Programming Dynamic and Open Multi-Agent Systems with Organization Metaphor

Cuiyun Hu, Xinjun Mao, Yin Chen
School of Computer, National University of Defense Technology, Changsha, Hunan Province, China 410073
Email: { hcy56316@163.com, mao.xinjun@gmail.com}

***Abstract*—Operating in highly dynamic and unpredictable environments with partial requirements, modern software systems are characterized by context-awareness, flexible organization and dynamic interactions. The increasing complexity desires natural high-level abstractions and effective programming mechanisms to facilitate the development and maintenance of such systems. However, the abstraction, composition and interaction mechanisms in current programming approaches do not directly support those characteristics. This paper proposes an organization-based agent-oriented programming (OrgAOP) approach that treats organizations, roles and groups explicitly, and provides mechanisms to support context-awareness, flexible organization and dynamic interactions. This paper describes the philosophy of OrgAOP, designs a programming language OragentL with explicit language facilities supporting OrgAOP and illustrates how to program and implement MAS with OragentL by a case study.**

***Index Terms*—Agent-oriented programming, Organization theory, Organization-based programming, Dynamics**

I. Introduction

Software systems today are increasingly expected to mirror real-world process, i.e. various parts of the system even the whole system are often derived from the structure and process of human society and co-evolve with the social systems (e.g. e-business applications, information management systems) [1][2][6]. Therefore, current trends in software engineering are facing up the challenges of constructing and managing software systems operating in dynamic and unpredictable environments with partial requirements and interacting with the real-world in a seamless way. The complexity of modern software systems undermines the assumptions of traditional software engineering (i.e. object-orientation), such as fixed requirements, static environment or with priori know changes and infrequent failures[1][3], and has increased the need for natural high-level abstractions to facilitate the development and maintenance of the systems.

Agent-oriented software engineering (AOSE) seems to provide a proper abstraction and method for developing dynamic and open systems, especially for the mobile systems[4] and virtual organizations [5], since it advocates decomposing problems in terms of autonomous agents that can engage in flexible, high-level interactions [2]. In particular, with inspiration from organization theory, AOSE provides a natural representation of such systems with organizational concepts. Recently a variety of organization-oriented approaches to multi-agent systems engineering have been brought forth including modeling approaches (e.g. ARG, Moise+, ODML), methodologies (e.g. Gaia, MaSE, OMNI, Tropos), infrastructures(e.g. MADKIT, S-Moise) and programming languages (e.g. 2OPL). A detail review can be found in [7].

While organization metaphor has made significant contributions to analysis and design multi-agent system (MAS), when it comes to implementation, it becomes obvious that the true potentials of the organization metaphor have not been entirely exploited. In fact, current MASs are usually implemented as a set of agents in terms of mental concepts (e.g. goals, beliefs, plans and etc.), where information about organization structure and collective behavior is lost [8][9]. As a result, programmers have to manually translate and incorporate the organizational concept from design model to metal concepts, which leads to poor engineering practice and hinders engineers to exploit the full potentials of AOSE.

A recent trend in the AOSE is to employ organization concepts in agent-oriented programming. However, currently most organization-based programming approaches for MAS oblige the programmers to programming organization concepts and agents using two different languages without a unify programming and computation model. For example, in J-Moise, organization concepts are presented with XML and agents are programmed with Jason [10]. Moreover, current researches in this field usually focus on the normative MAS with the aims to deal with the openness and heterogeneity [7], but inadequate in handling dynamics and flexibility.

This paper proposes a new agent-oriented programming approach with organization metaphor, namely Organization-based agent-oriented programming (OrgAOP), to construct dynamic and open systems. An OrgAOP language – OragentL, is designed to allow programmers to construct the systems with first-class organizational concepts, such as organizations and roles. In addition, mechanisms supporting dynamics and flexibility are proposed. The rest of this paper is organized as follows. Section II discusses the

Corresponding author: Cuiyun Hu, hcy56316@163.com

doi:10.4304/jsw.8.1.151-159

characteristics and programming challenges of dynamic and open MAS. Section III introduces the philosophy of the new programming approach OrgAOP. Section IV designs the OrgAOP language OragentL with a detail description of its syntax and formal execution model. Section V analyzes the related works and finally conclusions are made and future works are discussed in section VI.

II. Challenges for Programming Dynamic and Open MAS

This section starts from the characteristics of dynamic and open MAS, forming the basis from which we distill a number of fundamental challenges when programming such systems. A dynamic and open MAS usually consists of a changing agents and is situated in a dynamic environment. The following characteristics distinguish dynamic and open MAS from traditional ones.

- *Context-awareness*. The context of dynamic and open MAS, including devices, network conditions, personal preference and etc., is not known until they are in operation. Therefore, the entities in the systems should have the ability of context-awareness to sense and dynamically adapt their behavior according to the execution context.
- *Flexible organization*. The agents involved are usually developed, owned and controlled by different organizations and autonomous to enter or leave the system and decide whether to provide services or not [6]. Therefore, a flexible organizing mechanism is needed to enable the whole system to behave in a reasonable manner in presence of changing membership and internal failures, e.g. unpredictable leaving and refusing service of some entities.
- *Dynamic interactions*. Owing to the openness, the agents and interactions can not be known at design time. Instead, it is necessary for the agents to get their partners and interaction patterns dynamically at run-time.

The characteristics described above challenges traditional programming approach (i.e. object-orientation): (i) adequate programming abstractions for specifying context-specific behaviors and dynamic composition mechanisms are lacked to support context-awareness; (ii) the composition mechanism in object-oriented programming is reference mechanism, which is considered statically and can not support the flexible organization; (iii) both in object-oriented programming and agent-oriented programming, basic interaction mechanism is message passing, which implicitly assumes that an entity knows all the entities that it may interact with. So message passing mechanism is not realistic for programming dynamic and open MAS with highly unpredictable dynamic interactions.

Therefore, current programming languages focus on static design and offer few mechanisms to support dynamics and flexibility. And it is necessary to develop high-level language abstractions and mechanisms that can address fine and coarse grained dynamics, coping with radical changes from both environments and requirements [1].

III. OrgAOP: A New Organization-based Agent-Oriented Programming Ap-proach

The development of dynamic and open MAS becomes more complex than necessary since there is a lack of explicit support for the challenges in programming language. This section presents a new programming approach, called organization-based agent-oriented programming (OrgAOP), which is meant to satisfy the above challenges by applying organization metaphor in agent-oriented programming.

A. Philosophy

In OrgAOP, a MAS is viewed as a set of groups, each of which can be viewed as a dynamic aggregation of agents, i.e. an agent can dynamically enter or leave a group at run-time. A group provides interaction context for its members, i.e. only interactions between agents in the same group are allowed. Moreover, each group can also regulate its structure (mainly through adding or removing members) to adapt to the dynamic environment and memberships. A set of groups with the same structure and regulation behavior are abstracted as an organization. A role defines the behavior of agents in a special type of groups (i.e. an organization). At run-time, agents can enact multiple roles in multiple groups. So an organization is composed with a set of roles and a regulation behavior. The following presents the philosophy of OrgAOP according to the three characteristics of dynamic and open MAS: context-awareness, flexible organization and dynamic interactions.

- *Context-awareness.* OrgAOP represents context-specific behaviors with roles, which can be dynamically composed at run-time depending on execution context with a role enactment mechanism. Role enactment is a new mechanism introduced in AOP by Mehdi Dastani in [14] with the aim to capture role dynamics in terms of four operations: enact, deact, activate and deactivate. However, in OrgAOP, role enactment has been attached more meanings. In contrast to the instantiation mechanism in object-oriented programming (OOP), enactment mechanism describes the relationship between agents and roles. While an object has to adhere to one class that cannot be changed once it is instantiated, an agent can possess multiple roles that can be changed dynamically during its lifetime.
- *Flexible organization.* At design time, OrgAOP models a system as a hierarchical organization, which means that an organization is a self-contained entity and can composite a set of sub-organizations. However, at run-time, an origination can be instantiated as several groups and the structure of an OrgAOP application can be viewed as a social network, since agents can play multiple roles in multiple groups. Groups are considered as dynamical aggregation of agents with self-management ability to maintaining a homeostatic relationship between their environments

and their internal members.

- *Dynamic interactions*. OrgAOP supports dynamic interactions with a role-based interaction mechanism, which allows the programmers to specify interactions between agents based on roles independently from the executing agents. At run-time, OrgAOP also takes roles as execution entities that are responsible for dispatching the messages to their players. As the players of each role are changed from time to time, when receiving a message, the role check its current available players and dispatches the message to one or all of its players. OrgAOP provides two kinds of interaction patterns based on roles: one-to-one interaction, which means to choose a player of the special role to send a message; one-to-many, which means to send message to all the players of the special role.

B. An Example: On-line Auction System

Throughout this paper, an example of online auction system is studied to illustrate how to programming MAS with OrgAOP. The on-line auction system is implemented as an organization named *AuctionOrg*, which is composed by two sub-organizations, one implementing the online bidding function named *BiddingOrg* and one implementing the online payment function named *PaymentOrg*. *BiddingOrg* has three roles: *Auctioneer* managing the status of an auction, *Provider* providing goods to auction and *Bidder* bidding on the goods. The group of *BiddingOrg* is dynamically created when there are goods on auction and is terminated when the auction is completed. *PaymentOrg* also has three roles: *Payer* paying for the winning goods, *Payee* receiving the payment and *Broker* transferring the money from the account of the payer to the one of the payee. *Broker* agents can be dynamically created and killed according to the number of the money transfer request. In the *AuctionOrg*, *Buyer* specifies the agents that play *Bidder* in *BiddingOrg* to place bidding and play *Payer* in *PaymentOrg* to pay for the goods; and *Seller* specifies the agents that play *Provider* to provide goods and play *Payee* to receive money.

IV. ORAGENTL: AN ORGANIZATION-BASED AGENT-ORIENTED PROGRAMMING LANGUAGE

OragentL provides programming facilities to program both organizations and roles and to support the programming mechanisms such as role enactment and role-based interactions based on the computational and programming model defined in [13]. In the follows, the syntax of OragentL is presented by means of the on-line auction system example, and then the execution of the Oragent program is explained.

A. Syntax of OragentL

The syntax of OragentL is presented in Fig. 1 using the EBNF notation. <group_id> and <agent_id> are <identifier>, sequences of letters and digits starting with a lower letter, to denote the addresses of a group and an agent, respectively. <org_name> and <role_name> are <literal>, sequences of letters and digits starting with a capital letter, to denote the names of an organization and a role, respectively. In following, the core constructs of OragentL, i.e. organizations, roles and behaviors, are explained in detail.

```
<org_prog> ::= ["within" <org_fname> ";"]
                {"employ" <org_fname> ";"}<org_def>
<org_def> ::= "organization" <org_name>
                ["(" <parameters> ")"] "{"
                  [<attribute_declarations>]
                  {<role_def>}
                   <init_beh>
                  [<regulate_beh>] "}"
<parameters> ::= <var_exp> {"," <var_exp>}
<var_exp> ::= <type> <identifier>
<type> ::= <literal>
<attribute_declarations> ::= {<var_exp> ";"}+
<org_fname> ::= <org_name> ["." <org_name>]
<org_name> ::= <literal>

<role_prog> ::= ["within" <org_fname> ";"] <role_def>
<role_def> ::= <modifier> "role" <role_name>
                ["(" <parameters> ")"] ["play" <role_names>]"{"
                  [<attribute_declarations>] <behaviors> "}"
<modifier> ::= <visibility> [<multiplicity>]
<visibility> ::= "internal" | "external"
<multiplicity> ::= "single"
<role_name> ::= <literal>
<role_names> ::= <role_name> {"," <role_name> }

<behaviors> ::= {<behavior>}+
<behavior> ::= ["loop"]"when(" ("initialize"|<msg_template>)"){"
                  <statements>"}"
<statements> ::= {<statement>}
<statement> ::= ";"| <assign_stm> | <if_stm> | <while_stm> |
                  <action_stm> | <receive_stm> | <on_stm> | <statements>
<action_stm>::= <role_transfer> | <send_msg> | <agent_act> | <defined_act>
<role_transfer>::= ("enact" | "deact" | "activate" | "deactivate") <role_fname>
<role_fname>::= <group_id> "." <role_name>
<send_msg>::= <receiver_exp> "." <msg>
<receiver_exp> ::= <agent_id> | <role_name> | <role_name>"*" | "_"
<agent_id> ::= <identifiter>
<msg> ::= <msg_name> "(" [<vars>] ")"
<vars> ::= <var> {"," <var>}
<var> ::= <identifier>
<on_stm> ::= "on(" <msg_template> "){" <behavior> "}"
<receive_stm> ::= "receive {" <msg_match_stms> "}"
<msg_match_stms> ::= {<msg_match_stm>}+
<msg_match_stm> ::= <msg_temp> ":"  <behavior> "."
<msg_template> ::= <sender_exp> "!" <msg_name> "(" <msg_pattern> ")"
<msg_pattern> ::= {<vars> | <var_exps>}
<agent_act> ::= "new" <role_name> | "kill" <role_name>["*"]
                  | "fire" <role_name>["*"]
<defined_act> ::= <act_name> "(" <vars> ")"
<action_def> ::= <type> <act_name> "(" <var_exps> "){" <statements> "}"
```

Figure 1. EBNF grammar of OragentL.

Organizations in OragentL are allowed to be nested, i.e. an organization can recursively defines sub-organizations. An organization program starts with the keyword “within” to declare its nested organization if there is one, and the keyword “employ” to declare its sub-organizations that are developed by others or have already existed. The formal parameters of the organization that is needed when creating a new group is declared <parameters> following the <org_name>. The roles of the organization can be either defined inside the organization code or defined in a new role program independently.

A role can be viewed as a set of behaviors that an agent can gain by enacting the role. OragentL distinguishes two kinds of roles: internal roles, which can only be enacted by the native agents (i.e. the agents created by its group), and external roles, which is

provided for outside and can be viewed as the services provided by its group. If the "single" modifier appears in the definition of a role, the role will have at most one player. The keyword "play" is used to declare the roles that the players of a role can enact. For example, "R1 play R2, R3" means that the agents that enact role R1 can also enact both R2 and R3, which can be dynamically transferred at run-time. OragentL calls R2 and R3 are sub-roles of R1, and the role R1 is a super-role of both R2 and R3. If the "play" keyword appears, the super-role should define the role transformation among its sub-roles.

The specification of a behavior is started with the keyword "when", followed by the trigger event (i.e. messages form other agents or itself). The keyword "loop" is used to declare the behavior is circular or one-shot. The body of a behavior is a statement block in the form <statements>. OragentL distinguishes three special kinds of behaviors. (i) Initialization behavior, denoted by the "initialize" message, executes as long as the entities are created. (ii) Regulation behavior, within organization program, is specified based on the regulation action to manage its members. OragentL allows a group to regulate its structure by create a new agent, kill a native agent or fire a foreign agent. (iii) Role transfer behavior, defined in the role that has sub-roles, is specified based on the role transformation actions, such as **enact**, **deact**, **activate** and **deactivate**.

OragentL provides multiple actions for message sending. Firstly, each OragentL agent can send message to a known agent with the agent's identifier. Secondly, an OragentL agent can send a message to an arbitrary player of a special role. Thirdly, an OragentL agent can send a message to all the players of a special role in the form of "<rolename>"*"."<msg>". At last, an OragentL agent can publish an event by sending the message without an explicit receiver. Each OragentL agent owns a message queue, <on_stm> and <receive_stm> are used to fetch a message from the queue. The difference between them is that <on_stm> can subscribe event. So besides role-based interaction mechanism, OragentL also supports event-based interaction based on subscribe/publish mechanism.

The OragentL code of the on-line auction system is given in Fig. 2. The buyer agent firstly subscribes wanted good by "on(action(good, BiddingOrg g))", where "good" is a variable storing the wanted good and the "g" is a variable to get the bidding group address from the receiving message. The seller agent creates a bidding group for its auction good by "new BiddingOrg()" and enacts the *Provider* role in the created group. As a provider, the agent can send message with the auction good and initial price to the *Auctioneer* agent that publishes the auction good and the group address. The buyer agents who subscribe the good will receive the message and then join the group and enact *Bidder* role to place a bid on the good. The following describes how to program role transfer and regulation behaviors and interaction actions in detail.

In Fig.2 (a), *Buyer* role has two sub-roles: *BiddingOrg.Bidder* and *PaymentOrg.Payer*, and the role transfer behavior means: if receiving a winning message from *Auctioneer* role, the agent will enact *Payer* in the payment group g; if receiving a lower message from *Auctioneer* role in bidding group g, the agent will deact *Bidder* role in g; if receiving a payment success message from *Broker* role in payment group g, the agent will deact *Payer* role in g.

```
organization AuctionOrg{
 internal role Buyer(string good, int price) play
    BiddingOrg.Bidder, PaymentOrg.Payer{
  ……
  when(){
   //subscribe the auction of good
   on(auction(good, BiddingOrg g)){
      enact g.Bidder(price).
   }
   //roles transfer behavior
   loop receive{
     Auctioneer !lower(BiddingOrg g):   deact g.Bidder.
     Auctioneer!win(good, PaymentOrg g):  enact g.Payer(price).
     Broker!payment(PaymentOrg g,"success"):  deact g.Payer.
     }
 } }
 internal role Seller(string good, int price) play
    BiddingOrg.Provider, PaymentOrg.Payee{ ... }
}
```

(a) code fragments of AuctionOrg

```
within AuctionOrg;
organization BiddingOrg{
  ……
 external role Bidder(int price){...}
 external role Provider(int price){...}
 internal single role Auctioneer{
   when(Provider provider ! auction( string good, int price)){
     setAuction(good, price);
     _.auction(mygroup, good)
     loop receive{
       Bidder b ! bid(int p):  handleBidding(b, p).
     }after (2000)
     makeWinner();
     gid=findPaymentGroup();  winner.win(good, gid);
     provider.trading(gid, good);
     receive{
       provider ! payment("failure"):  handlePaymentFailure().
     }
   }
   //action
   handlePaymentFailure(){...}
} }
```

(b) code fragment of BiddingOrg

```
within AuctionOrg;
organization PaymentOrg{
  ……
 internal role Broker{...}
 external role Payer(int money){...}
 external role Payee(int money){...}
 // structure regulating  behavior
 loop when(){
  if(playersNum(Payer)/  playersNum(Broker)>5){
     new Broker();
  }
  if(playersNum(Broker)<playersNum(Payer)){
    kill Broker;
} } }
```

(c) code fragments of PaymentOrg

Figure 2. Code fragments of on-line auction system.

Considering the regulation behavior of the payment group in Fig.2 (c), if the payment request for each *Broker*

agent is more than 5, the group create a new *Broker* agent, and if a Broker agent is free and it is not the only *Broker* agent in the group, the group will kill it.

B. Execution Machine of OragentL Program

In OragentL, there are three kinds of execution entities: agents, roles and groups. All these entities execute in parallel in a MAS. This section defines three abstract state machines, namely $execOragent_A$, $execOragent_R$ and $execOragent_G$, for agents, roles and groups respectively. Some basic symbols and functions defined in [11] are used to specify the OragentL program state. The meanings of these symbols and functions are given in Table I.

TABLE I.
MEANINGS OF THE BASIC SYMBOLS AND FUNCTIONS TAKEN FROM [9]

Symbol/Function	Meaning
$^{\alpha}exp$ /$^{\alpha}stm$	α denotes the position of the expression or the statement
$^{\triangleright}exp$/$^{\triangleright}stm$	$\triangleright$ indicate where the machine is positioned
pos: Pos	*pos* denotes the current position
restbody: *Pos*→*Phrase*	*restbody* denotes the part of the current program that still to be executed
restbody/pos	The context of the pending computation of current phrase
up: *Pos*→*Pos*	*up* yields the parent position of a position, thus allowing to retrieve for a phrase and to move to the next enclosing phrase.

A phrase in Oragent can be seen as a set of statements that are surrounded by a pair of "{" and "}". When the phrase is terminated (denoted by *Termination*), the context switch can be captured by the following definition:

$$context(pos) = \textbf{if}\ \ pos = Termination\ \ \textbf{then}\ \ restbody / up(pos)\ \ \textbf{else}\ \ restbody / pos$$

Every behavior of an agent can be blocked by waiting a message or interrupted by receiving regulation messages from its groups. So when a behavior is blocked or interrupted, the state of the behavior *beh*, consisting of *beh*, *restbody* and *pos*, is stored to be resumed when *beh* is scheduled again. A behavior is active if the role it belongs to is active, and a behavior is inactive if the role it belongs to is inactive. So two lists *activeBehs* and *inactiveBehs* are used to store the states of the active behaviors and inactive behaviors of an agent, respectively, and the elements of the lists are added at end and removed at head. When a behavior is scheduled at the first time, *restbody* is initialized with the behavior body and *pos* is initialized with its start position *firstPos*. The following defines the transition rules for each machine to simulate the execution of agents, groups and roles respectively.

- *Transition Rules for the Machine $execOragent_A$*

Each agent executes an $execOragent_A$ machine, which manages a set of enacted roles and their corresponding states, a behavior queue and a message queue. The role of an agent can be either active or inactive, and only the behaviors of the active roles can be executed by the machine. Moreover, every behavior can be blocked by waiting a message or interrupted by receiving a regulation message from its groups. When a new behavior is added, or a behavior is blocked or interrupted, the state of the behavior is stored in the behavior queue to be scheduled. In order to reaction to the regulation of its groups in time, the handling of the regulation messages is prior to all others. Therefore, the $execOragent_A$ machine has two parts executing in parallel. The behavior manager gets the behavior to be executed at the head of the behavior queue, executes the behavior and adds a new/blocked/interrupted behavior at the end of the behavior queue. The message manager is responsible for monitoring regulation messages. If a regulation message is received, it is added at the head of the message queue; otherwise, the message is added at the end of the message queue.

The transition rules for $execOragent_A$ are described in Fig. 3, where each agent schedules a behavior at one moment and executes the behavior based on the syntactical structure of the program. The behaviors can be either user-defined behaviors or reactions to the received regulation messages form its groups. In Fig. 3, the function *players*(*r*) returns a set of agents that enact the role *r*; the function *getMsg*() returns the first message of the message queue of the agent, and the function *regulationMsg*(*msg*) is used to estimate whether the given message *msg* is a regulation message or not.

$execOragent_A$ = **case** $context(pos)$ **of**
  $enact\ \ g.r \rightarrow enact(g.r)$
  $deact\ \ g.r \rightarrow deact(g.r)$
  $activate\ \ g.r \rightarrow activate(g.r)$
  $deactivate\ \ g.r \rightarrow deactivate(g.r)$
  $r.m(^{\alpha}exp) \rightarrow pos := \alpha$
  $r.m(^{\triangleright}val) \rightarrow$ **choose** $a \in players(r)$
    $send(a, m, val)$
    $up(pos)$
  $r*.m(^{\alpha}exp) \rightarrow pos := \alpha$
  $r*.m(^{\triangleright}val) \rightarrow$ **forall** $a \in players(r)$
    $send(a, m, val)$
    $up(pos)$
  $^{\triangleright}receive\{^{\beta}guards\} \rightarrow$
    let $msg = getMsg()$
      if $msg \neq null \wedge \neg regulationMsg(msg)$ **then** $pos := \beta$
      else
        $activeBehs := activeBehs : [(beh, restbody, pos)]$
        if $regulationMsg(msg)$ **then**
          $respond(msg)$
        else
          let $activeBehs = [(beh', restbody', pos')] : activeBehs'$
            $beh := beh'$
            $restbody := restbody'$
            $pos := pos'$
  $guard :^{\alpha} stm.^{\beta} guard : stm. \rightarrow$
    if $match(guard, msg)$ **then** $pos := \alpha$
    else $pos := \beta$
  $guard :^{\triangleright} stm. \rightarrow up(pos)$
  $^{\alpha}receive\{^{\triangleright}NotMatch\} \rightarrow pos := \alpha$

Figure 3. Execution of $execOragent_A$

In Fig. 3, the rule *enact*(*g.r*) adds the role *g.r* as the agent's new active role and adds all the behaviors of *g.r* at the end of *activeBehs*, and sends a message to the role. All the roles enacted by the agent is stored in *enactedRoles*, the elements of which consists the role's identifier (as the form *g.r* with *g* is its group's identifier and *r* is the role name) and the role state (either *Active* or *Inactive*). The *enact*(*g.r*) rule is defined as follows:

$enact(g.r) =$
  $enactedRoles = enactedRoles \cup \{(g.r, Active)\}$
  forall $b \in behaviors(g.r)$
    $activeBehs := activeBehs : [(b, body(b), firstpos)]$
    $send(g.r, "enact")$
    $up(pos)$

The rule *deact*(*g.r*) removes the role *g.r* and all its sub-roles, removes all the behaviors of it and its sub-roles, and sends messages to all the removed roles. The function *own* computes whether the agent own one behavior or not, which is defined as:

$$own(behList, b) = \textbf{let } beh_b = (b, \_, \_) \textbf{ if } beh_b \in behList \textbf{ then } Ture \textbf{ else } False$$

So the *deact*(*g.r*) rule is defined as follows (where the function *sub*(*r*) returns all the sub-roles of role *r*.):

$deact(g.r) =$
  $enactedRoles = enactedRoles / \{(g.r, Active)\}$
  $send(g, r, "deact")$
  forall $b \in behaviors(r')$
    if $exist(activeBehs, b)$ **then** $remove(activeBehs, b)$
    if $exist(inactiveBehs, b)$ **then** $remove(inactiveBehs, b)$
  forall $(role', s) \in enactedRoles \wedge role' \in sub(r)$
    $deact(role')$

The rule *activate*(*g.r*) changes state of the role *g.r* and its behaviors from inactive to active. Especially, if the agent has not enacted *g.r*, the $execOragent_A$ will generate an exception. A function is defined to implement the movement a behavior from a behavior list to another:

$move(behList_1, b, behList_2) =$
  let $(b, restbody, pos) \in behList_1$
    $behList_1 = behList_1 \setminus \{(b, restbody, pos)\}$
    $behList_2 = behList_2 \cup \{(b, restbody, pos)\}$

And the *activate* (*g.r*) rule is defined as follows:

$activate(g.r) =$
  if $(g.r, s) \notin enactedRoles$
    then $failUp(NotEnactedRoleException)$
  else
    if $active(g.r)$ **then** $up(pos)$
    else
      $enactedRoles = (enactedRoles \setminus \{(g.r, Inactive)\}) \cup \{(g.r, Active)\}$
      $send(role, "activate")$
      forall $b \in behaviors(g.r)$
        if $exist(inactiveBehs, b)$ **then**
          $move(inactiveBeh, b, activeBeh)$
      $up(pos)$

The rule *deactivate*(*g.r*) changes state of the role *g.r* and all its sub-roles, and moves all the behaviors of these roles from active behavior list to inactive behavior list. The *deactivate* (*g.r*) rule is defined as follows:

$deactivate(g.r) =$
  if $g.r \notin enactedRoles$ **then**
    $failUp(NotEnactedRoleException)$
  else
    if $inactive(g.r)$ **then** $up(pos)$
    else
      forall $b \in behaviors(g.r)$
        if $exist(activeBehs, b)$ **then**
          $move(activeBehs, b, inactiveBehs)$
      forall $role' \in enactedRoles$
        if $active(role') \wedge role' \notin super(g.r)$ **then**
          $deactivate(role')$
      $up(pos)$

Moreover, although an agent can send messages to another agent with the agent identifier, we only consider the role-based interaction which is new for OragentL. From Fig. 3, we can see that, in this way, an agent just send the message and parameters to the corresponding role but leaving the actual receiver agent open. Then the role handles the receiving messages based on the transition rules defined in section 4.2.3.

- *Transition Rules for the Machine $execOragent_G$*

In OragentL, each group executes an $execOragent_G$ machine, which executes the user-defined regulation behavior to manage its memberships. The execution of the $execOragent_G$ machine is described in Fig. 4. An agent creation based on a role requires the role to be initialized, i.e. the role is ready to receive messages from others. A new agent is created with a function "**create** agent" which means to create an agent with a fresh agent identifier "agent". The initialization state of the agent is recorded by the function *heap*(*agent*). And the data structure $Agent(\{r\},\varnothing,\varnothing)$ stores the enacted roles, active behaviors and inactive behaviors of the agent, respectively. Then, the group notifies the corresponding role with the new agent identifier. The execution of the agent is started by the following function:

$start(agent, group, r) =$
  let $role = groupNm(group).r$
    $rolesOf(agent) = \{role\}$
    $roleState(role) = Active$
    forall $b \in behaviors(role)$
      $activeBehs(agent) = activeBehs(agent) \cup \{b\}$
      $behaviorState(b) = Ready$

Moreover, the regulation actions are executed by sending a corresponding message to the involved role. In other words, a group regulates its structure through its roles.

$execOragentGroup_B =$ **case** $context(pos)$ **of**
  $new\ r \rightarrow$ **if** $initialized(r)$ **then**
    create *agent*
    $heap(agent) = Agent(\{r\},\varnothing,\varnothing)$
    $send(r, "new", agent)$
    $payersNum(r) = payersNum(r) + 1$
    $start(agent, group, r)$
  else $initialize(r)$
  $kill\ r \rightarrow$ **if** $\neg internal(r)$ **then** $failUp(KillNotInteranlException)$
  else
    $send(r, "kill")$
    $payersNum(r) = payersNm(r) - 1$
  $kill\ r^* \rightarrow$ **if** $\neg internal(r)$ **then** $failUp(KillNotInteranlException)$
  else
    $send(r, "killAll")$
    $payersNum(r) = 0$
  $fire\ r \rightarrow send(r, "fire")$
    $payersNum(r) = payersNm(r) - 1$
  $fire\ r^* \rightarrow send(r, "fireAll")$
    $payersNum(r) = 0$

Figure 4. Execution of $execOragent_G$

- *Transition Rules for the Machine $execOragent_R$*

In OragentL, when a group is created, the initialization of the group should create all the roles that defined in its organization. A role is responsible for dispatching messages to its players, so a role can be viewed as a reactive entity to react to the received messages. Each role executes an $execOragent_R$ machine, which manages a message queue storing the messages from its group and

its players, and dispatches the messages to its group or its players. The messages received by roles can be classified into three types: regulation messages received from its group, role transformation messages received form its players and normal communication messages received from other agents and required to be dispatched to its players. Handling a communication message is simple, as it will not change the state of the role or its group, and the role just dispatches the message to one or all of its players.

Fig. 5 describes how a role responds to regulation and role transformation messages. When the role receives a *new* message from its group, it just adds the agent as its active player. When the role receives a *kill* message from its group, it first removes one of its players or clears all its players depending on the parameter of the message whether *one* or *all* and then sends a dead message to the removed agent(s). A *fire* message is handled similarly to the *kill* message at the first step, but in the second step, the role sends a *fired* message to the removed agent(s). The selection mechanism for the removed player (both in the *kill one* and *fire one* messages) is out of the scope of this paper. For the *enact* and *deact* messages, the role first adds or removes the agent and then notifies its group. Moreover, for the *activate* and *deactivate* messages, the role just changes the state of the agent in the role.

```
execOragentRole =
  while msgQueue ≠ null
    case msg = top(msgQueue) of
      ("new", a) → players = players ∪ {(a, Active)}
      ("kill","one") → choose (a,s) ∈ players
                       players = players \ {(a,s)}
                       send(a,"dead")
      ("kill","all") → players = ∅
                       forall (a,s) ∈ players
                         send(a,"dead")
      ("fire","one") → choose (a,s) ∈ players
                       players = players \ {(a,s)}
                       send(a,"fired")
      ("fire","all") → players = ∅
                       forall (a,s) ∈ players
                         send(a,"fired")
      ("enact",ε) → players = players ∪ {(sender(msg), Active)}
                    send(mygroup,"enact")
      ("deact",ε) → if ¬exist(players, sender(msg)) then
                      fialUp(UnknownPlayerDeactRoleException)
                    else
                      let (a,s) = lookup(players, sender(msg))
                        players = players \ {(a,s)}
                        send(mygroup,"deact")
      ("activate",ε) → let a = sender(msg)
                       if ¬exist(players,a) then
                         fialUp(UnknownPlayerActivateRoleException)
                       else
                         if inactive(a) then
                           players = (players \ {(a, Inactive)}) ∪ {(a, Active)}
      ("deactivate",ε) → let a = sender(msg)
                         if ¬exist(players,a) then
                           fialUp(UnknownPlayerDeactivateRoleException)
                         else
                           if active(a) then
                             players = (players \ {(a, Active)}) ∪ {(a, Inactive)}
```

Figure 5. Execution of $execOragent_R$ according to regulation and role transformation messages

V. RELATED WORKS

With the increasing demand for dynamics and openness of software systems, there are a number of research proposals in programming field that attempt to present and manage the dynamics and openness with organization metaphor. This section will try to summarize some representative works based on the programming challenges of dynamic and open systems. These works have addressed one or more of the challenges, and are classified into two categories according to the underlying programming paradigms.

The first category is based on OOP, and devises some new programming model by extending objects, such as COP[12], AmOP [20], role-based programming (RBP) [21][22] and interaction-oriented programming (IOP) [23]. AmOP proposes an ambient actor model, which can dynamically detect available/unavailable resources in its context and recover from partial failures. So AmOP supports dynamic interactions and flexible organization. COP introduces dedicated abstraction for the modularization (e.g. layers, roles and etc.) and dynamic composition of crosscutting context-dependent behavior [12]. COP focuses the context-awareness but leaving the other characteristics open. RBP aims to support dynamic behavior composition and complex interactions among objects based on roles, which provides a potential for the context-awareness and dynamic interactions. IOP aims to manage and control the complexity of interactions by reifying interactions. Table 2 gives the evaluation of these works based on the characteristics of dynamic and open systems. Although these works attempt to support dynamic and open systems by extending OOP, the intrinsic and static features of OOPL limit the potential to support dynamics and openness.

TABLE II.
EVALUATION OF OOP-EXTENDED APPROACHES BASED ON CHARACTERISTICS OF DYNAMIC AND OPEN SYSTEMS

Challenges	COP	AmOP	RBP	IOP
Context-awareness	√		√	
Flexible organization		√		
Dynamic interaction		√	√	√

The second category is based on AOP, and provides explicit organization abstractions with a programming language or middleware. Many organization frameworks and middleware have been proposed in the literature. For example, Janus[15] is a platform based on java for holonic agent, in which roles are used to define the context-specified behaviors, an organization is used to abstract the groups of agents, a role-based interaction mechanism allows agent to get their partners dynamically at run-time and a holonic agent can recruit agents and manage its memberships according to the programmer-defined conditions. powerJade[16] allows an agent to dynamically compose its behavior by playing different roles, and as an autonomous agent, an organization can manage itself and reify its internal interactions by defining management and interaction behaviors, respectively. MACODO[17] does not consider agents, just focuses on the merging and slipping of organizations. J-Moise+ [10] allows the organization dynamics such as dynamic creation of a group, group merging, dissolving a group, but the organization dynamics are specified inside the agents, which pollutes the agent's code. AMELI [18] provides dialogues and scenes to support open systems.

An alternative to middleware is programming languages. However, currently there are very few programming languages providing explicit organization abstraction. MetataM [19] is one of such works which

introduces the notion of group by enlarging the notion of agent with a context and content. MetataM agents have the ability to decide to join or leave a context with the primitives of *moveInto*(Agent) and *moveUp*(Agent), respectively, and recruit or remove its content with the primitives of *goInto*(Agent) and *goUp*(Agent), respectively. So we can see the MetataM agents can be dynamic organized, i.e. an agent can decide its members. Another work is in [14], where agent organizations are programmed with norms and roles. The role enactment enables the agents to dynamically compose their behaviors by enacting different roles according to organization context. However, this work lacks a clear description of how the approach integrates with the agent level from a practical programming point of view. Finally, in [24] a general-purpose agent-oriented programming language is designed to support the concurrency and distributed systems, which employs a new programming construct – artifact to support the dynamics and openness rather than an organization abstraction.

A detail comparison of the above approaches is given Table 3. Firstly, the languages are used to specify the organization abstraction and agents. Current approaches can be classified in to three types: (i) both organization abstractions and agents are specified with object-oriented programming language (e.g. Janus, PowerJade); (ii) organization abstractions are specified with XML and agents are specified with either object-oriented language or agent language event undefined (e.g. MACODO, J-Moise, EI); (iii) both organization abstraction and agents are specified with agent languages (e.g. MetataM). The context-awareness and dynamic interaction can be supported by library (platform), middleware or language facilities. And the flexible organization can be implemented either by agents, organizations or middleware. In Table 2, "–" means the approach does not support the ability, and "÷" means that the concept is undefined by the approach.

TABLE III.
COMPARISON OF AGENT ORGANIZATIONAL APPROACHES

Criteria	Language		Context-awareness	Flexible organization	Dynamic interaction
	Org	agent			
Janus	Java	Java	library	holon(agent)	platform
powerJade	Java	Java	library	–	platform
MACODO	Java	÷	middleware	middleware	middleware
J-Moise	XML	Jason	library	agent	–
EI	XML	÷	middleware	–	middleware
MetataM	MetataM	MetataM	language	agent	language
2OPL	2OPL	2APL	language	–	–
simpAL	simplAL		Language (artifact)	–	Language (artifact)

VI. CONCLUSIONS AND FUTURE WORK

This paper proposes a new programming approach—OrgAOP to support the characteristics, such as context-awareness, dynamic interactions and flexible organization, of dynamic and open systems. Firstly, to support context-awareness, roles are used to specify the context-specific behaviors, which agents can dynamically compose with enactment mechanism at run-time. Secondly, to support dynamic interactions, both role-based and event-based interaction mechanisms are provided. Thirdly, to support flexible organization, organizations are used to abstract the agent groups, of which the memberships changes dynamically at runtime. The main contribution of this paper is that the above mechanisms are supported in the programming language level by defining an OrgAOP language named OragentL. OragentL provides explicit language facilities for high-level organizational abstraction (i.e. organizations and roles), role enactment and role-based programming mechanism. Moreover, the syntax and formal execution model are defined in detail.

Furthermore, as a new programming approach, OrgAOP follows the main principles in programmming techinques such as abstraction, modualrity and reusability. (1) OrgAOP provides different levels of abstractions, such as organizations/groups for aggregation of agents, roles for context-specific behaivors of agents. (2) An agent can be viewed as a dynamic set of roles, each of which is relatively independent modular and can be composed dyanmically with enactment mechanism, (3) In OrgAOP, there are multiple entities can be reused, such as oragnizations, roles even agents.

OrgAOP is composed of programming model, language and infrastructure. The development of an infrastructure is ongoing based on the execution model defined in section 4.2. Moreover, some more complex cases should be studied to validate the availability and advantage for the development of dynamic and open systems.

ACKNOWLEDGMENT

Acknowledge the financial support from Natural Science Foundation of China under granted number 61070034 and 91024030, Program for New Century Excellent Talents in University, Opening Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University.

REFERENCES

[1] M. Broy, "The Grand Challenge in Informatics: Engineering Software-Intensive Systems," *IEEE Computer Society*, pp. 54-62, 2006.
[2] N.R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, vol. 117, no.2, pp. 277–296, 2000.
[3] F. Zambonelli and H. Van Dyke Parunak, "Towards a Paradigm Change in Computer Science and Software Engineering: a Synthesis," *The Knowledge Engineering Review*, vol.18, no.4, pp. 329-342, 2003.
[4] Y. Xu, Z. Zhao, W. Wu and Y. Zhao, "RPPA: A Remote Parallel Program Performance Analysis Tool," *Journal of Software*, vol. 6, no. 12, pp. 2399–2406, 2011.
[5] J. Hu, Y. Song and Y. Sun, "Multi-agent Oriented Policy-based Management System for Virtual Enterprise," *Journal of Software*, vol. 7, no. 10, pp. 2399–2406, 2012.
[6] W. Jiao, Y. Sun and H. Mei, "Automated Assembly of Internet-Scale Software Systems Involving Autonomous Agents," *The Journal of Systems and Software*, vol. 83, pp.

1838–1850, 2010.
[7] N.A.M. Tinnemeier, "Organizing Agent Organizations: Syntax and Operational Semantics of an Organization-Oriented Programming Language," SIKS Dissertation Series 2011(2), Utrecht University, 2011.
[8] R.H. Bordini, M. Dastani, and M. Winikoff, "Current Issues in Multi-Agent Systems Development," in *Proceedings of the Seventh Annual International Workshop on Engineering Societies in the Agents World*, 2007, pp. 38-61.
[9] N.A. Tinnemeier, M. Dastani, and J.-J.C. Meyer, "Roles and Norms for Programming Agent Organizations," in *Proceedings of the Eighth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Budapest, 2009, pp. 121–128.
[10] O. Boissier, J.F. Hübner, and J.S. Sichman, "Organization Oriented Programming: From Closed to Open Organizations," in *ESAW 2006*, LNAI 4457, P G. O'Hare et al., Eds., 2006, pp. 86–105.
[11] R. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.
[12] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-Oriented Programming," *Journal of Object Technology*, vol.7, no.3, pp. 125–151, 2008.
[13] C. Hu, X. Mao, Y. Chen, and H. Zhou, "OrgMAP: An Organization-based Approach for Multi-Agent Programming," in *AAMAS 2012*, Valencia, Spain, in press.
[14] M.M. Dastani, M.B. van Riemsdijk, J. Hulstijn, F.P.M. Dignum, and J.-J.C. Meyer, "Enacting and Deacting Roles in Agent Programming," in *AOSE 2004*, LNCS 3382, J.J. Odell, P. Giorgini, J.P. Muller, Eds., 2004, pp. 189–204.
[15] N. Gaud, S. Galland, V. Hilaire, and A. Koukam, "An Organisational Platform for Holonic and Multiagent Systems," in *PROMAS-6@AAMAS'08*, Estoril, Portugal, 2008, pp. 111–126.
[16] M. Baldoni, G. Boella, V. Genovese, R. Grenna, and L. van der Torre, "How to Program Organizations and Roles in the JADE Framework," in *MATES 2008*, LNAI 5244, R. Bergmann et al., Eds., 2008, pp. 25–36.
[17] D. Weyns, R. Heasevoets, A. Helleboogh, T. Holvoet, and W. Joosen, "MACODO: Middleware Architecture for Context-Driven Dynamic Agent Organizations," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 5, no. 1, pp. 1–25, 2009.
[18] M. Esteva, J.A. Rodriguez-Aguilar, B. Rosel, L. Joseph, "MELI: An Agent-based Middleware for Electronic Institutions," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems 2004*, ACM Press, New York, 2004, pp. 236-243.
[19] C. Ghidini, B. Hirsh, and M. Fisher, "Programming Group Computations," in *Proceedings of EUMAS '03*, 2003.
[20] J. Dedecker, T.V. Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-Oriented Programming," *in Proceedings of Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, New York: ACM Press, 2005, pp. 31-40.
[21] M. Baldoni, G. Boella, and L. van der Torre, "Interaction Between Objects in powerJava," *Journal of Object Technology*, vol. 6, no. 2, pp. 7 –12, 2007.
[22] T. Tamai, N. Ubayashi and R. Ichiyama, "An Adaptive Object Model with Dynamic Role Binding," in *Proceedings of ICSE'05*, St. Louis, Missouri, USA, May 2005.
[23] J.C. Cruz, *A Group Based Approach for Coordinating Active Objects*, Phd Thesis, der Universität Bern, 2006.
[24] A. Ricci and A. Santi, "Designing a General-Purpose Programming Language based on Agent-Oriented Abstractions: The simpAL Project," in *SPLASH'11 Workshops*, Portland, Oregon, USA, October 2011.

Cuiyun Hu, born in Huixian of Henan province, China, March 3rd, 1985. She received her Bachelor's degree in Computer Science and Technology from Central South University, Changsha, China, 2006, and her Master's degree in Computer Science and Technology from National University of Defense Technology, Changsha, China, 2008. She is a Phd student in National University of Defense Technology and her major researching field is agent-oriented software engineering.

Xinjun Mao, born in Jiangshan of Zhejiang province, China, 1970. He received the Bachelor's degree in Computer Science and Technology from College of Information Engineering, Zhenzhou, China, 2006, the Master's and PhD's degrees in Computer Science and Technology from National University of Defense Technology, Changsha, China in 1995 and 1998 respectively. His current main research interests include software engineering, agent theory and technology, self-adaptive and self-organizing systems.
Prof. Mao is the membership of IEEE and ACM, editor board member of several international journals and PC member of more than 20 international conferences/workshops. He has published two books and 100 papers in his interesting research area.

Yin Chen, born in Beijing, China, 1988. He is a Master candidate in National University of Defense Technology. His major researching field is agent-oriented software engineering.