Formal Specification of Agent-Object Oriented Programs

Francesco Pagliarecci, Luca Spalazzi
Dipartimento di Ingegneria Informatica, Gestionale e dell'Automazione
Università Politecnica delle Marche
Ancona, Italy
{pagliarecci, spalazzi}@diiga.univpm.it

M.-O. Stehr, C. L. Talcott
Computer Science Laboratory
SRI International,
Menlo Park, CA 94025-3493 USA
{stehr, clt}@csl.sri.com

ABSTRACT

This paper presents a methodology for the formal specification of agent-object oriented programs. *Agent-object oriented programming* is a programming paradigm that integrates both agent-oriented programming and object-oriented programming (e.g, see Jack, Jadex). Even if there are several formal specification frameworks and methodologies both for agent-oriented programs and for object-oriented programs, nothing exists for agent-object programming. In this paper, the rewriting logic language Maude has been used as a formal framework. This opens to us the possibility of using the wide-spectrum of formal modeling and reasoning supported by Maude: analyzing agent-object programs by means of execution, search, model checking, or theorem proving to verify properties of a given program such as goal satisfaction and plan termination.

KEYWORDS: Frameworks and Methodologies for Collaboration, Intelligent and Autonomous Agents in Collaboration.

1. INTRODUCTION

Agent-object oriented programming is an emergent programming paradigm that integrates both agent-oriented programming (e.g., the so called Belief-Desire-Intention (BDI) model [11]) and object-oriented programming. According to this paradigm, programming means defining the *mental attitudes* (i.e., beliefs, desires, and plans) that programs must exhibit (as in agent-oriented programming). Beliefs, desires, and plans can be programmed in terms of abstraction, encapsulation, inheritance, and polymorphism (as in object-oriented programming). Nowadays, this paradigm has been adopted in several examples of programming languages (e.g, see [6, 10, 9]) and methodologies (e.g, see [14, 1]).

This paper presents a methodology for the formal specification of agent-object oriented programs. Even if this programming paradigm inherits all the work about methodologies and formal frameworks done for agent-oriented programming (e.g., see [15, 2]) and object-oriented programming (e.g., see [12]), nothing exists specifically conceived for agent-object programming. Formal frameworks for agent-oriented programming are usually based on some sort of modal temporal logic, where beliefs, desires, and intentions are represented by operators. In agent-object programming, mental attitudes are complex structures with their own attributes and methods and thus they can be hardly represented by simple modal operators. Formal frameworks for object-oriented programming are usually based on process algebras, where classes are represented by algebraic structures. These structures can be easily exploited to represent mental attitudes in agent-object programming, as well. Nevertheless, in agent-object programming, we often need to represent temporal properties.

For all the above reasons, in this work we propose to use the rewriting logic language Maude. Maude [4] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications (from the formalization of mathematical structures to the modeling of biological systems). Its significance is threefold. First, it is based on a logic that can be used for the precise specification of a program semantics. Second, it is general in the sense that various paradigms, especially the agent- and object-oriented paradigms as well as concurrent behavior can be integrated by using Maude as a framework. Third, Maude is not only a language but also a system that supports light-weight symbolic analysis of specifications, an important feature for model validation and a first step on the path to formal verification.

The paper is structured as follows. In Section 2, we provide a brief introduction to Maude. Section 3 contains the example of agent-object program that is formalized in Section 4 using Maude. The properties to verify and the results are reported in Section 5. Finally, in Section 6 we give some conclusions.

2. A BRIEF INTRODUCION TO MAUDE

Maude has been successfully used as a logical framework [7], e.g. in the development of logics and theorem provers, and as a semantic framework, e.g in the formalization of programming language semantics [13].

Maude's logical foundation is membership equational logic [3] and rewriting logic [8], and hence it supports the specification of structured state transition systems over a rich class of algebraic specifications. Local state transitions are expressed as rewrite rules $L \Rightarrow R$ over an algebraic specification of the state space. Such rewrite rules can be conditional, written $L \Rightarrow R$ `if` C with C being a condition, and L, R and C are terms involving operations which again can be specified algebraically in membership equational logic. Since many specification, modeling, and programming paradigms can be expressed as rewriting modulo a suitable equational theory characterizing the state space, Maude is an excellent choice for a framework in which different paradigms ranging from functional programming to Petri nets can be integrated [8].

The Maude library provides basic syntax for specification of concurrent object-oriented programming via multi-set rewriting that we use in the present work. In a nutshell, the state of a system is modeled as a multiset of objects and messages. An object is of the form $< oid : cid \mid attrs >$, where oid and cid are the object and class identifiers, respectively, and $attrs$ is a set of attributes, i.e. labelled fields of the form $aid : val$, where val is the value associated with attribute aid. An object obj can perform internal state transitions using rewrite rules such as $obj \Rightarrow obj'$ `if` C, where obj' has the same identity as obj but with some of the attributes modified. Alternatively, an object obj can participate in communication using rewrite rules such as $obj\, msg \Rightarrow obj'\, msg'_1 \ldots msg'_n$ `if` C, where msg is a message consumed and $msg'_1 \ldots msg'_n$ is a multiset of messages produced in this atomic transition. More generally, obj' can be a multiset of objects in the above rules so that new objects can be created. Finally synchronization between objects can be represented by rules rewriting multiple objects and messages.

Maude specifications are organized in a hierarchy of modules. A module may contain sort declarations specifying the types of things to be represented; operator declarations (keyword `op` or `ops`) giving the sorts of the arguments and the value returned; equations (keyword `eq` or `ceq` if there is a condition) defining the operators; and rewrite rules (keyword `rl` or `crl`). Maude specifications are executable. Such specification can be used as formal prototypes of systems, and their behavior and properties can be studied using symbolic execution, symbolic state space exploration, and model checking techniques which are all supported by the Maude rewriting engine [4]. This will be illustrated in Section 5.

3. RUNNING EXAMPLE

Agent-object oriented programming means to establish what are the interactions of the program with the other programs, what are its beliefs, its desires, and its plans. Beliefs, desires, and plans should be defined in an abstract way, encapsulating their attributes and methods, exploiting inheritance relations among them, and exploiting polymorphism. In the rest of the paper, we consider the following example. Let us suppose we have to program an agent that must search for a partner (another agent) capable of a given service. This example is partially described in Figure 3.

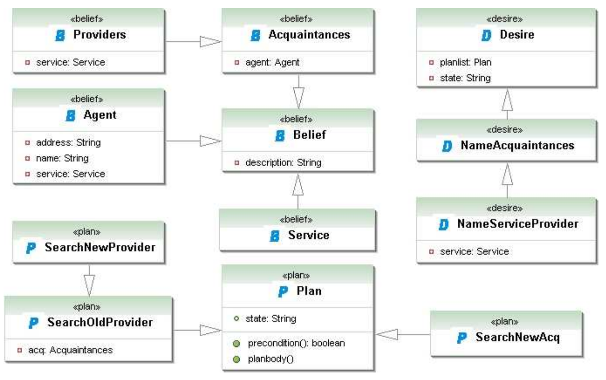

Figure 1: The UML class diagram of the running example

Notice that, we describe the program using UML class diagram (with appropriate stereotypes). We do not choose a specific programming language (as Jack, Jadex, or Alan) since the proposed framework is independent on the programming language. The UML diagram can be easily translated in the preferred language.

Beliefs can be modeled using object-oriented programming principles. This means that, for each belief, we encapsulate all the attributes that describe it and all the methods that manipulate it. Each belief can be modified by any action (e.g., a reception of a message) that satisfies its encapsulation constraints. The beliefs of the example consist in beliefs about the agents that the program is able to contact (the belief `Agent`) and their corresponding services (the belief `Service`). We also have `Acquaintances` and `Providers` that are arrays of all the known agents and all the agents providing a given service, respectively.

Desires are objects with their attributes and methods, as well [1]. When a new desire is asserted, its state becomes `ready` (see the attribute `state`), an appropriate plan (see the attribute `planlist`) is selected and then executed. Desires that have been requested to be satisfied become intentions (its state becomes `running`). Finally, the desire state can be `succeeded` or `failed` depending on the final result of the last plan execution. In the example, we have reported only two desires: `NameAcquaintances` and `NameServiceProvider`. The former represents the goal of finding new acquantancies, the latter the goal of finding the providers of a given service. Notice that, each desire declares what plan types to try.

Plans are objects, as well. For each plan, the programmer must define at least two methods: `precondition` and `planbody`. `precondition` is a Boolean method. [2] When a plan is selected and instantiated, the precondition is executed. When it returns true, the plan is executed. `planbody` is the method that contains the set of actions to be executed in order to satisfy a given desire. In Figure 3, we have reported three plans. `SearchNewAcq` is a plan to satisfy `NameAcquaintances`. It sends a message to the *facilitator* agent `df`[3] and waits for a list of agents as answer. `SearchOldProvider` is a plan that searches the array `acq` for an agent providing a given service. `SearchNewProvider` is a plan that first asks a facilitator for information about new agents, saves these information in the array `acq`, and then applies the `planbody` of `SearchOldProvider` to perform a search in the array `acq`.

4. MODELING AN AGENT-OBJECT PROGRAM IN MAUDE

The formal specification of an agent-object program proceeds in three steps: *first*, we have to represent the *operational semantics* of an agent-object program (how the interpreter works); *second*, we have to formalize the specific *program* (e.g., the program modeled in Figure 3); *third*, we have to formalize the *specifications* and verify them. Here we describe the first two steps, the third one is reported in Section 5. To specify the *operational semantics*, we first explain what is the *global state* of a program, then we discuss the *behavior* of the interpreter.

The **global state** of a program must contain all the elements of the program (i.e., beliefs, desires, plans, intentions) and the operations that must be executed by the interpreter. Beliefs, desires, and plans are modeled as objects with their own attributes; operations are modeled as messages. Therefore, the global state of a program is modeled as a multiset of objects and messages. Thus a belief is an object of the form `< Bid : BX | ATTS >`, where `Bid` is a belief object identifier, `BX` is a belief class identifier, and `ATTS` stands for the the set of belief attributes. As a consequence, as reported in Figure 4, we need to define a subsort of `Oid` (`Oid` is predefined in Maude and represents the sort of all the object identifiers) called `BeliefID` and a subsort of `Cid` (`Cid` is predefined in Maude and represents the sort of all the class identifiers) called `Belief`. Letting `BX` be a variable of sort `Belief` allows us to model the inheritance mechanism, (we have an example in Figure 4). Similarly, plans and desires are objects with the following structures: `< Pid : PX | planState : ..., precondition : ..., planBody : ..., ATTS >` and `< Did : DX | desireState : ..., planList : ..., ATTS >`, respectively (see Figure 4). The only difference is the set of attributes of `Plan` and `Desire`. A plan has at least three predefined attributes: `planState`, `precondition`, and `planBody`; a desire has at least two predefined attributes: `desireState` and `planList`. Both `planState` and `desireState` can assume one of the following values: `Ready`, `Running`, `Wait`, `Succeeded`, `Failed`. The two attributes `precondition` and `planBody` model the two predefined methods of a plan. The attribute `planList` is a list of plans that must be tried in order to satisfy the desire. These plans are ordered in a queue according to an order established by the programmer. The definition of plans and desires in Figure 4 contains rewrite rules. Rules such as `start`, `success`, `failure` in modules `DESIRE` and `PLAN` produce a state change. The rule `planbody` of `PLAN` starts the execution of the method `planbody` of a given plan instance. These rules are activated by appropriate messages (see Figure 4). The conditional rule `start` of `PLAN` describes a transition of the plan instance `Pid` from a state where `planState` is `Ready` to a state where `planState` is `Running`. This rule is activated by the message `start(Pid)` and produces a new message: `planbody(Pid)`. This rule can be applied only if the `precondition P` is true. The message `planbody(Pid)` activates a chain of rules that models the behavior of the plan, in other words the set of statements that must be executed in order to satisfy the intention. The activation of any kind of (belief, desire, or plan) method is modeled in a similar way. Concerning intentions, from an intuitive point of view, the selection of a desire produces the instantiation of a new intention.

Once we have defined the elements of a program, the next step is the definition of lists of plan identifiers (`PlanIDList`) and lists of intentions (`IntentionBase`). For instance, let us consider `PlanIDList` (see Figure 4). The relation `subsort PlanID < PlanIDList` states that each `PlanID` is also a `PlanIDList` (we may have `PlanIDList` composed by only one `PlanID`). The defini-

[1] In traditional agent-oriented programming, desires are represented as logical formulae

[2] In traditional agent-oriented programming, preconditions are logical formulae

[3] A facilitator is an agent that manages an agent directory.

```
mod BELIEF is
 including CONFIGURATION .
 including STRING .
 sort Belief BeliefID .
 subsort Belief < Cid .
 subsort BeliefID < Oid .
 op belief : -> Belief [ctor] .
 op description :_ :
       String -> Attribute [ctor] .
 var BX : Belief .
 var Bid : BeliefID .
 var ATTS : AttributeSet .
endm

mod DESIRE is
 including CONFIGURATION .
 protecting PLAN-ID-LIST .
 sorts Desire DesireID DesireState .
 subsort Desire < Cid .
 subsort DesireID < Oid .
 op desire : -> Desire [ctor] .
 op planList :_ :
      PlanIDList -> Attribute [ctor] .
 op desireState :_ :
      DesireState -> Attribute [ctor] .
 op Ready : -> DesireState [ctor] .
 op Running : -> DesireState [ctor] .
 op Wait : -> DesireState [ctor] .
 op Succeeded : -> DesireState [ctor] .
 op Failed : -> DesireState [ctor] .
 ops start suspended resumed
     success failure :
      DesireID -> Msg [ctor] .
 ...
endm

mod INTENTION is
 protecting DESIRE .
 sort Intention .
 op i : DesireID PlanIDList ->
           Intention [ctor] .
endm

mod PLAN is
 including CONFIGURATION .
 sorts Plan PlanID PlanState AllSorts .
 subsort Plan < Cid .
 subsort PlanID < Oid .
 op plan : -> Plan [ctor] .
 op planState :_ : PlanState ->
        Attribute [ctor] .
 op precondition :_ : Bool -> Attribute .
 op planbody :_ : AllSorts -> Attribute .
 op Ready : -> PlanState [ctor] .
 op Running : -> PlanState [ctor] .
 op Wait : -> PlanState [ctor] .
 op Succeeded : -> PlanState [ctor] .
 op Failed : -> PlanState [ctor] .
 ops start suspended resumed
     success failure planbody : PlanID ->
        Msg [ctor] .
 var Pid : PlanID . var PX : Plan .
 var P : Object .   var Pr : Bool .
 vars ATTS : AttributeSet .
 crl [start] :
    < Pid : PX | planState:Ready,
      precondition:Pr > start(Pid) =>
    < Pid : PX | planState:Running,
      precondition:Pr > planbody(Pid)
    if Pr = true .
 rl [planbody] :
    < Pid : PX | planState : Running,
      ATTS > planbody (Pid) =>
    < Pid : PX | planState : Wait, ATTS > .
endm

mod PLAN-ID-LIST is
 including PLAN .
 sort PlanIDList .
 subsort PlanID < PlanIDList .
 op noPlan : -> PlanIDList [ctor] .
 op _ , _ : PlanIDList PlanIDList ->
     PlanIDList [ctor assoc id: noPlan] .
endm

mod INTENTION-BASE is
 protecting INTENTION .
 sort IntentionBase .
 subsort Intention < IntentionBase .
 op noneI : -> IntentionBase [ctor] .
 op _ , _ : IntentionBase IntentionBase ->
     IntentionBase [ctor assoc id: noneI] .
endm
```

Figure 2: The definition in Maude of Belief, Desires, and Plans

tion `op noPlan : -> PlanIDList` states that `noPlan` is a constant plan list: the empty list. Moreover, the concatenation between plan lists is denoted by the operator `_ , _` and it is defined as follows: `op _ , _ : PlanIDList PlanIDList -> PlanIDList`. The annotation `[ctor assoc id: noPlan]` says that concatenation is associative with identity `noPlan`. The addition of a plan identifier `Pid` to a plan list `PidL` is simply be expressed by the term `Pid, PidL`. Lists of `Intention`s are modeled in a similar way. At this point, we can finally define the notion of `State` of a program. As defined in the module `INTERPRETER`, (the operator declaration for `_-_-_`) the sort `State` is a triple consisting of a `Configuration`, an `IntentionBase`, and a `MsgList`. The sort `Configuration` is a predefined sort that denotes a multiset of objects (the sort `Object`) and messages (the sort `Msg`). We use a configuration to denote the multiset of beliefs, desires, plans, and messages sent to them. These messages represent the operations that must be performed by beliefs, desires, and plans. The sort `MsgList` is a list of messages (the sort `Msg`) sent to the interpreter. They represent the operations that must be performed by the interpreter.

The **behavior** of the interpreter is based on [5] and consists in a set of operations for managing (adding, removing) beliefs, desires, intentions, and plans; for selecting desires and plans; for executing methods. Their actions are modeled by operators `[ctor]` (e.g., `newDesire`, `selectDesire`, `newIntention`, `selectPlan` in Figure 4) that map program elements to messages (sort `Msg`). Messages are used to build the state and thus to select the appropriate transition rule. More in detail, let us consider the example of adding a new belief. The activation of this operation is modeled by a message `newBelief(< Bid : BX | ATTS >)`. The rule `newBelief` (see Figure 4) models the state transition associated to this operation. Namely, it is a transition from a state where the configuration is `CC` and the operation list contains the message `newBelief(< Bid : BX | ATTS >)`; to a state where the configuration

```
mod INTERPRETER is
 protecting BELIEF .
 protecting DESIRE .
 protecting INTENTION-BASE .

 sorts State MsgList .
 subsort Msg < MsgList .

 op nomsg : -> MsgList .
 op _;_ : MsgList MsgList -> MsgList
         [ctor assoc comm id: nomsg] .
 op _-_-_ : Configuration IntentionBase
           MsgList -> State [ctor] .
 op newBelief : Object -> Msg [ctor] .
 op delBelief : BeliefID -> Msg [ctor] .
 op newDesire : Object -> Msg [ctor] .
 ops successD failedD :
       DesireID -> Msg [ctor] .
 op selectDesire : Object ->
       Msg [ctor] .
 ops newIntention delIntention :
       Intention -> Msg [ctor] .
 op selectPlan : PlanIDList ->
       Msg [ctor] .

 vars B D P P' : Object .
 var I : Intention .
 vars IB IB' : IntentionBase .
 var PidL PidL' : PlanIDList .
 var CC : Configuration .
 var Bid : BeliefID . var DX : Desire .
 vars Pid Pid' : PlanID .
 var BX : Belief . var Did : DesireID .
 var PX : Plan .   var Pr : Bool .
 var ATTS : AttributeSet .
 var M : Msg .     var ML : MsgList .
 rl [newBelief] : (CC - IB -
    (newBelief (B) ; ML)) =>
    (CC B - IB - ML) .
 crl [delBelief] : (CC B - IB -
     (delBelief (Bid) ; ML)) =>
     (CC - IB - ML)
     if < Bid : BX | ATTS > := B .
 crl [newDesire] : (CC - IB -
         (newDesire (D) ; ML)) =>
    (CC D - IB -
         (selectDesire (Did) ; ML))
    if < Did : DX | desireState : Ready,
         ATTS > := D .
crl [selectDesire] : (CC D - IB -
        (selectDesire (D) ; ML)) =>
   (CC D start(Did) - IB -
        newIntention(i(Did, PidL)) ; ML)
   if < Did : DX | desireState : Ready,
        planList : PidL > := D .
crl [newIntention] : (CC - IB -
        (newIntention (I) ; ML)) =>
   (CC - addIntention(I, IB) -
        (selectPlan (PidL) ; ML))
   if i(Did, PidL) := I .
crl [selectPlan] : (CC P - I , IB -
        (selectPlan(Pid, PidL') ; ML)) =>
   (CC P start(Pid) - I , IB - ML)
   if i(Did, (Pid , PidL')) := I
   /\ < Pid : PX | planState : Ready,
        ATTS > := P .
crl [successPlan] : (CC P success(Pid) -
        I , IB - ML) =>
   (CC P - I , IB -
        delIntention (I) ; ML)
   if i(Did, (Pid , PidL)) := I
   /\ < Pid : PX | planState : Succeeded,
        ATTS > := P .
crl [failedPlan] : (CC P' failure(Pid) -
        i(Did, (Pid , Pid' , PidL)) ,
        IB - ML) =>
   (CC P' start(Pid') -
        i(Did, (Pid' , PidL)) , IB - ML)
   if < Pid' : PX | planState : Ready,
        ATTS > := P' .
   /\ < Pid : PX | planState : Failed,
        ATTS > := P .
crl [delIntention] : (CC - I , IB -
        delIntention (I) ; ML) =>
   (CC - IB - successD (Did) ; ML)
   if i(Did, (Pid , PidL')) := I .
rl [successDesire] :
   (CC D - IB - successD (Did) ; ML) =>
   (D CC success(Did) - IB - ML) .
crl [failedLastPlan] : (CC P failure(Pid) -
        i(Did, Pid) , IB - ML) =>
   (CC - i(Did, Pid) , IB -
        failedD (Did) ; ML)
   if < Pid : PX | planState : Ready,
        ATTS > := P .
rl [failedDesire] :
   (CC D - IB - failedD (Did) ; ML) =>
   (D CC failure(Did) - IB - ML) .
endm
```

Figure 3: The definition in Maude of the Interpreter

is `CC < Bid : BX | ATTS >` (i.e., the belief `< Bid : BX | ATTS >` has been added to `CC`) and the message `newBelief(< Bid : BX | ATTS >)` has been removed from the operation list. Removing a belief, adding or removing a desire, an intention, or a plan are modeled in a similar way. From an intuitive point of view, the selection of a desire produces the instantiation of a new intention. Each intention has a list of plans. They are tried one by one until a plan able to satisfy the intention is found. Therefore, the intention is modeled by a pair `i(Did, PidL)`, where `Did` is the identifier of the selected desire and `PidL` is the list of plan identifiers. This behavior is described by rules `selectDesire`, `newIntention`, and `selectPlan` of Figure 4. The conditional rule `selectDesire` models a transition from a state where the operation list contains the message `selectDesire(< Did : DX | ...>)` to a state that contains the request of a new intention (the message `newIntention(I)` sent to the interpreter) and the request of changing the status of the desire (the message `start(Did)` sent to `Did`). The message `start(Did)` activates the rewrite rule `start` of the module `DESIRE` (see Figure 4). The message `newIntention(I)` activates the rule `newIntention` that models a transition from a state where the intention base is `IB` and interpreter has received the message `newIntention(I)` to a state where `I` has been added to `IB`. The rule `selectPlan` models a transition from a state that contains the message `selectPlan(Pid , PidL')` to a state that contains the message `start(Pid)` for the plan with identifier is `Pid`. The message `start(Pid)` activates the conditional rule `start` of the module `PLAN`. The success and the failure of a plan is represented by rules: `successPlan`, `failedPlan`, `delIntention`, `successDesire`, `failedLastPlan`, and `failedDesire`. The rule `successPlan` is activated when a plan succeeds (the plan has received the message `success(Pid)`) and has as effect the deletion of the corresponding intention (`delIntention(I)` is sent to the interpreter). On the other hand, when a plan fails (the rule `failedPlan` is activated) the next plan in the intention is tried. When all the plans have been tried (the rule `failedLastPlan` is activated), the intention is removed and a message `failedD(Did)` is sent to the interpreter.

Now we are ready to model the **program** reported in Section 3. First of all, this means we have to formalize the inheritance mechanism of beliefs, desires, and plans. For instance, `subsort SearchOldProvider < Plan` in the module `SEARCH-OLD-PROVIDER` states that `SearchOldProvider` is a subclass of `Plan`. In other words, each object of type `SearchOldProvider` is a plan and, thus, it inherits the structure of a plan. Furthermore, the rules for a plan can be also applied to objects of type `SearchOldProvider`. The sentence

```
subsort SearchNewProvider < SearchOldProvider
```

in the module `SEARCH-NEW-PROVIDER` models the fact that `SearchNewProvider` is a plan type that inherits the structure and the rules of `SearchOldProvider`. In a similar way, we model the inheritance mechanism for beliefs and desires.

Second, we have to formalize the encapsulation mechanism. This means we have to formalize the specific attributes and methods of each belief, desire, or plan. Similarly to what we have already seen, attributes are represented by operators of the type `op ... : ... -> Attribute [ctor]`, methods by equations (i.e., `eq`) and transition rules (i.e., `rl`). Therefore, concerning our example, the formalization of the attributes for beliefs, desires, and plans of Figure 3 is quite straightforward, and the result is reported in Figure 4. Notice the attribute `planList`, that is defined as an operator whose domain is the list of all the plans that can be tried to satisfy the desire itself. Concerning the methods, we have exemplified the formalization of the `planbody` of `SEARCH-OLD-PROVIDER` (see Figure 4). Notice the combined use of equations and rules. For instance, the `planbody` of `SEARCH-OLD-PROVIDER` is composed by a rule (i.e., `rl [planbody] ...`) that models the transition from a state where we have a running plan (`< Pid ... planState : Running ...>`) and a method activation (`planbody (Pid)`) to a state where we have the instantiation of a new belief `Providers` that contains a list of acquantances built by means of the function `op makeLP : Providers Service Acquaintances -> Providers`. This function is defined by a set of equations. The equations represent the fact that a list of providers is composed by a set of acquantancies that provide the requested service.

5. FORMAL SPECIFICATION

Once we have specified the different aspects of an agents behavior, as explained above, we can take advantage of Maude's support for wide spectrum formal analysis to analyze different initial agent configurations. We describe several examples to illustrate the basic ideas. We use Maude's rewrite strategy to see the result of a possible execution. We use search to determine whether, given an intended desire, the system can reach a state where `desireState` of the given desire is `Succeeded` and we use model-checking to see if all executions lead to such a state.

First, we define the elements of initial states of interest and combine them to form two initial states (`IS1` and `IS2`), the first contains a single desire to be satisfied (of type `NameServiceProvider`) and a message `newInternalD(D-011)` to initiate the satisfaction process, and the second initial state contains an additional desire of the same type with its initiator message. In addition there are plan objects corresponding to the plans to be tried in order to satisfy the desires. Formally, we declare object identifier and object constants

```
ops B-001 B-002 B-003 B-004 B-005 B-009 : -> BeliefID .
ops D-001 D-001x : -> DesireID .
ops P-001 P-002 : -> PlanID .
ops nsp1 nsp1x s2 p1 p2 aq a1 a2 a3 pv1 : -> Object .
```

and define the object constants using the following equations.

```
eq nsp1 = <D-001 : nameServiceProvider |
                   desireState : Ready,
                   planIDList : (P-001, P-002),
                   wantedService : B-002> .
eq nsp1x = <D-001x : nameServiceProvider |
                     desireState : Ready,
                     planIDList : (P-001, P-002),
                     wantedService : B-002> .
eq s2 = <B-002 : service | description : "Servizio 2"> .
eq pv1 = <B-009 : providers | description:"ST9",
                 agents:nil, serviceID:B-002> .
eq p1 = <P-001 : searchOldProvider | planState : Ready,
                 precondition : Pr,
                 service : B-002,
                 acq : B-001, agentList : a1> .
eq p2 = <P-002 : searchNewProvider | planState : Ready,
                 precondition : true, service : B-002,
                 acq : B-001, agentList : none,
                 NADesire : D-002> .
eq aq = <B-001 : acquaintances |
                 agents : (B-003 B-004 B-005)> .
eq a1 = <B-003 : agent | name : "N1", address : "AD1",
                 services : (B-002 B-003)> .
...
```

The initial states are defined in terms of configurations `CC1` and `CC2` and empty intention and message sets.

```
ops CC1 CC2 : -> Configuration .
eq CC1 = nsp1 s2 pv1 p1 p2 aq  a1 a2 a3 newInternalD(D-001)
eq CC1 =  CC1 nsp1x newInternalD(D-001x)
```

```
mod ACQUAINTANCES is
 including BELIEF .
 protecting OID-LIST .
 sort Acquaintances .
 subsorts Acquaintances < Belief .
 op acquaintances : -> Acquaintances [ctor] .
 op agents :_ : List{Oid} -> Attribute [ctor] .
endm

mod PROVIDERS is
 protecting SERVICE .
 including ACQUAINTANCES .
 sort Providers .
 subsorts Providers < Acquaintances .
 op providers : -> Providers [ctor] .
 op serviceSort :_ : Service -> Attribute [ctor] .
 op newProvider : List{Oid} -> Msg [ctor] .
 vars Bid1 Bid2 : BeliefID .
 var ST : String .
 var OidLS : List{Oid} .
 rl [new] : < Bid1 : providers | description : ST,
      agents : nil, serviceID : Bid2 >
      newProvider(OidLS) =>
    < Bid1 : providers | description : ST,
      agents : OidLS, serviceID : Bid2 > .
endm

mod AGENT is
 including BELIEF .
 protecting SERVICE OID-LIST .
 sorts Agent .   subsort Agent < Belief .
 op agent : -> Agent [ctor] .
 op name :_ : String -> Attribute [ctor] .
 op address :_ : String -> Attribute [ctor] .
 op services :_ : List{Oid} -> Attribute [ctor] .
endm

mod SERVICE is
 including BELIEF .
 sort Service .  subsort Service < Belief .
 op service : -> Service [ctor] .
endm

mod NAME-SERVICE-PROVIDER is
 including DESIRE .
 protecting BELIEF .
 sorts NameServiceProvider .
 subsort NameServiceProvider < Desire .
 op nameServiceProvider : -> NameServiceProvider .
 op wantedService :_ : BeliefID -> Attribute .
andm

mod SEARCH-OLD-PROVIDER is
 including PLAN .
 protecting AGENT PROVIDERS .
 sorts SearchOldProvider .
 subsort SearchOldProvider < Plan .
 op searchOldProvider : -> SearchOldProvider [ctor] .
 op service :_ : BeliefID -> Attribute [ctor] .
 op acq :_ : BeliefID -> Attribute [ctor] .
 op agentList :_ : Configuration -> Attribute [ctor] .
 op serviceSort :_ : BeliefID -> Attribute [ctor] .
 op resetAL : PlanID -> Msg [ctor] .
 op makeList : Object PlanID -> Msg [ctor] .
 op makeList4Provider : PlanID -> Msg [ctor] .
 op checkService : Object BeliefID -> Bool .
 op makeLP : List{Oid} BeliefID Configuration -> List{Oid} .
```

```
 vars Bid1 Bid2 Bid3 : BeliefID .
 vars AG SV AQ : Object .
 var C : Configuration .
 var DS : String .
 var LS1 LS2 LS3 : List{Oid} .
 var ATTS ATTS’ ATTS’’ : AttributeSet .
 var Pid : PlanID .

 ceq checkService (AG, Bid2) = true
    if < Bid1 : agent | services : LS1, ATTS > := AG
    /\ occurs (Bid2 , LS1) .
 eq checkService (AG, Bid2) = false [owise] .
 ceq makeLP (LS1, Bid2, AG C) =
    makeLP ((Bid3 LS1), Bid2, C)
    if < Bid3 : agent | services : LS3, ATTS > := AG
    /\ checkService (AG, Bid2) .
 ceq makeLP (LS1, Bid2, AG C) =
    makeLP (LS1, Bid2, C)
    if < Bid3 : agent | services : LS3, ATTS > := AG
    /\ not (checkService (AG, Bid2)) .
 eq makeLP (LS1, Bid2, none) = LS1 .

 rl [planbody] :
    <Pid:searchOldProvider | planState:Running ,
     acq:Bid1, ATTS> planbody (Pid) =>
    <Pid:searchOldProvider | planState:Running,
     acq:Bid1, ATTS> resetAL (Pid) .
 crl [resetAL] :
    <Pid:searchOldProvider | planState:Running ,
     acq:Bid1, agentList:C, ATTS>
    resetAL (Pid) =>
    <Pid:searchOldProvider | planState:Running,
     acq:Bid1, agentList:none, ATTS>
    makeList (AQ, Pid)
    if AQ := < Bid1 : acquaintances | ATTS > .
 crl [makeList] :
    <Pid:searchOldProvider | planState:Running,
     acq:Bid1, agentList:C, ATTS>
    < Bid2 : agent | ATTS’’ >
    makeList (< Bid1 : acquaintances |
      agents : (Bid2 LS1), ATTS>, Pid) =>
    <Pid:searchOldProvider | planState:Running,
     acq:Bid1, agentList:(AG C), ATTS>
    < Bid2 : agent | ATTS’’ >
    makeList (< Bid1 : acquaintances |
      agents : LS1, ATTS >, Pid)
    if AG := < Bid2 : agent | ATTS > /\ LS1 =/= nil .
 crl [makeList] :
    <Pid:searchOldProvider | planState:Running,
     acq:Bid1, agentList:C, ATTS>
    < Bid2 : agent | ATTS’’ >
    makeList (< Bid1 : acquaintances |
      agents : (Bid2 LS1), ATTS>, Pid) =>
    <Pid:searchOldProvider | planState:Running,
     acq:Bid1, agentList:(AG C), ATTS>
    < Bid2 : agent | ATTS’’ > makeList4Provider (Pid)
    if AG := < Bid2 : agent | ATTS > /\ LS1 == nil .
 rl [makeList4Provider] :
    < Pid : SOPX | planState : Running, service : Bid1,
      agentList : C, ATTS >
    makeList4Provider (Pid) =>
    < Pid : SOPX | planState : Succeeded, service : Bid1,
      agentList : C, ATTS >
    newProvider ( makeLP (nil, Bid1, C)) success(Pid) .
endm
```

Figure 4: The definition in Maude of the program of Section 3

```
ops IS1 IS2 : -> State .
eq IS1 = CC1 - noneI - nomsg .
eq IS2 = CC2 - noneI - nomsg .
```

As the first analysis we ask Maude to **rewrite** `IS1` and examine the resulting final state. (We show only objects with changed attributes.)

```
Maude> rew IS1 .
result State: (
<B-009 : providers | description:"ST9",
         agents:(B-005 B-003), serviceID:B-002>
<D-001 : nameServiceProvider | desireState : Succeeded,
 planIDList : (P-001,P-002),wantedService : B-002>
<P-001 : searchOldProvider | planState : Succeeded,
         precondition : true, service : B-002,acq : B-001,
         agentList : ... >
                ... ) - noneI - nomsg
```

We see that the plan `P-001` and the desire `D-001` have reached success states. If we rewrite `IS2` Maude picks an execution in which desire `D-001x` succeeds but `D-001` does not. We then use **search** to see if a state in which `D-001` succeeds can be reached from `IS2`.

```
search IS2 =>! C:Configuration
  < D-001 : nameServiceProvider | desireState : Succeeded,
    atts:AttributeSet > - II:IntentionBase - ml:MsgList .
```

The search succeeds and Maude returns the search state identifier, and values of the pattern variables `C:Configuration`, `atts:AttributeSet`, `II:IntentionBase`, `ml:MsgList`. We can get information about how the state was reached by asking `Maude> show path labels`. This results in the list of labels of rewrite rules applied to reach the named state.

```
newInternalD, newInternalD, selectDesire, selectDesire,
newIntention, selectPlan, (start)3, planbody, resetAL,
(makeList)4, makeList4Provider, successPlan, delIntention,
newIntention, successDesire, success, new
```

We can use **model checking** to get similar information. For example, we can ask whether in any execution of `IS2` in which `D-001` is started (there is an intention `i(D-001, (P-001, pidl))` and the `planState` of `P-001` is `Running`) then it completes (the computation eventually reaches a state in which the value of the `desireState` attribute is `Succeeded` or `Failed`). This is done by model checking the temporal formula `started(D-001,P-001) => <> decide(D-001)`. (The formala $P \Rightarrow <> Q$ says that in any state of an execution if P holds then in that state or some later state Q holds.) If the property fails to hold the model-checker returns a counter-example from which we can extract the states visited and the rules applied. In our example situation, the rule list found by the model-checker is equivalent to that found by search. The problem is that plan objects are not reusable. This could be a design decision, in which case the initial state is badly defined, or it could be a missing rule for plan behavior, and that can be fixed. As a final example we can check if a desire such as `D-001` with a given plan list `(P-001,P-002)` is in progress, `running(D-001,(P-001,P-002))`, then it continuos running until one plan succeeds or all fail, `plansTried((P-001,P-002))`. For the initial state above the model checker confirms that this property holds.

```
red modelCheck(IS2, running(D-001,(P-001,P-002))
     => (running(D-001,(P-001,P-002))
          W plansTried((P-001,P-002)))) .
result Bool: true
```

6. CONCLUSIONS

Formal specifications are an extremely important part of programming. This paper presents the formal specifications of key concepts of agent-object programming, an emerging programming paradigm. The main contributions are providing executable specifications, and combining object-oriented and agent-oriented features in one framework. We use the rewriting logic language Maude to provide a formal definition of the notion of beliefs, desires, plans, and their behavior. Furthermore, we have shown how an agent-object program can be represented in Maude. We then showed how the specification can be used to analyze the program using execution, search, and model-checking (Section 5). This is only a first step towards formal verification of agent-object programs. Next steps include developing a mapping from a specific agent-object programming language and use this to reason about specific programs. Another step is to codify properties of interest and specify them in Maude to aid program developers in their analysis tasks. Substantial case studies need to be carried out.

REFERENCES

[1] Agent UML Web Site, `http://www.auml.org`.

[2] Benerecetti, M., F. Giunchiglia, and L. Serafini, “Model checking multiagent systems,” *Journal of Logic and Computation*, vol. 8, no. 3, 1998, pp. 401–423.

[3] Bouhoula, A., J.-P. Jouannaud, and J. Meseguer, “Specification and proof in membership equational logic,” *Theoretical Computer Science*, vol. 236, 2000, pp. 35–132.

[4] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, MAUDE 2.0 MANUAL, http://maude.cs.uiuc.edu, 2003.

[5] Georgeff, M. P., and A. L. Lansky, ”Reactive Reasoning and Planning”, AAAI, 1987.

[6] Howden, N., R. Rönnquist, A. Hodgson, and A. Lucas, “Jack^{TM} - summary of an agent infrastructure,” 5th International Conference on Autonomous Agents, Montreal, Canada, 2001.

[7] Martí-Oliet, N., and J. Meseguer, “Rewriting logic as a logical and semantic framework,” HANDBOOK OF PHILOSOPHICAL LOGIC, Kluwer Academic Publishers.

[8] Meseguer, J., “Conditional rewriting logic as a unified model of concurrency,” *Theoretical Computer Science*, vol. 96, 1992, pp. 73–155.

[9] Pagliarecci, F., L. Spalazzi, and G. Capuzzi, “Formal Definition of an Agent-Object Pogramming Language” The 2006 International Symposium on Collaborative Technologies and Systems (CTS 2006), IEEE Computer Society Press, Las Vegas (USA), 2006, pp. 298–305.

[10] Pokahr, A., L. Braubach, and W. Lamersdorf, “Jadex: Implementing a BDI-Infrastructure for JADE Agents,” *EXP - In Search of Innovation* Telecom Italia Lab, Vol. 3, No. 3, 2003, pp. 76–85.

[11] Rao, A. S., and M. P. Georgeff, “Modeling rational agents within a BDI architecture,” KRR’91.

[12] Smith, G., “The Object-Z Specification Language,” in *Advances in Formal Methods*, Kluwer Academic Publisher, 2000.

[13] Stehr, M.-O., and C. Talcott, “Plan in Maude Specifying an Active Network Programming Language,” *Electronic Notes in Theoretical Computer Science*, vol. 71, 2002.

[14] Wagner, G., “A uml profile for agent-oriented modeling,” Third International Workshop on AgentOriented Software Engineering, Bologna (Italy), 2002.

[15] Wooldridge, M., REASONING ABOUT RATIONAL AGENTS, The MIT Press, Cambridge Massachusetts, 2000.