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.

I. 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 II, we provide a brief introduction to Maude. Section III contains the example of agent-object program that is formalized in Section IV using Maude. The properties to verify and the results are reported in Section V. Finally, in Section VI we give some conclusions.

II. A BRIEF INTRODUCTION 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 \( L \Rightarrow R \text{ if } C \) 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 \(< oid: cid | attrs >\), 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 \(obj msg \Rightarrow obj' msg'\) if \(C\), where \(msg\) is a message consumed and \(msg'\) 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 V.

III. 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 1.

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. 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 acquaintances, 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. 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 1, we have reported three plans. SearchNewAcq is a plan to satisfy NameAcquaintances. It sends a message to the facilitator agent df 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.

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.
IV. 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 1); third, we have to formalize the specifications and verify them. Here we describe the first two steps, the third one is reported in Section V. 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 2, 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 2). 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 2 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 2). 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 2). The relation subset PlanID < PlanIDList states that each PlanID is also a PlanIDList (we may have PlanIDList composed by only one PlanID). The definition 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 Intentions 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 3) 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 3) 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 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 3. The conditional rule selectDesire models a transition from a state where the operation list contains the message selectDesire($< Did : DX \mid \ldots>$) 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 2). 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 III. 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 1 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 (i.e., lx ... 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 acquaintances 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 acquaintances that provide the requested service.

V. 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 deter-
mine 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(0-011) 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

cr

vars Bid1 Bid2 Bid3 : BeliefID 
vars AG SV Aq : Object .
var C : Configuration .
var D : String .
vars LS1 LS2 LS3 : List{Old} .
var ATS ATS1 ATS2 : AttributeSet .
var Pid : PlanID .

cr
cr

endm

cr

cr

cr
cr

cr
cr

cr

cr

cr

cr

cr

cr

cr
cr

cr
cr

cr
cr

cr
cr

cr
cr

cr
cr

cr

Fig. 4. The definition in Maude of the program of Section III

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

cr

cr
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.

newInternalID, newInternalID, 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, pid1)) 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 formula P =><> 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 continues 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.

VI. 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 V). 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