Generalised Object-Oriented Concepts for Inter-Agent Communication

Rogier M. van Eijk, Frank S. de Boer,
Wiebe van der Hoek and John-Jules Ch. Meyer

Utrecht University, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{rogier, frankb, wiebe, jj}@cs.uu.nl

Abstract. In this paper, we describe a framework to program open societies of concurrently operating agents. The agents maintain a subjective theory about their environment and interact with each other via a communication mechanism suited for the exchange of information, which is a generalisation of the traditional rendez-vous communication mechanism from the object-oriented programming paradigm. Moreover, following object-oriented programming, agents are grouped into agent classes according to their particular characteristics; viz. the program that governs their behaviour, the language they employ to represent information and most interestingly the questions they can be asked to answer. We give and operational model of the programming language in terms of a transition system for the formal derivation of computations of multi-agent programs.

1 Introduction

The field of multi-agent systems is a rapidly growing research area. Although in this field there is no real consensus on what exactly constitutes an agent (in fact, this also applies to the notion of an object, which nonetheless has proven to be a successful concept for the design of a new generation of programming languages), there are some generally accepted properties attributed to it [14]. An *agent* is viewed upon as an *autonomous* entity that shows both a *reactive* and *proactive* behaviour by perceiving and acting in the environment it inhabits. Moreover, it has a *social* ability to interact with other agents in a multi-agent context. In the stronger conception of agency, an agent is additionally assumed to have a mental state comprised of *informational* attitudes (like knowledge and belief) and *motivational* attitudes (like goals, desires and intentions).

What this enumeration of properties shows, is that rather than being thought of as a computational entity in the traditional sense, an agent is viewed upon as a more elaborate software entity that embodies particular human-like characteristics. For instance, an important issue in the rapidly growing research area of electronic commerce, is the study of whether agents can assist humans in their tedious tasks of localising, negotiating and purchasing goods. The negotiation activities, for example, in general comprise the exchange of information of a highly complex nature, requiring the involved parties to employ high-level modalities as knowledge and belief about the knowledge and belief of the other parties.

The emerging novel application areas such as electronic commerce require the development of new *programming paradigms*, as the emphasis of programming involves a shift from performing *computations* towards the more involved concepts of *interaction* and *communication*. The object-oriented languages, for instance, as the term indicates, are primarily designed to program systems consisting of a collection of objects. In general, such an *object* is modelled as an entity whose state is stored in a set of variables and that is assigned a set of methods that can be invoked to operate on these variables. In fact, it constitutes a type of *data encapsulation*; other objects can inspect and change the state of the object through the invocation of one of its methods. The central interaction mechanism of this paradigm is thus one that proceeds via method invocations, which in our opinion, is not a mechanism suited to the communication of high-level types of information. Moreover, the field of multi-agent systems is not so concerned with programming objects, but, if we draw the classical philosophical distinction, more with programming *subjects*. That is, in this paradigm the subjective point of view of computational entities is explicated: the state of an agent is not given by an objective state mapping variables to values, but constitutes a mental state that expresses the agent's subjective view on itself and its environment. Hence, the object-oriented languages, in their current forms, are not well-suited to program multi-agent systems.

Among multi-agent programming languages like AgentSpeak [13] and DESIRE [3], in previous research, we have developed an abstract multi-agent programming framework that has a well-defined *formal semantics* [4–6]. The framework is fit to program open societies of concurrently operating agents, which maintain a subjective theory about their environment, and which interact with each other via a communication mechanism for the *exchange of information* rather than the communication of values (like in [10]). In this paper, we further extend this framework with generalisations of concepts from the object-oriented programming paradigm such as *agent classes*, communication based on a *rendez-vous* mechanism and the introduction of *questions*.

2 Inter-Agent Communication

We view an agent has a computational entity that operates together with a collection of other agents in some environment. It maintains a private, subjective theory about its environment, given by its specific expertise and reasoning capabilities, and interacts with other agents via the communication of information. In particular, we explicate the following constituents of an agent.

First of all, an agent has its own activity given by the execution of a private *program*, which describes the agent's reasoning patterns and its behaviour. The program is executed in the context of a *mental state* that consists of motivational attitudes (as goals) and informational attitudes (as beliefs). In this paper, we only consider the second category of mental attitudes; the motivational attitudes are outside the scope of the present paper and studied in other papers in this volume [9, 12]. That is, we assume that each agent has its own belief base. Additionally, the agent has a private *first-order system* to represent and process information. In addition to a first-order language, this system comprises an entailment relation $\vdash$ (e.g., a prolog interpreter, theorem-prover) to decide whether formulae constitute a consequence of the current belief base. This

operator for instance indicates how the agent deals with negations; i.e., whether it employs a negation-as-failure strategy, a finite-failure strategy, and so on. Moreover, it can be thought of representing the agent's decision-making capabilities. Finally, the first-order system comprises an operation $\circ$ to *update* the agent's belief base with newly acquired information and newly established conclusions. That is, we assume that each agent employs its private belief revision strategy [8].

Communication via Questions. Let us give a sketch of a communication mechanism that is based upon the notion of a *question*, by analogy with the notion of a *method* from object-oriented programming. Given a first-order system that comprises a set C of first-order formulae and an entailment relation $\vdash$, a question is of the form:

$$q(\mathbf{x}) \leftarrow \varphi,$$

where q constitutes the name of the question, $\mathbf{x}$ is a sequence of formal parameters and φ denotes a formula in $\mathcal{C}$. Consider for instance the following question to ask whether a particular liquid x has reached its boiling point:

$$\begin{aligned} boiling(x) \leftarrow\ & Hydrogen\_oxide(x) \wedge temp(x) \geq 100°C) \vee \\ & Hydrogen\_sulfide(x) \wedge temp(x) \geq -60°C). \end{aligned}$$

According to this definition, a liquid x has reached its boiling point in case it concerns hydrogen oxide (water) and its current temperature is at least 100 degrees Celcius, or the liquid is hydrogen sulfide (hepatic acid) in which case the temperature should be equal to or higher than -60 degrees Celcius. A question q is also assigned an *interface signature* Σ_q that determines which details of the vocabulary of the body of the question are visible from the outside. Let us consider the following two instances $\Sigma_1 = \{Hydrogen\_oxide,\ Hydrogen\_sulfide\}$ and $\Sigma_2 = \Sigma_1 \cup \{temp,\ \ldots, -2°C,\ -1°C,\ 0°C,\ 1°C,\ 2°C, \ldots\}$ of the signature $\Sigma_{boiling}$. In case $\Sigma_{boiling} = \Sigma_1$ the details of the question that are visible from the outside are the types of liquids for which the question is defined. In case $\Sigma_{boiling} = \Sigma_2$ the function $temp$ and the constants to denote temperatures are also visible.

An *invocation* of the above question $q(\mathbf{x}) \leftarrow \varphi$ is of the form:

$$q(\mathbf{t} \mid \psi),$$

where $\mathbf{t}$ is a sequence of closed terms of the same length as $\mathbf{x}$ and ψ is a formula that is expressed in the interface signature Σ_q of the question. The purpose of ψ is to give additional information about the actual parameters $\mathbf{t}$. The invocation amounts to checking that the following holds: $B \wedge (\mathbf{x} = \mathbf{t}) \wedge \psi \vdash \varphi$, where B denotes the current information store of the answering agent. Thus, the body φ of the question should follow from the current information store B together with the instantiation of the formal parameters $\mathbf{x}$ with the actual parameters $\mathbf{t}$ and the additional information ψ as provided in the invocation.

For instance, a possible invocation of the question $boiling$ in the situation its interface signature is given by Σ_1, is the following: $boiling(c \mid Hydrogen\_oxide(c))$. This invocation amounts to testing that the following holds:

$$\begin{aligned} B \wedge x = c \wedge \\ Hydrogen\_oxide(c) \end{aligned} \quad \vdash \quad \begin{aligned} & (Hydrogen\_oxide(x) \wedge temp(x) \geq 100°C) \vee \\ & (Hydrogen\_sulfide(x) \wedge temp(x) \geq -60°C), \end{aligned}$$

where B denotes the answering agent's information store. Thus the invocation is successful if the information state B contains information that confirms the temperature of the fluid c to be at least 100 degrees Celcius.

In case the signature $\Sigma_{boiling}$ is equal to Σ_2, information about the temperature of the liquid c can also be included in the invocation of the question, such that it does not need to be contained in the answering agent's belief state. For instance, the following is then a possible invocation: $boiling(c \mid Hydrogen\_oxide(c) \wedge temp(c) = 102°C)$, in which the additional information $temp(c) = 102°C$ is supplied.

We make one more refinement: we admit the signature of the invocation to be different from the signature of the question. That is, we allow agents to invoke a question with a distinct signature, after which a translation takes place, in which the symbols in the invocation are mapped to the symbols of the question. An example is the following invocation: $boiling(c \mid Water(c) \wedge temp(c) = 215°F)$. We assume a translation function that maps the predicate $Water$ to the predicate $Hydrogen\_oxide$ and the temperature $215°F$ in degrees Fahrenheit to the corresponding temperature $102°C$ in degrees Celcius.

An important feature of the communication mechanism is that it not only allows us to define *what* questions an agent can be asked, but also allows us to hide the details of the underlying first-order system by analogy with the ideas of object-oriented programming. Consider for instance an alternative definition of the question $boiling$ like the following:

$$\begin{aligned} boiling(x) \leftarrow\ & (Hydrogen\_oxide(x) \wedge temp(x) \geq 212°F) \vee \\ & (Hydrogen\_sulfide(x) \wedge temp(x) \geq -76°F), \end{aligned}$$

where the interface signature is given by $\{Hydrogen\_oxide,\ Hydrogen\_sulfide\}$. This question is associated with agents that represent the temperature in degrees Fahrenheit instead of degrees Celcius. From the outside however, this difference in implementation is not visible. That is, an asking agent is not concerned with the representation of the temperature, that is, it does not need to know whether the temperature is measured in degrees Celcius or in degrees Fahrenheit.

3 Concepts from Object-Oriented Programming

In defining the above communication mechanism, we adapt and generalise mechanisms and techniques from the object-oriented programming paradigm.

Object Classes. An important characteristic of the object-oriented programming paradigm is that object populations are structured into *object classes*. That is, each object in a population is an *instance* of a particular class, which is of the form: $C = \langle \mathbf{x}, M \rangle$ where C is the name of the class, $\mathbf{x}$ is comprised of the variables each object of the class employs and M collects the methods that are used to operate on these variables. That is to say, each object has its *own* set of variables and methods, but the names of these variables and the code that implements these methods are the same among all objects in the class. The class thus defines a *blueprint* for the creation of its instances.

Active Objects. In several languages, like for instance the parallel object-oriented programming language POOL [1], rather than being *passive* entities, objects are assigned an activity of their own. That is, in these languages, object classes are of the form: $C = \langle \mathbf{x}, M, S \rangle$ where the additional constituent S denotes a program which governs the behaviour of the objects of the class. The main purpose of this program S is to maintain control over the invocation of the object's methods. That is, these methods cannot be invoked at any arbitrary point in the execution, but only at certain points, which are controlled by the program S. Moreover, at each point, the invokable methods typically constitute a *subset* of the entire set of methods.

One of the issues in the field of object-oriented programming is the design of programming languages for *concurrent* systems in which the object population *dynamically evolves* over time. An illustrative representative of these languages is the above mentioned language POOL [1]. A program in this language is comprised of a set of object class definitions, in which one class is identified as the *root class*. The execution of the program starts with the creation of an instance of the root class, which is marked as the *root object*. This root object executes the program that is defined in its class during which it creates new objects of the other classes. In this manner, a dynamically evolving population of concurrently operating objects is attained.

Communication Between Objects. The interaction mechanism that is used in object-oriented languages like POOL, is based on the classical notion of a *rendezvous* [2]. This constitutes a communication mechanism in which a process, say A, executes one of its procedures on behalf of another process B. In particular, the rendezvous can be viewed upon as to consist of three distinct stages. First, the process B *calls* one of the procedures of the process A. This is followed by the *execution* of the corresponding procedure by A, during which the execution of the calling process B is suspended. Finally, there is the communication of the *result* of the execution back to A, upon which A resumes its execution. It follows that a rendezvous comprises two points of synchronisation. First, there is the call involving the exchange of the actual procedure parameters from the caller to the callee and secondly, there is the communication of the results back from the callee to the caller

4 Programming Language

In this section, we define the syntax of our multi-agent programming language, in which we adapt and generalise the above concepts from the object-oriented programming paradigm in the light of communication between agents. It is important to keep in mind that the driving motivation of this generalisation is the fact that, in contrast to the notion of an object, agents are viewed upon as computational entities that process and reason with high-level forms of *information* rather than with low-level data as expressions and values.

First, we introduce the notion of a *first-order system*. We assume a set Var of logical variables with typical element x, where we use the notation Vec to denote the set of finite sequences over Var, with typical element $\mathbf{x}$.

Definition 1 (*First-order systems*)
A first-order system $\mathcal{C}$ is a tuple $\mathcal{C} = (C, \vdash, \circ)$, where C is a set $form(\Sigma)$ of formulae from a sorted first-order language over a particular signature Σ. A signature is comprised of constant, function and relation symbols. We use the notation $term(\Sigma)$ to denote the set of terms over the signature Σ and $cterm(\Sigma)$ to denote the set of closed terms, which are the terms that do not contain variables from Var. Additionally, $\vdash \subseteq C \times C$ constitutes an entailment relation, and $\circ : (C \times C) \to C$ denotes a belief update operator, such that $\varphi \circ \psi$ constitutes the belief base φ that has been updated with the information ψ.

We assume a global set $\mathcal{Q}$ that consists of *question names*. Questions templates and question invocations are then defined as follows.

Definition 2 (*Question templates and question invocations*)
Given a first-order system $\mathcal{C} = (C, \vdash, \circ)$ over a signature Δ, the set $\mathcal{Q}_t(\mathcal{C})$ of question *templates* and the set $\mathcal{Q}_i(\mathcal{C})$ of question *instances* over $\mathcal{C}$, are defined as follows:

$$\mathcal{Q}_t(\mathcal{C}) = \{q(\mathbf{x}, \mathbf{y}) \leftarrow \varphi \mid \ q \in \mathcal{Q},\ \mathbf{x}, \mathbf{y} \in Vec,\ \varphi \in \mathcal{C}\}$$

$$\mathcal{Q}_i(\mathcal{C}) = \{q(\mathbf{s}, \mathbf{t} \mid \psi) \mid \ q \in \mathcal{Q},\ \mathbf{s}, \mathbf{t} \in cterm(\Delta),\ \psi \in \mathcal{C}\}.$$

A question template is of the form $q(\mathbf{x}, \mathbf{y}) \leftarrow \varphi$, where the *head* $q(\mathbf{x}, \mathbf{y})$ is comprised of the name q of the question and the sequences $\mathbf{x}$ and $\mathbf{y}$ of variables constitute the formal parameters that in a communication step are to be instantiated with actual values. The difference between the sequences is that the variables $\mathbf{x}$ are global variables of which the scope extends over the boundaries of the question, whereas the variables $\mathbf{y}$ are strictly local to the question. As we will see later on, the purpose of the parameters $\mathbf{x}$ is to support the communication of dynamically generated agent names. The *body* of the question is a logical formula φ that comes from the underlying first-order system $\mathcal{C}$. Moreover, each question q is also assigned an interface signature Σ_q that defines the visible vocabulary of the question body.

A question invocation is of the form $q(\mathbf{s}, \mathbf{t} \mid \psi)$, where q denotes the name of the question, the sequences $\mathbf{s}$ and $\mathbf{t}$ of closed terms denote the actual parameters that are to be substituted for the formal parameters $\mathbf{x}$ and $\mathbf{y}$ of the question, respectively, and the formula ψ constitutes additional information regarding these actual parameters, which is either already expressed in the interface signature Σ_q or will be translated into this signature (via a translation function).

By analogy with the object-oriented programming paradigm, we introduce the notion of an agent class.

Definition 3 (*Agent classes*)
An *agent class* $\mathcal{A}$ is defined to be a tuple of the following form $\mathcal{A} ::= (\mathcal{C}, Q, D, S, \varphi)$, where $\mathcal{C}$ is a first-order system, $Q \subseteq \mathcal{Q}_t(\mathcal{C})$ is a set of question templates, S denotes a programming statement in which procedures that are declared in the set D can be invoked, and $\varphi \in \mathcal{C}$ constitutes an information store.

A class $\mathcal{A}$ thus consists of a first-order system $\mathcal{C}$ describing the language and operators the agents in the class employ to represent and process information. Secondly, it comprises a set Q of question templates that the agents in the class can be asked to answer,

where for technical convenience we require that Q does not contain two questions with the same name. Thirdly, the class definition contains a statement S that describes the behaviour of the agents in the class; that is, upon its creation each agent of the class will start to execute this statement. The syntax of atomic statement is given in Definition 5, while the syntax of complex statements is discussed in Definition 6. The set D collects the declarations of the procedures that can be invoked in S, which, in case it is clear from the context, is usually omitted from notation. Finally, φ denotes the initial information store of the agents in the class.

A program then simply consists of a collection of agent classes.

Definition 4 (*Multi-agent programs*)
A *multi-agent program* $\mathcal{P}$ is a tuple $(\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k)$ of agent classes.

Each class in a multi-agent program $\mathcal{P}$ defines a blueprint for the creation of its instances. The execution of the program consists of a dynamically evolving agent population in which *existing instances* have the ability to integrate new instances. Additionally, we can think of the possibility to extend the population with agents that have been created *outside* the system (for instance, by a particular user). However, the latter type of agent integration falls outside the scope of the current framework, as it requires an outer layer of user interfaces on top of the programming language.

We assume that each class $\mathcal{A}$ in the program also makes up a *sort* $\mathcal{A}$, which we will refer to as an *agent sort*. We assume that the first-order system of each class in the program includes constants and variables of the different agent sorts, which are used to denote instances of the corresponding classes. In particular, we assume that there is a constant $self$ that is used by an agent to refer to itself.

Definition 5 (*Actions*)
For each class $\mathcal{A} = (\mathcal{C}, Q, D, S, \psi)$ in a multi-agent program $\mathcal{P}$, the *atomic actions* a of the complex statement S and the procedures in D are defined as follows.

Actions for information processing $a ::= \texttt{update}(\varphi) \mid \texttt{query}(\varphi)$
The action $\texttt{update}(\varphi)$ denotes the update of the agent's information store with the information φ, while the action $\texttt{query}(\varphi)$ denotes a test that the formula $\varphi \in \mathcal{C}$ is a consequence of the current information store.

Action for integration $a ::= \texttt{new}(x)$
An action of the form $\texttt{new}(x)$ denotes the act of integrating a new agent in the system. This new agent will be an instance of the agent class $\mathcal{B}$, where $\mathcal{B}$ denotes the sort of the variable x. Moreover, the action constitutes a *binding* operation in the sense that it binds the free occurrences of the variable x in the subsequent program.

Action for asking $a ::= \texttt{ask}(x, q(\mathbf{s}, \mathbf{t} \mid \varphi))$
This action denotes the act of asking the agent x the question $q(\mathbf{s}, \mathbf{t} \mid \varphi)$ in the set $\mathcal{Q}_i(\mathcal{C})$. If the class $\mathcal{B}$ denotes the sort of the variable x, it is required that the set of question templates defined in $\mathcal{B}$ includes a template with the head $q(\mathbf{x}, \mathbf{y})$, for some formal parameters $\mathbf{x}$ and $\mathbf{y}$ that are of the same length as $\mathbf{s}$ and $\mathbf{t}$, respectively. Additionally, we require the variable x to be *bound*, that is, the action should occur in the scope of

the binding action $\texttt{new}(x)$ or of the binding action of the form $\texttt{answer}(p(\mathbf{u}, \mathbf{v}))$, where x is an element of $\mathbf{u}$.

Action for answering $a ::= \texttt{answer}(q(\mathbf{x}, \mathbf{y}))$
This action denotes the act of answering a question template $q(\mathbf{x}, \mathbf{y}) \leftarrow \varphi$ in Q, where the sequence $\mathbf{x}$ collects the global formal parameters, $\mathbf{y}$ denotes the local formal parameters and φ denotes the body of the question.

Like the operation $\texttt{new}(x)$, the action $\texttt{answer}(q(\mathbf{x}, \mathbf{y}))$ constitutes a *binding* operator that binds the global variables $\mathbf{x}$ (but not the local variables $\mathbf{y}$) in the subsequent agent program. In fact, it gives rise to a *block construction*; that is, the variables can be referred to inside the scope of the operator, but outside the scope these variables do not exist. In particular, the idea is that the variables $\mathbf{x}$ are used to store the actual parameters of the question, which can referred to in the agent's subsequent program. Since the names of dynamically integrated agents are unknown at compile-time, it is indispensable that these identifiers are communicated at run-time. Consider for instance the following programming statement: $\texttt{answer}(q_1(x, y)) \cdot \texttt{ask}(x, q_2)$. In the statement, the occurrence of x in the second action is bound by the first action. That is, in answering the question q_1 the agent comes to know about a particular agent which name will be substituted for the variable x, after which this agent is asked the question q_2. So, in answering questions an agent can extend its circle of acquaintances with new agents that it had not been aware of before.

In the communication mechanism, after answering a question, the actual parameters of a question can thus be referred to in the subsequent agent program. In the literature, this technique is also known as *scope extrusion*: the feature to extend the original scope of variables to larger parts of a program, like for instance in the π-calculus [11].

Definition 6 *(Statements)*

$$S ::= a \cdot S \mid S_1 + S_2 \mid S_1 \,\&\, S_2 \mid \texttt{loc}_x S \mid p(\mathbf{x}) \mid \texttt{skip}.$$

Complex statements are composed by means of the sequential composition operator $\cdot$, the non-deterministic choice operator $+$ and the internal parallelism operator $\&$. Additionally, there is the possibility to define local variables and to have procedure calls. Finally, the statement `skip` denotes the empty statement that has no effects.

Example 7 We illustrate the syntax of the programming language by means of the following example. Consider a multi-agent program $\mathcal{P}$ that consists of two classes: $(Booksellers, Bookbuyers)$. Let us first describe in detail how the class of book-selling agents is defined: $Booksellers = (\mathcal{C}, Q, D, \texttt{HandleOffer}, \varphi)$. The first constituent of this class is a first-order system $\mathcal{C} = (C, \vdash, \circ)$, which contains formulae over a signature Δ. The predicates in this signature Δ are given by the ordering relations $<$ and $\geq$, a predicate $Customer(x)$ to denote that the agent x is a customer, a predicate $Defaulter(x)$ to denote that the agent x is notorious for not paying for its purchases, predicates $Novel(y)$ and $Comic(y)$ to denote that the book y is a novel and a comic, respectively, and finally a relation $Sold(x, y)$ to denote that the book x has been sold to agent y. Additionally, Δ contains a set Σ of constants: $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$, which

is comprised of a set Σ_1 of constants to denote *agents* with typical element n, a set Σ_2 of constants to denote *books* with typical element b and finally, a set Σ_3 to denote the *prices* of books in euros with typical element $\in m$. Furthermore, the entailment relation $\vdash$ implements a negation-as-failure strategy; that is, for all $\varphi, \psi \in \mathcal{C}$, the following holds: $\varphi \nvdash \psi \Rightarrow \varphi \vdash \neg\psi$. The set Q of the class consists of the following question templates:

$$
\begin{array}{ll}
NewBuyer(x, \epsilon) & \leftarrow \neg\, Customer(x) \\
AcceptOffer(xy, z) & \leftarrow Customer(x) \wedge \\
 & \quad \neg Defaulter(x) \wedge \neg\exists u(Sold(y, u)) \wedge \\
 & \quad ((Novel(y) \wedge z \geq \in\!20) \vee (Comic(y) \wedge z \geq \in\!4)) \\
RefuseOffer(\epsilon, xyz) & \leftarrow Customer(x) \wedge \\
 & \quad (Defaulter(x) \vee \exists u(Sold(y, u)) \vee \\
 & \quad (Novel(y) \wedge z < \in\!20) \vee (Comic(y) \wedge z < \in\!4)).
\end{array}
$$

Moreover, the interface signature of the question $NewBuyer$ is given by Σ_1, while the signatures of the other two questions is given by Σ. So, the question $NewBuyer(x, \epsilon)$ amounts to asking that the agent x is not already a customer. This question is typically asked for its side-effects, as we will see below. The question $AcceptOffer(xy, z)$ is employed to ask that z is an acceptable offer from the agent x for the book y, while $RefuseOffer(\epsilon, xyz)$ is used to ask that such an offer is unacceptable.

The procedure `HandleOffer`, which is declared in the set D, is defined as follows:

$$
\begin{array}{l}
\texttt{HandleOffer} :- \\
\quad (\texttt{answer}(AcceptOffer(xy, z)) \cdot \\
\qquad \texttt{update}(Sold(y, x)) \cdot \\
\qquad \texttt{HandleOffer}) \\
\quad + \\
\quad ((\texttt{answer}(RefuseOffer(\epsilon, xyz)) \cdot \\
\qquad \texttt{HandleOffer}) \\
\quad + \\
\quad (\texttt{answer}(NewBuyer(x, \epsilon)) \cdot \\
\qquad \texttt{update}(customer(x)) \cdot \\
\qquad \texttt{HandleOffer}))
\end{array}
$$

Thus, the procedure loops over the non-deterministic choice between the acceptance of an offer, the refusal of an offer and the registration of a new customer. Note that the question $NewBuyer$ is typically answered for its side effect, namely the binding of the global formal parameter x to an actual parameter. In particular, the scope of the variable x is extruded to the next action of the program in which the information $customer(x)$ about x is added to the information store. Provided that a buyer has been registered as a customer, novels and comics are sold in case the offered price exceeds 20 euros and 4 euros, respectively. In the other cases, including the situation that the customer is known to be a defaulter and the situation that the book has been sold already, offers are refused.

Finally, the information store φ of the class is a conjunction of atomic formulae of the form $Novel(b)$ and $Comic(b)$, where b ranges over the elements of Σ_2. The store

thus defines for each of the books whether it is a novel or a comic. In particular, we assume it to contain the information $Novel(b_{737})$.

Additionally, there is the following class of agents that buy want to buy a book b_{737} from the booksellers: $Bookbuyers = (\mathcal{D}, R, F, \texttt{BuyBook737}, true)$. The first-order system $\mathcal{D}$ is defined over a signature that is comprised of a relation $Bought(x, y)$ to denote that the book x has been bought from agent y. Additionally, it contains the sub-signatures Σ_1 of constants to denote *agents* with typical element n, a set Σ_2 of constants to denote *books* with typical element b and thirdly, instead of constants to denote the price of books in euros, a set Σ_4 of constants to represent the prices of books in *dollars* with typical element $\$m$. The set R consists of the following question:

$$NewSeller(x, \epsilon) \leftarrow true,$$

where x is a variable of the sort $Booksellers$. This question is typically employed for its side effect, namely to come to know about a selling agent x.

The procedure `BuyBook737`, which is declared in the set F, is defined as follows:

```
BuyBook737 :−
   answer(NewSeller(x, ϵ))·
      ask(x, NewBuyer(self, ϵ | true))·
         ask(x, AcceptOffer((self, b737), $22) | true))·
            update(Bought(b737, x))
         +
         ask(x, RefuseOffer(ϵ, (self, b737, $22) | true))·
            BuyBook737
```

So, in this procedure, the agent comes to know about a bookselling agent x, with which it registers itself through an invocation of the question $NewBuyer$. Subsequently, the bookseller x is asked whether or not it sells the book b_{737} for a price of 22 dollars. If this offer is accepted the information store is subsequently updated with the information $Bought(b_{737}, x)$ upon which the procedure terminates. On the other hand, if it is refused, the procedure is recursively invoked in order to come to know about (another) bookselling agent. It is important to note here that the price $\$22$ in dollars needs to be translated into a price in euros. For instance, one can think of a translation function that multiplies a price in dollars with a factor 0.95, yielding the corresponding price €21 in euros.

5 Operational Semantics

In this section, we define the operational semantics of multi-agent programs. Let us first consider the notion of an agent, which is an instance of a particular agent class.

Definition 8 (*Agents*)
An instance of an agent class is called an *agent*. Given a class $\mathcal{A} = (\mathcal{C}, Q, D, S, \varphi)$, the initial configuration of an agent is the tuple $\langle n, S, \varphi \rangle$, where n is a unique constant of sort $\mathcal{A}$.

An agent is assigned a unique name n, which can be used by other agents in a system to address it. Upon integration in the system, the initial information store of the agent is given by the store φ from the class and its initial programming statement is S. Moreover, the agent uses the first-order system $\mathcal{C}$ to represent and process information, while the questions it can be asked to answer are collected in the set Q. Additionally, the procedures that can be invoked in S are collected in the set D. As the constituents $\mathcal{C}$, Q and D remain invariant under execution and can be inferred from the class, they are not included in the agent's configuration. Although also the name n of the agent stays the same during execution, it is still included in the agent's configuration simply because it allows us to distinguish between the different instances of the same class.

Definition 9 (*Multi-Agent Systems*)
Given a program $\mathcal{P} = (\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k)$, a *multi-agent system* is a set of instances of the agent classes $\mathcal{A}_i$ $(0 \leq i \leq k)$.

A multi-agent system is a population of agents that dynamically evolves itself through the dynamic integration of new instances of the classes of the program.

The operational semantics of the programming language is defined in terms of transitions between multi-agent systems. Such a *transition* is of the form:

$$A \longrightarrow A' \quad \text{if } \textit{cond},$$

which denotes a computation step of the multi-agent system A, where A' constitutes the part of the agent system that still needs to be executed. This computation step is only allowed to take place in case the condition $cond$ is satisfied.

In order to be able to define the transition rules, we use the notation $S[\mathbf{t}/\mathbf{x}]$ to denote the simultaneous substitution of the terms $\mathbf{t}$ for the variables $\mathbf{x}$ in the statement S. We define the transition rules for the integration of agents, communication and parallel execution. The transition rules for the other actions and for statements are similar to those in [5, 4, 6] and therefore omitted here.

Definition 10 (*Transition for agent integration*)
Let the class $\mathcal{A} = (\mathcal{C}, Q, D, T, \psi)$ be the agent sort of the variable x, and let m be a *fresh* agent constant of sort $\mathcal{A}$, we have the transition:

$$\langle n, \mathtt{new}(x) \cdot S, \varphi \rangle \longrightarrow$$

$$\langle n, S[m/x], \varphi \rangle, \langle m, T[m/self], \psi \rangle$$

In this transition, a new agent m of the class $\mathcal{A}$ is created that starts to execute the statement T defined in its class, and which has an information store that is given by the formula ψ. In the program S of the integrating agent n, the variable x is substituted by the actual name m of the integrated agent. Additionally, in the program of the new agent, the constant $self$ is replaced by its actual name m. Note that we assume a global naming mechanism: the constant m is not only fresh with respect to the above local transition rule, but fresh with respect to the entire agent population. This assumption ensures that all agents in the population are assigned a distinct name.

Definition 11 (*Transition for rendezvous*)
Given the question template $q(\mathbf{x}, \mathbf{y}) \leftarrow \psi$ with interface signature Σ, let Δ denote the signature (excluding agent constants) of a formula φ and terms $\mathbf{s}$ and $\mathbf{t}$. The transition is as follows:

$$\begin{array}{ll} \langle n, \mathtt{ask}(m, q(\mathbf{s}, \mathbf{t} \mid \varphi)) \cdot S, B_1 \rangle, & \longrightarrow \quad \langle n, S, B_1 \rangle, \\ \langle m, \mathtt{answer}(q(\mathbf{x}, \mathbf{y})) \cdot T, B_2 \rangle & \qquad\quad \langle m, T[\mathbf{s}/\mathbf{x}], B_2 \rangle \end{array}$$

- if $B_1 \vdash \varphi$ and
- $B_2 \circ ((\mathbf{x} = f(\mathbf{s})) \wedge (\mathbf{y} = f(\mathbf{t})) \wedge f(\varphi)) \vdash \psi$, for some $f : \Delta \to \Sigma$.

Let us explain this transition. A rendezvous between an agent that asks the question $q(\mathbf{s}, \mathbf{t} \mid \varphi)$ and an agent that answers the question $q(\mathbf{x}, \mathbf{y})$, takes place in case the body ψ of the question follows from the answering agent's belief base B_2 in conjunction with the instantiation $\mathbf{x} = \mathbf{s}$ of the global parameters, the instantiation $\mathbf{y} = \mathbf{t}$ of the local parameters and the additional information φ, which is required to be a consequence of the asking agent's belief base B_1. As the actual parameters $\mathbf{s}$ and $\mathbf{t}$ and the information φ are expressed in the signature of the asking agent, they need to be translated into the interface signature of the question. This is achieved via a translation function f that maps the symbols in Δ into the symbols of the signature Σ, where it is required that the function f does *not* translate agent constants. Consequently, in the transition, the vocabulary of the answering agent m is *extended* with the agent constants that occur in $\mathbf{s}$, as these are not translated into the interface signature of the question. Note that possible inconsistencies need to be resolved by the update operator $\circ$.

Moreover, the transition reflects how the scope of the global formal parameters of a question is extended to the agent's subsequent program; that is, the formal parameters $\mathbf{x}$ of the question q can be referred to in the subsequent statement T of the answering agent. As mentioned before, this technique of extending the original scope of variables, is in the literature known as *scope extrusion*. In fact, these variables are replaced by the actual parameters $\mathbf{s}$.

Definition 12 (*Transition rule for external parallelism*) If $A_1 \cap A_2 = \emptyset$, we have:

$$\frac{A_2 \longrightarrow A_2'}{A_1 \cup A_2 \longrightarrow A_1 \cup A_2'}$$

The rule defines that the computation steps of a multi-agent system are derivable from the computation steps of its subsystems. That is, if a subsystem A_2 evolves itself to the system A_2', the resulting configuration of the overall multi-agent system $A_1 \cup A_2$ is given by the system $A_1 \cup A_2'$.

Example 13 (*Selling and buying books*)
Recall the multi-agent program $\mathcal{P} = (Booksellers, Bookbuyers)$ from the previous section. Consider the initial system $\{(n_1, \mathtt{IntegrateSellerAndBuyer}, true)\}$, which consists of an agent with name n_1 that has an empty information store $true$ and executes the procedure `IntegrateSellerAndBuyer` that is defined as follows:

$$
\begin{array}{l}
\texttt{IntegrateSellerAndBuyer} :- \\
\quad \texttt{loc}_y(\texttt{new}(y) \cdot \\
\qquad \texttt{loc}_z(\texttt{new}(z) \cdot \\
\qquad\quad \texttt{ask}(z, \mathit{NewSeller}(y, \epsilon \mid \mathit{true})))) \cdot \\
\qquad\quad \texttt{IntegrateSellerAndBuyer},
\end{array}
$$

where y is a variable of the sort $\mathit{Booksellers}$ and z a variable of the sort $\mathit{Bookbuyers}$. Thus, in this procedure, a seller y and a buyer z are integrated, after which the name of the seller is communicated to the buyer by means of an invocation of the buyer's question $\mathit{NewSeller}$.

Using the transition for agent integration this system evolves itself in one step to the multi-agent system:

$$
\begin{array}{l}
\{(n_1, \texttt{loc}_z(\texttt{new}(z) \cdot S), \mathit{true}), \\
(n_2, \texttt{HandleOffer}, \varphi)\},
\end{array}
$$

in which a selling agent n_2 has been integrated. The statement S abbreviates the remainder of the procedure `IntegrateSellerAndBuyer`, in which the variable y is substituted by the constant n_2.

Subsequently, the agent n_1 integrates an instance of the class $\mathit{Bookbuyers}$. Using the rules for integration and external parallelism, we derive the system:

$$
\begin{array}{l}
\{(n_1, \texttt{ask}(n_3, \mathit{NewSeller}(n_2, \epsilon \mid \mathit{true})) \cdot S', \mathit{true}), \\
(n_2, \texttt{HandleOffer}, \varphi), \\
(n_3, \texttt{BuyBook737}, \mathit{true})\},
\end{array}
$$

where S' abbreviates the remainder of the program of agent n_1.

After that, a rendezvous between the agent n_1 and the buying agent n_3 takes place through an invocation of the question $\mathit{NewSeller}$, which has the side-effect that the name of the selling agent n_2 is communicated. By means of the rules for rendezvous and parallel execution, this results in the following configuration of the program:

$$
\begin{array}{l}
\{(n_1, \texttt{IntegrateSellerAndBuyer}, \mathit{true}), \\
(n_2, \texttt{HandleOffer}, \varphi), \\
(n_3, \texttt{ask}(n_2, \mathit{NewBuyer}(n_3, \epsilon \mid \mathit{true})) \cdot (U_1 + U_2), \mathit{true})\},
\end{array}
$$

where U_1 and U_2 abbreviate the corresponding parts of the program of agent n_3.

In the next two steps of the program, a rendezvous between the buyer n_3 and the seller n_2 takes place through an invocation of the question $\mathit{NewBuyer}$. After that the information store of the seller is updated with the information $\mathit{Customer}(n_3)$. These two steps lead to the following multi-agent system:

$$
\begin{array}{l}
\{(n_1, \texttt{IntegrateSellerAndBuyer}, \mathit{true}), \\
(n_2, \texttt{HandleOffer}, \varphi \circ \mathit{Customer}(n_3)), \\
(n_3, (\texttt{ask}(n_2, \mathit{AcceptOffer}((n_3, b_{737}), \$22 \mid \mathit{true})) \cdot U_3), \mathit{true})\},
\end{array}
$$

where U_3 abbreviates the remainder of the program of agent n_3.

Finally, as we assume that φ contains the information $\mathit{novel}(b_{737})$, a rendezvous between the buyer n_3 and the seller n_2 takes place through an invocation of the question

$AcceptOffer$. This results in the following configuration:

$$\{(n_1, \texttt{IntegrateSellerAndBuyer}, true),$$
$$(n_2, \texttt{update}(Sold(b_{737}, n_3)) \cdot \texttt{HandleOffer}, \varphi \circ Customer(n_3),$$
$$(n_3, \texttt{update}(Bought(b_{737}, n_2)), true)\}.$$

Let us examine whether the requirements of the rendezvous are satisfied: the body of the question $AcceptOffer$ should be implied by the information store of the answering agent together with the information provided by the asking agent, modulo a translation into the signature of the question. This condition is the following:

$$\begin{gathered} Customer(n_3) \wedge Novel(b_{737}) \\ x = n_3 \wedge y = b_{737} \wedge z = f(\$22) \wedge \\ \vdash \\ Customer(x) \wedge \neg Defaulter(x) \wedge \neg\exists u(Sold(y, u)) \wedge \\ ((Novel(y) \wedge z \geq \text{€}20) \vee (Comic(y) \wedge z \geq \text{€}4)) \end{gathered}$$

where f denotes the translation function that maps dollars to euros using the equality $\$1 = \text{€}0.95$. Hence, $f(\$22)$ stands for $\text{€}21$. The condition holds as $\neg Defaulter(x)$ and $\neg\exists u(Sold(y, u))$ follow from the fact that $\vdash$ is a negation-as-failure operator.

6 Conclusions and Future Research

One of the benefits of following the structuring mechanisms of object-oriented programming is that it allows us to study how the Unified Modeling Language (UML) [7], which has become a significant software engineering tool, can be applied for the modeling of agent communication. Moreover an interesting topic of feature research is the study of the concept of *inheritance* in the context of agent communication, which denotes the reuse of code, and the concept of *subtyping*, which denotes the specialistion of behaviour. That is, agent classes can be organised into hierarchical classifications, which describe what constituents classes inherit from other classes. An important ingredient of this study is the development of a semantic characterisation of the subtyping relation, which describes under what circumstances agents from a particular class specialise the behaviour of agents from an other class. One of the conditions of this relation is for example the condition that agents of the subtype can be asked the same questions as the agents of the supertype and moreover, provide the same answers. Subtyping is a very significant concept from a software-engineering point of view, as in modifying existing multi-agent programs, it allows the local replacement of agents by agents from a subtype, without affecting the overall behaviour of the system. A subsequent step is the development of an equivalent *syntactic* characterisation of the subtyping relation, possibly based on a refinement calculus, which yields a method of formally proving subtype relations. An additional interesting issue of future research is to extend the framework with mechanisms that can be used by agents to *advertise* the questions they can be asked to answer.

References

1. P.H.M. America, J. de Bakker, J.N. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 194–208, St. Petersburg Beach, Florida, 1986.
2. G.R. Andrews. *Concurrent Programming, Principles and Practice*. The Benjamin Cummings Publishing Company, Inc., Redwood City, California, 1991.
3. F. Brazier, B. Dunin-Keplicz, N. Jennings, and J. Treur. Formal specification of multi-agent systems: a real-world case. In *Proceedings of International Conference on Multi-Agent Systems (ICMAS'95)*, pages 25–32. MIT Press, 1995.
4. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Systems of communicating agents. In Henri Prade, editor, *Proceedings of the 13th biennial European Conference on Artificial Intelligence (ECAI'98)*, pages 293–297. John Wiley & Sons, Ltd, 1998.
5. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Information-passing and belief revision in multi-agent systems. In J. P. M. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V, Proceedings of 5th International Workshop on Agent Theories, Architectures, and Languages (ATAL'98)*, volume 1555 of *Lecture Notes in Artificial Intelligence*, pages 29–45. Springer-Verlag, Heidelberg, 1999.
6. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Open multi-agent systems: Agent communication and integration. In N.R. Jennings and Y. Lespèrance, editors, *Intelligent Agents VI, Proceedings of 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL'99)*, volume 1757 of *Lecture Notes in Artificial Intelligence*, pages 218–232. Springer-Verlag, Heidelberg, 2000.
7. A. Evans, S. Kent, and B. Selic, editors. *The Unified Modeling Language (UML 2000)*, volume 1939 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2000.
8. P. Gärdenfors. *Knowledge in flux: Modelling the dynamics of epistemic states*. Bradford books, MIT, Cambridge, 1988.
9. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
10. Simone Marini, Maurizio Martelli, Viviana Mascardi, and Floriano Zini. Specification of heterogeneous agent architectures. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
11. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
12. Steven Shapiro and Yves Lespérance. Modeling multiagent systems with CASL — a feature interaction resolution application. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
13. D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents - Workshop on Agent Theories, Architectures, and Languages (ATAL'94)*, volume 890 of *Lecture Notes in Artificial Intelligence*, pages 386–401. Springer-Verlag, 1995.
14. M. Wooldridge and N. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.