm, E., Nestmann, U., Stevens, P., eds.: *Formal Methods for Open Object-Based Distributed Systems (FMOODS) VI*. Volume 2884 of LNCS. (2003) 64–78
12. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. *Acta Informatica* **6** (1976) 319–340
13. Apt, K., Francez, N., de Roever, W.: A proof system for communicating sequential processes. *ACM Transactions of Programming Languages and Systems* **2** (1980) 359–385
14. Wooldridge, M.: Semantic issues in the verification of agent communication. *Autonomous Agents and Multi-Agent Systems* **3** (2000) 9–31
15. Guerin, F., Pitt, J.: Verification and compliance testing. In Huget, M.P., ed.: *Communication in Multiagent systems: Agent communication languages and conversation policies*. Volume 2650 of LNAI. (2003) 98–112
16. Eijk, R.v., Boer, F.d., Hoek, W.v.d., Meyer, J.J.: A verification framework for agent communication. *Journal of Autonomous Agents and Multi-Agent Systems* **6** (2003) 185–219
17. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: *Concurrency Verification*. Volume 54 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
18. Meyer, J.J.C., van der Hoek, W.: *Epistemic Logic for AI and Computer Science*. Volume 41 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1995)
19. Hindriks, K., de Boer, F., van der Hoek, W., Meyer, J.J.: Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems* **2** (1999) 357–401