Specifying Distributed Multi-Agent Systems in Chemical Reaction Metaphor

Lin, H.; Yang, Chunsheng

*Specifying Distributed Multi-Agent Systems in Chemical Reaction Metaphor **

Lin, H., and Yang, C.

April 2006

* published in The International Journal of Artificial Intelligence, Neural Networks and Complex Problem-Solving Technologies. Volume 24, Number 2. April 2006. pp. 156-168. NRC 48483.

Appl Intell (2006) 24: 155–168
DOI 10.1007/s10489-006-6936-x

Specifying distributed multi-agent systems in chemical reaction metaphor*

Hong Lin · Chunsheng Yang

Abstract This paper presents an application of Chemical Reaction Metaphor (CRM) in distributed multi-agent systems (MAS). The suitability of using CRM to model multi-agent systems is justified by CRM's capacity in specifying dynamic features of multi-agent systems. A case study in an agent-based e-learning system (course material updating) demonstrates how the CRM based language, Gamma, can be used to specify the architectures of multi-agent systems. The effectiveness of specifying multi-agent systems in CRM from the view point of software engineering is further justified by introducing a transformational method for implementing the specified multi-agent systems. A computation model with a tree-structured architecture is proposed to base the design of the specified multi-agent system during the implementation phase. A module language based on the computation model is introduced as an intermediate language to facilitate the translation of the specification of multi-agent systems. The multicast networking technology pragmatizes the implementation of communications and synchronization among distributed agents. The paper also discusses the feasibility of implementing an automatic translation from the Gamma specification to a program in the module language.

*This work is supported by University of Houston-Downtown Organized Research Committee.

H. Lin (✉)
Department of Computer and Mathematical Sciences, University of Houston-Downtown, 1 Main Street, Houston, Texas 77002, USA
e-mail: linh@uhd.edu

C. Yang
Institute for Information Technology, National Research Council, 1200 Montreal Road, Ottawa, Ontario, Canada K1A 0R6
e-mail: Chunsheng.Yang@nrc.gc.ca

Keywords Multi-agent systems · Program specification · Very high-level languages · Distributed systems · Software architecture

1. Introduction

Agent-oriented design has become one of the most active areas in the field of software engineering. The agent concept provides a focal point for accountability and responsibility for coping with the complexity of software systems both during design and execution [1]. It is deemed that software engineering challenges in developing large scale distributed systems can be overcome by an agent-based approach [2]. In this approach, a distributed system can be modeled as a set of autonomous, cooperating agents that communicate intelligently with one another, automate or semi-automate functional operations, and interact with human users at the right time with the right information. Such a model should be general enough to address common architectural issues and not be specific to design issues of a particular system. A direct benefit of such a model is expressiveness and extensibility—changes in the domain knowledge would not require an intensive system-wide modification to alter the information and objects that initiate actions based on that changing information.

For example, a distributed learning system typically involves many dynamically interacting educational components, each with its own goals and needs for resources, and engaged in complex coordination. It is very difficult to develop a system that could meet all the requirements for every level of educational hierarchy since no single designer of such a complex system can have full knowledge and the control of the system. In addition, these systems have to be scaleable and accommodate networking, computing and software facilities that support many thousands of simultaneous users concurrently working and communicating with one another [3]. Therefore, software engineering is burdened with unprecedented challenges in implementing such a learning environment, which should be of the following main features: adaptive curriculum sequencing, problem-solving support, adaptive presentation, student model matching. This gives justifications to find a model that can catch the interactive and dynamic nature of e-learning systems. One of the models in this specific area—Collaborative Agent System Architecture (CASA) [4] is an open, flexible model designed to meet the requirements from the resource-oriented nature of distributed learning systems. In CASA, agents are software entities that pursue their objectives while taking into account the resources and skills available to them. The collaborative architecture separates the modeling of multi-agent systems (MASs) from the specifications that designers need to commit with the given low-level mechanism of proprietary frameworks used in the implementation of multi-agent systems.

In this paper, we address the modeling issue in abstract computing machine level. Given the dynamic and concurrent nature of multi-agent systems, we find that the Chemical Reaction Metaphor [5, 6] provides a mechanism for describing the overall architecture of distributed multi-agent systems precisely and concisely, while giving the design of the real system a solid starting point and allowing step-by-step refinement of the system using transformational methods. The benefits of using CRM include: (1) The architectural design of the system can be separated from the design of individual units that have to deal with proprietary features of the underlying computing resources, because CRM allows us to treat each node in the distributed networking system as an element of a multi-set data structure, which in-turn can be an active program to be defined in a lower level of the program structure. (2) Parallelism can be easily achieved without extra effort in designing communication and synchronization mechanism because CRM expresses them implicitly. (3) Concurrency and the dynamic nature of MAS can be easily reflected by CRM's non-determinism feature. (4) Autonomy can be expressed naturally by CRM's locality of reaction feature. (5) It provides a framework for combining different programming technologies because no assumptions are made about the way of implementing each node in the system hierarchy. (6) The reusability of the agent systems can be promoted by higher-order CRM languages because the existing agents can be combined by using higher-order operations defined in those languages.

The presentation of our method will be in the following organization: In Section 2, we present a brief description of the Chemical Reaction Metaphor; In Section 3, we demonstrate the design of multi-agent systems in CRM by a case study in an e-learning system. Further design steps that lead to a concrete system are described in Section 4. Comparisons of our work to existing works are given in Section 5 and conclusions are drawn in Section 6.

2. The chemical reaction metaphor

Before we exploit the use of CRM in the specification of MAS, we need to consolidate the language we are going to use. Based on the computation model of CRM, the Gamma language [5, 6] was introduced to program the computation. In the Gamma language, parallelism is left implicit and therefore a Gamma program is a true natural parallel program. The Gamma language was found suitable for describing a distributed and/or evolving system consisting of distributed entities that execute and interact with one another asynchronously and that are added into the system or deleted from the system dynamically. The Gamma language successfully addresses the architectural design issues since its computation model captures the characteristics of a distributed system.

A Gamma program is composed of a set of rules governing the interactions among underlying program units in analogy to a reaction in chemical solution. Chemical solution is modeled by a multiset. For example, a list can be represented by multiset $M = \{(a, i) \mid a \text{ is value and } i \text{ an index and } i\text{'s are consecutive}\}$. Reaction rules are written in the form of an (R, A) pair where R denotes the condition of reaction and A the action when R evaluates to **true**. When a set of selected elements satisfies the reaction condition, they are replaced by another set of elements that are specified in the (R, A) pair. For example, given multiset M defined above, the following (R, A) pair replaces any ill-ordered pairs by two other pairs:

$$(a, i)\colon M, (b, j)\colon M \to (b, i)\colon M, (a, j)\colon M \leftarrow i < j \wedge a > b$$

It specifies that any two selected pairs (a, i) and (b, j) that satisfy the condition, $i < j \wedge a > b$ are replaced by two other pairs (b, i) and (a, j). No global control is imposed on the way multiset elements are selected to ignite the reaction. The execution of the program proceeds until no more reaction can take place, and the multiset at that point represents the result of the computation. For example, the following is a sorting program:

$$\begin{aligned} &\mathit{Sort}\, M_0 = [P, M = M_0] \text{ where} \\ &\quad P = (a, i)\colon M, (b, j)\colon M \\ &\qquad \to (b, i)\colon M, (a, j)\colon M \leftarrow i < j \wedge a > b \end{aligned}$$

where M_0 is the initial set of (*value*, *index*) pairs. When no more reactions can take place, the resulting multiset M represents the sorted set. In this program, $[P, M = M_0]$ is called a *configuration* which is composed of a (R, A) pair denoted by P and a typed multiset (also called environment variable) M. Here, we are using the higher-order Gamma notation presented in [7].

Higher-order Gamma is an extension of the Gamma formalism unifying the program and data syntactic categories, that is to say unifying multiset and (R, A) pair into a single notion of *configuration*. A configuration is made of a program and a record of named multisets. In the sequel, active program can be inserted into multisets and reactions can take place simultaneously in different levels. This extension much strengthens the expressiveness of the Gamma language. For example, the following is a producer-consumer program that is composed of two concurrent sub-programs. It takes the product of the first sub-program as the input to the second sub-program.

$$\begin{aligned}
Pc\,M_0 = {} & [P, M_1 = \{[Q_1, N_1 = M_{0,}\ N_2 = \Phi]\}, \\
& M_2 = \{[Q_2, R = \Phi]\}] \text{ where} \\
& P = [Q_1', N_1', N_2'] : M_1, [Q_2', R'] : M_2 \to [Q_1', N_1', \Phi] : \\
& \qquad M_1, [Q_2', R' + N_2'] : M_2 \leftarrow N_2{}' \neq \Phi \\
& Q_1 = \ldots \\
& Q_2 = \ldots
\end{aligned}$$

Although we omitted the definition of Q_1 and Q_2, which represent the first sub-program and the second sub-program, respectively, we assume that Q_1 operates on multiset N_1 and puts the result in multiset N_2. The higher-order rule P transfers the result in N_2 to R, which is the environment variable of Q_2. Operator $+$ is overloaded to represent the multiset union when its two operands are multisets. Similarly, operator $-$ is overloaded to represent multiset difference. Two operators, parallel composition $+$ and sequential composition $\circ$, are used to combine Gamma programs to form a compound program. Intuitively, let P and Q be two Gamma programs, then $P + Q$ is a program in which (R, A) pairs in P and Q work concurrently and $P \circ Q$ a program in which Q executes before P, i.e., P's (R, A) pairs are applied to the resulting multisets after no more Q's (R, A) pairs can cause a reaction. For a detailed description of the syntax and semantics of the Gamma languages please refer to [5–7].

In the above higher-order Gamma formalism, data and reaction rules are still distinguished. A more abstract formalism, γ-calculus [8, 9], is recently proposed as a unified model of chemical reaction based computation. Although γ-calculus' notation makes reasoning about a Gamma program easier, we use original higher-order Gamma syntax in our presentation because it is more apt to complex system specifications and eases the description of the program transformation we are to propose.

The Gamma language was widely used as a specification language in distributed systems. Its discipline makes it a distinguished language for architectural design in coordination programming [10], configuration programming [11], and software architecture [12, 13]. Some of the Gamma's applications in modeling distributed systems can be found in [14–16].

We found that the dynamic nature of distributed agents in e-learning environments makes it an ideal object for modeling by the Gamma languages. The concurrency and automation of agents require that the modeling language does not have any sequential bias or global control structure. In addition, the dynamic nature and non-determinism of interaction between an agent and its environment are suited to a computation model with a loose mechanism for specifying the underlying data structure. Therefore, the CRM provides a framework for the specification of the behavior of an agent. For example, data, which move around the Internet, can be well modeled by chemical solution; and mobile agents, which are created dynamically and transferred from clients to servers, can be included in the environment variable of a higher-order Gamma configuration. This provides a mechanism for describing inter-agent communications and agent migration. In the framework of higher-order Gamma, inter-agent communications and agent migration can be merged by viewing an inter-agent message as an inert program. For example, the following rule describes message transmission when M is a message or agent migration when M is a program:

$$M : E_1, M' : E_2 \to M + M' : E_2, \leftarrow C(M)$$

where E_1 and E_2 denote two environment variables (multisets), and $C(M)$ the condition. In a distributed system, E_1 and E_2 may represent the running environments of two processes (on the same computer or on different computers). The meaning of this rule is: select the element M and M' from environment variables E_1 and E_2, respectively. If the selected element M satisfies condition $C(M)$, M is removed from E_1 and added into E_2. Here, we adopt the "push" mechanism for data transfer/migration, i.e., the process is started by testing a condition at the source.

3. Specifying multi-agent systems in chemical reaction metaphor

As an active research area, the study in agent technology strives to apply intelligent information processing technologies to complex software systems. Although a precise definition of an agent system is yet to be given, features of an agent system have been summarized in the literature. According to Griss and Pour [17], an agent shows a combination of a number of the following characteristics: autonomy, adaptability, knowledge, mobility, collaboration, and persistence. These features exist in different types of agent systems such as collaborative agents, interface agents, reactive agents, mobile agents, information agents, heterogeneous agents, and economic agents [18]. Because of the Gamma language's higher-order operations and its closedness to specifications (no artificial sequentiality), these features can be described directly without being adapted to fit into proprietary frameworks. In [19], a sequence of case studies show that features of those different agent systems can be grasped by the Gamma language succinctly. In this Section, we give a comprehensive example of specifying a course material maintenance system using the Gamma language.

According to Flores and Lin [4, 20], the three elementary components identified as fundamental in the design of collaborative multi-agent systems are computer resources, agents, and owners. The architecture of the multi-agent system should be designed according to the workflow model of the educational tasks in distributed learning environments. From the workflow model of the course development, we can build a collaborative system model that partitions the problem into one or more smaller tasks, which are tackled by corresponding agents. A typical multi-agent system consists of the following agents:

1. user interface agents for different users in the system: program planner agents, course author agents, instructor agents, student agents, and tutor agents.
2. application agents such as: program planning agents, course generation agents, course maintenance agents, course recommendation agents, etc.
3. collaboration agents such as Local Area Coordinators and Conversation Managers.
4. knowledge management agents managing domain knowledge (ontology, concepts, etc.), knowledge about students, knowledge on tutoring, and knowledge about environment.
5. resource agents including course developer information agents, learning object directory agents, instructor information agents, tutor information agents, and student information agents. These information agents are responsible for getting information about resource needed.

One example of the multi-agent systems for course maintenance and recommendation in this paradigm was presented in [21]. In that system, online course materials, including all textbooks or e-books and study guides, plus the project handouts, can be downloaded and installed on the students' hard drive. Students only need to post material online on the conference, do the quizzes, or send/receive emails pertaining to the course. The online course materials are updated often in order to keep them as current as possible, especially in some rapidly changing fields such as computing and information systems. Because of the complexity of the materials, and the short development cycles within which they are produced, the course instructor should make the necessary adjustments from time to time for the benefit of the students. Whenever there is a significant change on the content of several designated web pages of online course materials, students who take the course should be notified by the course coordinator by e-mail.

The conversation model of the course material change notification consists of the following elements. For simplicity of illustration, we assume that a student who takes the course is in either of the 3 phases, numbered 1, 2, or 3. The interpretation of the phases is trivial and left undefined (for example, phase 1 might be the phase before the first exam, phase 2 the phase between the first exam and the second exam; and phase 3 the phase between the second exam and the final exam) except that we assume only students who have passed the previous phase are allowed to enter the next phase. A course web page also bears a phase number, indicating to which phase its content is significant. Once a change is made to a web page, all students taking the course and whose phase number matches the phase number carried by the web page will be sent the link pointing to that page.

Figure 1 shows the conversation schemata for course maintenance, which include four agents and two databases. Solid arrows denote control flow; dashed arrows denote agent migration; block arrows denote data flow. The agents included in the system are described below:

Registrar Agent (RGST): *RGST* adds a student into the student database or removes him/her from the database, or changes the phase number the student is currently in. Once there is an enrollment/drop action, *RGST* changes the student database (*STUD*) and signals student information agent

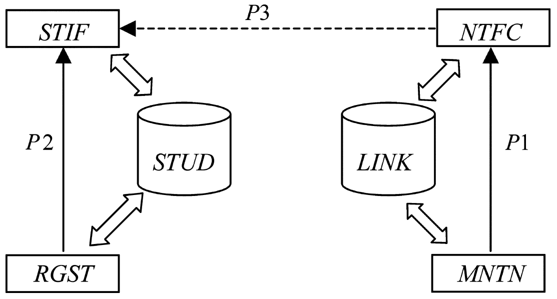

Fig. 1 A conversation schemata for course maintenance

(*STIF*). The action is denoted by $P2$ in Fig. 1 and in the following Gamma specification.

Student Information Agent (STIF): A Student Information Agent is designed for providing services about student information, such as providing an e-mail list for a course by automatically maintaining the email list of students taking a course; and maintaining the profile of each student.

Notification Agent (NTFC): The basic function of the Notification Agent is to send e-mails to students who take the course according to the student profiles stored in a database when the course material has been significantly changed. *NTFC* is a mobile agent, which migrates to *STIF* to perform the notification actions (denoted by $P3$). Since *STIF* has fast access to *STUD*, this measure eliminates the necessity for transfer of *STUD* data from *STIF* to *NTFC*. The volume of *STUD* data is typically large.

Maintenance Agent (MNTN): The maintenance agent provides proxy services to the instructor. It maintains the content of the topic tree, a course material URL database. When the agent detects a significant change, it sends a message to the Notification Agent *NTFC* (denoted by $P1$). Also, once a broken link is detected in the topic tree, it either corrects the link or deletes the orphaned page.

There are two databases used by this system:

Topic Tree or Link Database (LINK): The course material is organized in the form of a topic tree. Each entry in the topic tree is a link to a web page. Each entry of the link database is also a tuple (*link, phase, status*) where *link* is the link to the web page in the topic tree, *phase* the phase number this page is designed for, and *status* the status of the page, which can be either **normal, changed, or broken**. Note that in our notation constants are written in boldface words.

Student Information Database (STUD): Each student record is a tuple (*student, phase, mailbox*) where *student* is the name of the student, *phase* is the phase number where the student is in, and *mailbox* is the mailbox of the student, which is a multiset of email messages.

Let *INST* denote the multiset of instructor (we assume that there is only one instructor); and I, $S0$, and $L0$ denote the instructor, the initial roll of the class, and the initial content of the course (in the form of the set of links), respectively. The following is the Gamma program that specifies the above system. Note that following Gamma convention, if the reaction condition of an (R, A) pair is **true**, it is omitted.

$$\begin{aligned}
&MAIN\, i\; S0\; L0 = [P, RGST = \{[Q1, STUD = \emptyset]\},\\
&\quad NTFC = \{[Q2, STUD = \emptyset, LINK = \emptyset]\},\\
&\quad STIF = \{[Q3, STUD = S0, NTFC = \emptyset]\},\\
&\quad MNTN = \{[Q4, INST = \{i\}, LINK = L0]\}] \text{ where}\\
&P = P1 + P2 + P3\\
&\;P1 = [Q2, STUD = S, LINK = L + (l, p, \mathbf{normal})]\\
&\qquad : NTFC,\\
&\qquad [Q3, LINK = L' + \{(l, p, \mathbf{changed})\}] :\\
&\qquad\quad MNTN \rightarrow\\
&\qquad [Q2, STUD = S, LINK = L + (l, p, \mathbf{changed})]:\\
&\qquad\quad NTFC,\\
&\qquad [Q3, LINK = L' + \{(l, p, \mathbf{normal})\}] : MNTN\\
&\;P2 = [Q1, STUD = S]: RGST, [Q4, STUD = S',\\
&\qquad NTFC = N]: STIF \rightarrow\\
&\qquad [Q1, STUD = \emptyset] : RGST, [Q4, STUD = S' + S,\\
&\qquad NTFC = N]: STIF\\
&\;P3 = [Q2, STUD = S, LINK = L]: NTFC, [Q4,\\
&\qquad STUD = S', NTFC = N]: STIF \rightarrow\\
&\qquad [Q2, STUD = \emptyset, LINK = \emptyset]: NTFC,\\
&\qquad [Q4, STUD = S', NTFC = N + \{[Q2,\\
&\qquad STUD = S + S', LINK = L]\}]: STIF \leftarrow L \neq \emptyset\\
&\;Q1 = Enrl + Drop\\
&\qquad Enrl = (s, 1, \emptyset): STUD \leftarrow Enroll(s)\\
&\qquad Drop = (s, p, M): STUD \rightarrow (s, \mathbf{NULL}, M)\\
&\qquad \leftarrow Drop(s)\\
&\;Q2 = (l, p, \mathbf{changed}): LINK,\\
&\qquad (s, p \,.\, M) : STUD \rightarrow (l, p, \mathbf{changed}):\\
&\qquad LINK, (s, p \,.\, M + \{l\}):\\
&\qquad STUD \leftarrow l \notin M\\
&\;Q3 = Pass + Delete\\
&\qquad Pass = (s, p, M) : STUD \rightarrow (s, p + 1, M):\\
&\qquad STUD \leftarrow Pass(s, p)\\
&\qquad Delete = (s, \mathbf{NULL}, M) : STUD \rightarrow\\
&\;Q4 = AddInst + AddLink + Chng + Updt\\
&\qquad AddInst = i : INST \leftarrow AddInst(i)\\
&\qquad AddLink = i : INST \rightarrow (l, p, \mathbf{normal}) : LINK, i:
\end{aligned}$$

$$
\begin{aligned}
&\qquad \mathit{INST} \leftarrow (l,\,p) = \mathit{AddLink}(l,i)\\
\mathit{Chng} &= (l,\,p,\,\mathbf{normal})\colon \mathit{LINK},\, i\colon \mathit{INST}\\
&\quad \rightarrow (l',\,p,\,\mathbf{changed}) : \mathit{LINK},\, i : \mathit{INST}\\
&\quad \leftarrow l' = \mathit{Change}(l,i)\\
\mathit{Updt} &= (l,\,p,\,\mathbf{broken})\colon \mathit{LINK},\, i\colon \mathit{INST}\\
&\quad \rightarrow (l',\,p,\,\mathbf{normal})\colon \mathit{LINK},\, i : \mathit{INST}\\
&\quad \leftarrow l' = \mathit{Update}(I,i)
\end{aligned}
$$

Boolean functions *Enroll*(s) and *Drop*(s) return whether student s is enrolled in the class or wants to drop. *Pass*(s, p) function finds out whether student s has passed phase p or not. *Add*(l, i) function indicates whether instructor i wants to add a page pointed to by link l into the link database or not. *Change*(l, i) function returns the link to the changed page whose original is pointed to by l. *Update*(l, i) function updates the broken link l and returns the corrected link.

The program consists of configurations in two levels: the *MAIN* configuration in the higher level and all other configurations in the lower level. Program P in the *MAIN* configuration exchanges the elements of the multisets in the environments of the lower-level configurations.

For efficiency reasons, this program uses a mobile agent *NTFC*. The driver configuration transfers an *NTFC* configuration with updated *LINK* information (identified by condition $L \neq \emptyset$) to *STIF*, as follows:

$$
\begin{aligned}
P3 = {} & [Q2, \mathit{STUD} = S, \mathit{LINK} = L]\colon \mathit{NTFC},\\
& [Q4, \mathit{STUD} = S', \mathit{NTFC} = N]\colon \mathit{STIF} \rightarrow\\
& [Q2, \mathit{STUD} = \emptyset, \mathit{LINK} = \emptyset]\colon \mathit{NTFC},\\
& [Q4, \mathit{STUD} = S', \mathit{NTFC} = \mathrm{N} + \{[Q2, \mathit{STUD}\\
& \quad = S + S', \mathit{LINK} = L]\}]\colon \mathit{STIF} \leftarrow L \neq \emptyset
\end{aligned}
$$

instead of transferring *STUD* database to *NTFC*:

$$
\begin{aligned}
P3 = {} & [Q2, \mathit{STUD} = S, \mathit{LINK} = L] : \mathit{NTFC},\\
& [Q4, \mathit{STUD} = S'] : \mathit{STIF} \rightarrow\\
& [Q2, \mathit{STUD} = S + S', \mathit{LINK} = L] :\\
& \mathit{NTFC}, [Q4, \mathit{STUD} = S'] : \mathit{STIF} \leftarrow L \neq \emptyset
\end{aligned}
$$

Certainly transferring an agent is less costly than transferring a database. Therefore, this measure reduces the network traffic dramatically if the database is large.

This example shows how the Gamma language expresses the architecture of a multi-agent system succinctly. With the underlying computing model, we do not need to consider the specifications of nonessential features of the system, e.g., the number of program units, connection links for communications, and organizations of data, and therefore can focus on the specification of the overall architecture. It catches the way program units interact with one another and local computations, such as the implementations of those local functions, are left to the subsequent design phase.

The specification of the overall system benefits the subsequent design phases because details of the system can be added into the system in an accumulative fashion. The following section describes the specification of individual program unit.

4. From architecture to building blocks

Although there were discussions about implementing the Gamma language on parallel computers [22–24], it is commonly accepted that there is no straight implementation of the Gamma language that is efficient. After all, the Gamma language was designed as a very high level language for program specifications and is, therefore, used to specify the architectures of the coordinating systems, as described in Section 2. In the sequel, node-specific software design in a distributed system still relies on conventional software engineering methods. In a distributed multi-agent system, the separation of architectural design and concrete design on proprietary platforms is deemed even more necessary for dealing with the complexity of the system [21]. Therefore, we will restrict the following discussion to implementing the Gamma specification of the multi-agent systems in the architectural level with a minimum assumption about the computation model supported by the underlying system.

4.1. Computation model

The computation model on which we discuss the implementation of a Gamma specification is a multi-process system, in which processes are dynamically created and deleted and interacting with one another. No assumption is made about the allocation of the processes on distributed nodes of the underlying computing system. That is to say that multi-processes can run on a single node or on multiple nodes. The hierarchy of the multi-processes forms a tree structure, in which processes have full control about the creation/deletion of their descendent processes in the lower level. Communications among nodes are performed through communication channels which support unicast, multicast, and broadcast communications, which are supported transparently by the underlying network system. Moreover, mechanisms are provided to synchronize node activities.

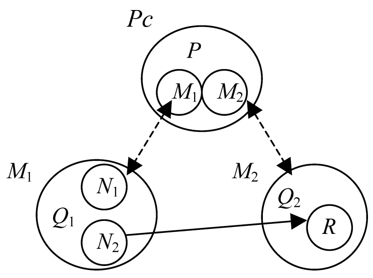

Fig. 2 Software architecture for the producer-consumer program

4.2. System architecture

A systematic design strategy based on the above computation model was proposed in [19]. In that approach, Gamma specification of an agent system is implemented in a hierarchical running environment composed of nodes in different levels of a tree. Interactions and synchronization among agents are implemented using a unified mechanism. Each configuration in the Gamma specification is implemented as a node type and the topology of the connections among nodes reflects the hierarchy of configurations in the Gamma program. As described in more detail in the following sections, a node type is a collection of nodes which form a multicast group, also called *module* in our module language. Therefore, the overall architecture of the system is a tree structure, which expands and shrinks dynamically. A node only communicates with another node in the immediate upper or lower level. Connections between nodes transfer data or status information that may cause an action in the upper level. The actions in the upper level (in which nodes are called control nodes) can create/delete nodes in the lower level or transform the states of nodes in the lower level by data transfer. For example, referring to the producer-consumer program in Section 2, whose first portion is repeated as follows:

$$\begin{aligned} Pc\, M_0 = [&P, M_1 = \{[Q_1, N_1 = M_{0,} N_2 = \Phi]\},\\ &M_2 = \{[Q_2, R = \Phi]\}] \quad \text{where}\\ P = [&Q_1', N_1', N_2']\colon M_1, [Q_2', R']\colon\\ &M_2 \rightarrow [Q_1', N_1', \emptyset]\colon M_1, [Q_2', R' + N_2']\colon\\ &M_2 \leftarrow N_2{}' \neq \Phi \end{aligned}$$

three nodes are created to represent the configuration which forms the main body of the program, the configuration in environment variable M_1, and the configuration in environment variable M_2, as illustrated in Fig. 2. Evaluation of reaction conditions (in this case, $N_2' \neq \Phi$) is done in the node (in this case, M_1) in the immediate lower level where the data reside. Dotted lines show the control channels, which are used for the lower level nodes to pass results of the evaluation of the condition to the upper level node, and for the upper level node to pass control signals to the lower level. The control signals cause creation/deletion of the lower level nodes. Solid lines show the data flow. In the control node, a handler is created for each environment variable to interact with the lower level nodes.

4.2.1. Synchronization

According to the Gamma computation model, the evaluation of the reaction condition and the action form a transaction that cannot be divided. Otherwise, concurrent execution of multiple (R, A) pairs may cause the data which are involved in the evaluation to evolve so that the bound action cannot take place. To ensure the atomicity of the transaction, a synchronization mechanism must be provided to block node activities until all the bound actions have been completed.

4.2.2. Multicast for the evaluation of reaction conditions

A multicast group is composed of all nodes that correspond to configurations in an environment variable (Note that the "multicast" here refers to soft multicast, viz., multicast among program units instead of computers). Multicast is used to transmit data for the evaluation of reaction conditions. In the above producer-consumer program, M_1 and M_2 are actually multicast groups (each containing only one node in this case), representing environment variable M_1 and M_2, respectively. Let's look at another program, which is an extension of the previous producer-consumer program:

$$\begin{aligned} Pmc\, M_0\, ID = [&P, M_1 = \{[Q_1, N_1 = M_{0,} N_2 = \Phi]\},\\ &M_2 = \{[Q_2, R = \Phi, F = \{id\}],\\ &id \in ID\}] \text{ where}\\ P = [&Q_1', N_1', N_2']\colon M_1, [Q_2', R', F']\colon M_2\\ &\rightarrow [Q_1', N_1', \Phi]\colon M_1, [Q_2', R' + N_2', F']\colon M_2\\ &\leftarrow N_2{}' \neq \Phi \wedge C(N_2', F') \end{aligned}$$

where ID is a set of identities of consumers. Therefore, the above program consists of one producer and multiple consumers. $C(N_2', F')$ is a condition that determines which consumer should receive the elements (products) in N_2'. Figure 3 shows the architecture of the above program. In Fig. 3, M_2 is a multicast group composed of the consumers. Whenever new elements are produced, they are multicast to all the consumers, each determines whether it is the receiver of the elements. The receiver adds the received elements

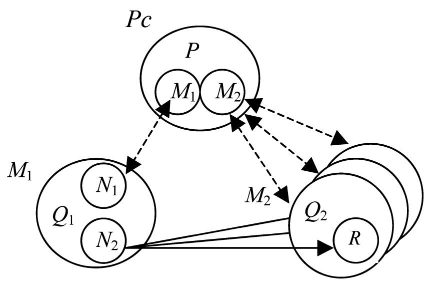

Fig. 3 Software architecture for the producer-multi-consumer program

into proper variables while other nodes discard the received elements.

Multicast ensures that all nodes of the same type (which corresponds to a configuration of Gamma program) be selected in a test of reaction condition. Termination detection can also be performed by using multicast to check whether further reaction can take place whenever there is a change in a node that could causes a reaction.

4.3. Node specification

Here we propose a language for specifying nodes that run on an execution environment that supports the above computation model. A process is a procedure type specifying a module and module is a class of nodes, which corresponds to configurations in the Gamma language. A process is composed of the process id, declarations of environment variables, imported variables, exported variables, and a body block consisting of sequentially executed statements.

```
process name(parameter-list)
environment
    Local environment variables
import
    Imported variables
export
    Exported variables
do
    condition-action pairs
od
```

Variables represent first-class environment variables, viz., multisets of first-class values. In our computation model, codes can be sent across nodes through environment variables. However, codes are still treated as first-class values since the execution of the code (including creating a runtime environment) is handled by the destination node. We leave the data structures for variables unspecified to maintain high-level abstraction. Their implementation is left to the implementation stage of each node, which is subject to proprietary platform technologies. Imported variables store values received from other nodes while exported variables stores the values that are sent to other nodes. Both imported and exported variables represent communication channels through which data are exchanged between nodes. Communication channels work in synchronous mode, i.e., data transmission does not occur until both sides are ready. Communication channels are full duplex. That means that we may use the same variable in both the import and export section. A parameter list is used to pass initial values to a node when it is created.

Note that communications are module based. Since a module is a group of nodes, communications among modules are one-to-multiple multicasts, i.e., when a node detects a condition that triggers communication, it sends the message to all nodes of the destination module. Although multicasts are primitive communication operations, unicasts are allowed by assigning ids to nodes and attaching an id to the message sent.

Operations performed by a process include local operations, communication operations, and process control operations. There are four local operations that can be performed by a process:

- Add(variable, data): add data into variable
- Delete(variable, data): delete data from variable
- Select(variable): select an element of the data set represented by variable. The selected element is returned by the function.
- element.#n: projection operation—extract the *n*th value of the tuple denoted by element

There are four communication operations (one is overloaded):

- Send(module, data): send data to all processes of module module. To allow for agent migration, the data parameter can be a configuration, which represents a program.
- Send(ID, data): send data to process with the given ID.
- Recv(module, data): receive data from any process of module module.
- Empty(module, ID): a Boolean function detecting whether there is any data sent from a process of module (module) to this process. The ID parameter returns the ID of the sending process. We allow the call to the Empty function using only the first argument, viz., in the form Empty(module), if ID is insignificant in the program.

There are five process control operations:

- Create(module, arg_1, arg_2, ..., arg_n): create a process of module (module) with the given argument list and returns the ID of the created process.
- Delete(module, IDs): delete processes whose IDs are specified in set IDs.
- Lock(module1, module2, ..., modulen): freeze local activities in all processes of module1 through modulen. Lock function does not affect communications among processes

in module1 through modulen. In addition, once a module is locked, further locking operation will have no effect.

- Unlock(module1, module2, ..., modulen): resume local activities in all processes of module1 through modulen. Symmetrically, once a module is unlocked, further unlocking operation will have no effect.
- Thread(config): create a thread, which runs the program represented by the config parameter. This feature is used to support mobile agents, codes sent by other processes and run on the environment of the process that receives it.

The body block of a module consists of a looping structure which has the following syntax:

```
do   cond1 -> statement1;
     cond2 -> statement2;
      ...
     condn: -> statementn;
od
```

Its semantics is: in each iteration, conditions are tested and one of the statements whose corresponding conditions tests to true is executed. This process is repeated until none of the conditions evaluates to true. This semantics is non-deterministic since no rule is set to govern how to select the statement to execute when multiple conditions are evaluated to true.
The branching statement has the following syntax:

```
if   cond1 -> statement1;
     cond2 -> statement2;
      ...
     condn: -> statementn;
fi
```

It is executed in the same way as do-od except that it is not repeated. If none of the conditions tests to true, the control falls through this if statement and continues to execute the statement that follows it.

The three modules designed for the producer-multi-consumer problem are in the following. Pmc is the module of the control node, which creates the producer node (of module M_1) and consumer nodes (of module M_2) and puts them in two multicast groups. The producer sends the data to consumers and signals (by sending a Boolean constant **true**) the control node at the same time, which freezes all local activities in both multicast groups. The consumers determine whether they are the targeted receivers and take proper actions if so and then signal the control node, which unlock both the producer and consumers.

```
process Pmc($N_1$ $M_0$, F ID)
environment
   multicast $M_1 = \Phi$, $M_2 = \Phi$;
   Boolean signal;
import
   Boolean signal;
do
   $M_1 = \Phi \rightarrow$ create($M_1$, $M_0$, $\Phi$);
   $M_2 = \Phi \rightarrow$ create($M_2$, $\Phi$, {id}) for each id $\in$ ID;
   !Empty($M_1$) $\rightarrow$ Recv($M_1$, signal), Lock($M_1$, $M_2$);
   !Empty($M_2$) $\rightarrow$ do
                    !Empty($M_2$) $\rightarrow$ Recv($M_2$, signal);
               od,
               Unlock($M_1$, $M_2$);
od

process $M_1$($N_1$ source$_0$, $N_2$ result$_0$)
environment
   $N_1$ source = source$_0$;
   $N_2$ result = result$_0$;
export
   $N_2$ result;
do
   result $\neq \Phi \rightarrow$ Send(Pmc, true), Send($M_2$, result);
   ... // the statements for $Q_1$
od

process $M_2$(R data$_0$, F id$_0$)
environment
   R data = data$_0$;
   F id = id$_0$;
   $N_2$ input = $\Phi$;
import
   $N_2$ input;
do
   !Empty($M_1$) $\rightarrow$ Recv($M_1$, input);
   C(input, id) $\rightarrow$ Add(data, input); Send(Pmc, true);
   ... // the statements for $Q_2$
od
```

The modules designed for the course maintenance program in the previous section can be found in Appendix A.

By removing higher-order operations in the module level, we make the specification of the system closer to actual program. Implementation of the program in the module language can be carried out fairly directly on a system that supports the computation model of the module language. Note that the implementation of local computations is out of the scope of this paper. It is left to the phase when the use of concrete language and platform are determined. We will rely on software engineering technologies for finding an efficient implementation of local computations. For example, further refinement of the specification should include the use of data structures to organize the data sets and implement the Select operation by an algorithm designed in accordance with the data structure.

4.4. Automatic transformation

The transformation from Gamma specification to module specification can well be automated. The translation is done on rule-by-rule basis. Each rule (viz., (R, A) pair) causes statements to be inserted into the do-od structure of the modules defining the immediate control node and the nodes involved in the reaction. The following is a general description of the translation process of a rule. Since each configuration is translated into a module, we use module in lieu of configuration in the description.

Step 1: Identify all modules involved in the rule. These are the modules that will be involved in the Lock/Unlock operation that is performed by the control node. Let l denote this set of modules.

Step 2: Identify all modules containing multisets that appear in the reaction condition (R) part of the rule. These are the modules that will be involved in communication. Let c denote this set of modules and $c \subseteq l$.

Step 3: For each module m in c, identify the multiset(s) that appear in reaction condition (R). Let $s(m)$ denote the set of these multisets.

Step 4: Focusing on a module m in c, declare a communication channel for each multiset in $s(m)$ in the export section of module m and the import section of each module in $l - \{m\}$.

Step 5: In the do-od structure of module m in c, use Select operation to detect any change in any multiset s in $s(m)$ and if so, signal the control node by sending boolean constant **true** and multicast s to every module in $l - \{m\}$ by using Send operation.

Step 6: In the do-od structure of each module n in l, apply Empty operation to each of the import channel to detect any message from other nodes and use Recv operation to receive the data.

Step 7: In the do-od structure of the control module, insert a statement that detects any signals from any module in c and lock all modules in l.

Step 8: In the do-od structure of each module n in l, insert a statement that checks the reaction condition and do the following:

a. If the reaction condition tests to false, send the control node a Boolean signal (**true**);
b. If the reaction condition tests to true, insert statement(s) that exchange data with other nodes using Send/Recv operation (global exchange), send a Boolean signal to the control node, and then insert statements that are described in Step 10.

Step 9: In the do-od structure of the control module, insert a statement that detects signals from each module in l and unlock every module in l when a signal is received from each node

Step 10: Perform the following operations according to the structure of the (R, A) pair:

a. For any module appearing on the left hand side of the transition rule (the A part):
   (i) if the same module appears on the right hand side of the transition rule, insert statements to modify the content of a multiset using Add/Delete operations and if there are any active elements (configurations) in the received data, create a new thread using thread operation.
   (ii) if the module does not appear on the right hand side of the transition rule, insert a Delete statement in the do-od structure of the control module to delete a node of that module (using node id as argument).

b. For any module appearing only on the right hand side of the transition rule, insert a Create statement in the do-od structure of the control module to create a node of that module (using multisets collected from the transition rule as arguments).

The above translation process can be implemented using a normal parsing technique with context-sensitive semantic rules. A more detailed description is presented in Appendix B.

5. Related works

A number of Architecture Description Languages (ADLs) have recently been proposed to cope with the complexity of architectural engineering. These include Rapide [25], Darwin [26], Aseop [27], Unicon [28], Wright [29] and ACME [30]. ADLs provide constructs for specifying architectural abstractions in a formal notation and provide mechanisms for reasoning about the architecture. They focus on defining architectural elements that can be combined to form a configuration. Few research efforts aim at truly defining an architectural description language for MAS architectures.

There are some formal languages proposed to address design issues of multi-agent systems. They focus on construct abstractions that can capture "social" behaviors of agents such as beliefs, desires, and intentions (BDI). Unfortunately, none of them is a complete formal system based on a finished computation model and they are still under development. For example, SkwyRL-ADL [31] is proposed as a BDI-MAS language to capture a "core" set of structural and behavioral abstractions, including relationships and constraints that are fundamental to the description of any BDI-MAS architecture. As it is still striving for a fully defined set of abstractions, it does not clarify the relationship of the architectural model to the underlying computation model and therefore does not serve as a language that encourages program design by derivation or transformation. Another attempt for addressing the architectural design of MAS is presented in [32] where a control theory based architecture for self-controlling software is developed. This model aims at a framework to accommodate formal methods for specification of agent functionality and inter-agent communication. However, no formal language is developed to facilitate the formal design. Instead, the author is attempting to use existing languages such as XML, DAML (DARPA Agent Markup Language), UML, and MOF to implement his vision.

6. Conclusions and future work

In this paper, we proposed a method for specifying a multi-agent system by using the Chemical Reaction Metaphor that is expressed with the Gamma language. We find that the architectural properties of a multi-agent system can be expressed succinctly and precisely in the chemical reaction metaphor: The non-determinism feature of the Gamma language fits well into concurrency; locality of reactions encourages modular design of MAS architecture; and closedness to specifications (no superficial sequentiality) couples with the distributed nature of MAS. Chemical Reaction language (Gamma) allows us to specify a complex system in succinct notations. We can express architectural behavior of the overall system while leaving operational details unspecified. We testify these assertions by a case study in which we demonstrate the applicability of this method in the design of a multi-agent based e-learning environment.

We also define a computation model that supports the implementation of Chemistry-inspired MAS specification. While this model supports straightforward implementation of functionality defined in the Gamma language, it removes all higher-order features of Gamma and bases all its operations on a set of primitives that are commonly supported by any networked computation sources. In this model, a program is composed of interacting nodes and communication among nodes is based on multicast. It also comprises mechanisms for process control (i.e., dynamic process creation/deletion) and multithreading.

Based on the above computation model, we present a method for transforming the Gamma specification of the agent system into the specification in a module language. By transforming the Gamma specification into the module language, we can remove higher-order multiset operations while allowing the advanced features of agent systems, such as mobility of agents, to be implementable. We use the multicast and process synchronization features of the underlying computation model to solve the problem in implementing a higher-order (R, A) pair. The use of multicast solves the problem in implementing the mechanisms for selecting configurations and testing reaction conditions in the higher-order (R, A) pair. Therefore, we solved the problem in implementing Gamma specification of agent systems in the network architecture level (inter-node level). Moreover, this method allows the implementation of computation at the intra-node level to be done using any program derivation strategy in any programming language. This paves the way for implementing the specified system by using a sequence of program transformations and provides an option for bridging the gap between specification languages and programming languages used in software engineering.

We have exploited the feasibility of an automatic translation and the development of the complete parsing system is underway. In addition, our future work includes a more precise definition of the interface between the architectural components (inter-node operations) and basic program units (intra-node operations).

Appendix A: The module Specification of the Course Maintenance Program

The modules designed for the course maintenance program in the previous section are described in the following:

```
process RGST(STUD firstRoll)
environment
   STUD roll = firstRoll;
export
   STUD roll;
do
   Enroll(s) → Send(MAIN, true);
    Send(STIF, {(s,1, Ø)});
   Drop(s) → Send(MAIN, true);
    Send(STIF, {(s,NULL, s.#3)});
od
process NTFC(STUD firstRoll, LINK origLink)
environment
   STUD roll = firstRoll;
   LINK link = origLink;
import
   LINK link;
do
   l = Select(link), l.#3 = changed →
     SEND(MAIN, true), Send(STIF, [P, {l}, Ø]) where
   P = do
     s = Select(roll), l ∉ s.#3 → Add(roll,
   {(s.#1, s.#2, s.#3 + {l})});
     od;
   !Empty(MNTN) → Recv(MNTN, link);
     Send(MAIN, true);
```

```
od
process STIF(STUD firstRoll)
environment
    STUD roll = firstRoll;
import
    STUD roll_n;
    NTFC mobile;
do
    s = Select(roll), Pass(s.#1, s.#2) →
        Delete(roll, s), Add(roll, (s.#1, s.#2 + 1, s.#3));
    !Empty(RGST) → Recv(RGST, roll_n); Add(roll,
      roll_n); Send(MAIN, true);
    s = Select(roll), s.#2 = NULL → Delete(roll, s);
    !Empty(NTFC) → Recv(NTFC, mobile),
      Send(MAIN, true),
       Thread([P, link, roll]) where mobile =
       [P, link, Ø];
od
process MNTN(INST initInst, LINK origLink)
environment
    INST inst = initInst;
    LINK link = origLink;
export
    LINK link;
do
    AddInst(i) → Add(inst, I);
    i = Select(inst), l = AddLink(l, i) → Add(link,
       (l.#1, l.#2, normal);
    i = Select(inst), l = Select(link), l.#3 = normal, l′=
Change(l, i) →
    Delete(link, l), Add(link, (l’, l.#2, normal)),
    Send(MAIN, true); Send(NTFC, {(l′, l.#2,
       changed)});
    i = Select(inst), l = Select(link), l.#3 = broken, l′ =
Update(l, i) →
            Delete(link, l), Add(link, (l’, l.#2, normal));
od
process MAIN(INST initInst, STUD firstRoll, LINK
origLink)
environment
    multicast RGST, NTFC, STIF, MNTN;
import
    Boolean signal;
do
    RGST = Ø → Create(RGST, firstRoll);
    NTFC = Ø → Create(NTFC, firstRoll, origLink);
    STIF = Ø → Create(STIF, firstRoll);
    MNTN = Ø → Create(MNTN, initInst, origLink);
    !Empty(RGST) → Recv(RGST, signal),
      Lock(RGST, STIF),
       Recv(STIF, signal), Unlock(RGST, STIF);
    !Empty(NTFC) → Recv(NTFC, signal),
      Lock(NTFC, STIF),
       Recv(STIF, signal), Unlock(NTFC, STIF);
    !Empty(MNTN) → Recv(MNTN, signal),
      Lock(MNTN, NTFC),
       Recv(NTFC, signal), Unlock(MNTN, NTFC);
od
```

Appendix B: Translation of an (R, A) pair

The following is a (R, A) pair in a configuration $[P, E_1{:}\ V_1, E_2{:}\ V_2, \ldots, E_n{:}\ V_n]$ of program M:

$$E_{i1} : V_{i1}, E_{i2} : V_{i2}, \ldots, E_{in} : V_{in} \rightarrow E'_{j1} : V_{j1},$$
$$E'_{j2} : V_{j2}, \ldots, E'_{jm} : V_{jm} \leftarrow C(E_{i1}, E_{i2}, \ldots, E_{in})$$

where *i1, i2,..., in*, and *j1, j2, ..., jm* are indexes in set $\{1, 2, \ldots, n\}$. $C(E_{i1}, E_{i2}, \ldots, E_{in})$ is the reaction condition. Each E_{il} can be either a configuration or a data item. Let $E(E_{il})$ denote the set of environment variables in E_{il} if E_{il} is a configuration or $\{E_{il}\}$ if E_{il} is a data item; and $E = \cup_{l=1}^{n} E(E_{il})$. Note that $E(E_{il})$ may include configuration elements if E_{il} is a configuration. Further assume that, in the resulting module program, the control node corresponding to program M is specified by module M, and each configuration in environment variables is specified by module V_{il} (if E_{il} is a configuration). Note that M and each such V_{il} are also multicast groups.

The test of condition $C(E_{i1}, E_{i2}, \ldots, E_{in})$ is done on each node involved in this reaction. Before testing the condition, all nodes do an exchange of data in set E. Data from other modules are sent to each node of a module using multicast. Also, during the transmission of data and performing of actions, all involved nodes must synchronize with one another, which is forced by the control node using the Lock() and Unlock() functions. This process is started by any node which detects a change in its environment variable.

Let C be the set of V_{il}'s whose corresponding E'_{il}s are configurations, and D the set of V'_{il}s whose corresponding E'_{il}s are data items. In addition, on the action side (viz., the expression $E'_{j1}{:}\ V_{j1}, E'_{j2}{:}\ V_{j2}, \ldots, E'_{jm}{:}\ V_{jm}$),let C' be the set of V'_{ik}'s whose corresponding E'_{ik}'s are configurations, and D' the set of V'_{ik}'s whose corresponding E'_{ik}'s are data items. The translator adds the following transition rules into the loop body of the specification of M module:

```
{For each $V_{il}$ ∈ $C$,
!Empty($V_{il}$, ID) → {for each $V_{il}$ ∈ C, Lock($V_{il}$);}
    {For each $V_{il}$ ∈ C, Recv($V_{il}$, $E_{il}$);}
    {For each $V_{ik}$ ∈ D, $E_{ik}$ = Select($V_{ik}$), For each
        $V_{il}$ ∈ $C$, Send($V_{il}$, $E_{ik}$);}
    Send(ID, C(E_{i1}, E_{i2}, ..., E_{in})),
    if
    C(E_{i1}, E_{i2}, ..., E_{in}) → {For each $V'_{ik} \in D'$,
        Add(V'_{ik}, E'_{ik});}
        {For each $V_{is} \in C - C'$, Delete
        (E_{ls}, ID of E_{is});}
        {For each $V_{il}' \in C' - C$, Create
        (E'_{il}, args of E'_{il});}
      fi
      {for each $V_{il} \in C$, Unlock(V_{il} );}
      }
    {For each $V_{ik} \in D$,
new(E_{ik}) → {for each $V_{il} \in$ C, Lock(V_{il});}
    {For each $V_{is} \in D - \{V_{is}\}$, E_{is} = Select(V_{is}),
    {for each $V_{il} \in$ C, Send(V_{il}, E_{is});}
    }
    {for each $V_{il} \in C$, Recv(V_{il}, signal);}
    if
    signal → {For each $V_{ik}' \in D'$, Add(V'_{ik}, E'_{ik});}
        {For each $V_{is} \in C - C'$, Delete(E_{ls}, ID of E_{is});}
        {For each $V_{il}' \in C' - C$, Create(E'_{il}, args of E'_{il});}
    fi
    {for each $V_{il} \in C$, Unlock(V_{il});}
}
```

For each $V_{is} \in C$, whenever a new element e_s is detected (we use new() function to denote the detecting process), the node starts the process of data exchange and condition evaluation. Also, once an element is received from another node, a new element is selected from local environment variable (denoted by $E(V_{is})$) and sent to other modules.

For each $V_{is} \in C - C'$, the translator adds the following transition rules into the loop body of the corresponding module V_{is}:

```
new(E_{is}) → Send(M, E_{is}),
    {for each $V_{il} \in C$, Send(V_{il}, E_{is});}
{For each $V_{il} \in C - \{V_{is}\}$,
!Empty($V_{il}$, ID) → E_{is} = Select(E($V_{is}$)),
    Send(M, E_{is});
    {for each $V_{il} \in C - \{V_{is}\}$, Recv(V_{il}, E_{il});}
    {For each $V_{ik} \in D$, Recv(M, E_{ik});}
    Send(ID, C(E_{i1}, E_{i2}, ..., E_{in}));
}
```

For each $V_{is} \in C \cap C'$, the translator adds the following transition rules into the loop body of the corresponding module V_{is}:

```
new(E_{is}) → Send(M, E_{is}), Recv(M, signal),
    {for each $V_{il} \in C - \{V_{is}\}$, Send(V_{il}, E_{is}),
      Recv(V_{il}, signal);}
    if
      signal → Delete(V_{is}, E_{is}),
        {Add(V_{is}, E'_{is}) if $E'_{is}$ is a data item ||
        Thread(E'_{is}, args of E'_{is}) if $E'_{is}$ is a configuration
      }
    fi
    {For each $V_{il} \in C - \{V_{is}\}$,
    !Empty($V_{il}$, ID) → E_{is} = Select(E(V_{is})),
      Send(M, E_{is});
    {for each $V_{il} \in C - \{V_{is}\}$, Send(V_{il}, E_{is});}
    {for each $V_{il} \in C - \{V_{is}\}$, Recv(V_{il}, E_{il});}
    {For each $V_{ik} \in D$, Recv(M, E_{ik});}
    Send(ID, C(E_{i1}, E_{i2}, ..., E_{in}));
    if
      C(E_{i1}, E_{i2}, ..., E_{in}) → Delete(V_{is}, E_{is}),
        {Add(V_{is}, E'_{is}) if $E'_{is}$ is a data item ||
        Thread(E'_{is}, args of E'_{is}) if $E'_{is}$ is a configuration
      }
    fi
}
```

Note that the above translation assumes that all $E_{i1}, E_{i2}, \ldots, E_{in}$ are variables and every constant is written as a logical expression E_{il}= *const* in the condition $C(E_{i1}, E_{i2}, \ldots, E_{in})$. Therefore, a (R, A) pair such as

$$Delete = (s, \mathbf{NULL}, M) : STUD \rightarrow$$

will be written as

$$Delete = \text{rec} : STUD \rightarrow \leftarrow \text{rec.\#2} = \mathbf{NULL}$$

However, this assumption does not influence the feasibility of the translation. We can enhance the translator to allow this flexibility. The process is trivial.

References

1. Yu E, Agent-oriented modelling: software versus the world. In: Agent-Oriented Software Engineering AOSE-2001 Workshop Proceedings. LNCS 2222. Springer Verlag, pp 206–225.
2. Paquette G (2001) Implementing a virtual learning center in an organization. In: Proc. of ITHET-2001, Kumamoto, Japan
3. Vouk MA, Bitzer DL, Klevans RL (1999) Workflow and end-user quality of service issues in web-based education. IEEE Trans. on Knowledge and Data Engineering 11(4):673–687
4. Flores RA, Kremer RC, Norrie DH (2001) An architecture for modeling internet-based collaborative agent systems. In Wagner T, Rana OF (eds) Infrastructure for agents, multi-agent systems, and scalable multi-agent systems. LNCS1887, Springer-Verlag, pp 56–63
5. Banatre J-P, Le Metayer D (1990) The gamma model and its discipline of programming. Science of Computer Programming 15:55–77

6. Banatre J-P, Le Metayer D (1993) Programming by multiset transformation. CACM 36(1):98–111
7. Le Metayer D (1994) Higher-order multiset processing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 18:179–200
8. Banâtre J-P, Fradet P, Radenac Y (2005) Principles of chemical programming. In: Abdennadher S, Ringeissen C (eds), Proc. of the 5th International Workshop on Rule-Based Programming (RULE'04), vol 124, ENTCS, pp 133–147
9. Banâtre J-P, Fradet P, Radenac Y (2004) Chemical specification of autonomic systems. In: Proc. of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)
10. Holzbacher AA (1996) A software environment for concurrent coordinated programming. In: Proc. of the 1st Int. Conf. on Coordination Models, Languages and Applications. Springer-Verlag, LNCS 1061, pp 249–266
11. Kramer J (1990) Configuration programming, a framework for the development of distributable systems. In: Proc. COMPEURO'90, IEEE, pp 374–384
12. Garlan D, Perry D (1995) Editor's Introduction, IEEE Trans. on Software Engineering, Special Issue on Software Architectures
13. Allen R, Garlan D (1994) Formalising architectural connection. In: Proc. of the IEEE 16th International Conference on Software Engineering, pp 71–80
14. Fradet P, Le Metayer D (1996) Type checking for a multiset language. INRIA Research Report
15. Le Metayer D (1998) Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering 24(7):521–533
16. Inverardi P, Wolf A (1995) Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Trans. on Software Engineering 21(4):373–386
17. Griss M, Pour G (2001) Accelerating development with agent components. Computer, IEEE, pp 37–43
18. Weiss M (2003) A gentle introduction to agents and their applications," Online presentation at http://www.magma.ca/∼mrw/agents/.
19. Lin H (2004) A language for specifying agent systems in e-learning environments. In: Lin FO (ed) Designing distributed learning environments with intelligent software agents, pp 242–272
20. Lin F, Norrie DH, Flores RA, Kremer RC (2000) Incorporating conversation managers into multi-agent systems. In Greaves M, Dignum F, Bradshaw J, Chaib-draa B (eds) Proc. of the Workshop on Agent Communication and Languages, 4th Inter. Conf. on Autonomous Agents (Agents 2000), Barcelona, Spain, pp 1–9
21. Lin FO, Lin H, Holt P (2003) A method for implementing distributed learning environments. In: Proc. 2003 Information Resources Management Association International Conference. Philadelphia, Pennsylvania, USA, pp 484–487
22. Creveuil C (1991) Implementation of gamma on the connection machine. In: Proc. Workshop on Research Directions in High-Level Parallel Programming Languages. Mont-Saint Michel, Springer-Verlag, LNCS 574, pp 219–230
23. Gladitz K, Kuchen H (1996) Shared memory implementation of the Gamma-operation. Journal of Symbolic Computation 21:577–591
24. Lin H, Chen G, Wang M (1997) Program transformation between Unity and Gamma. Neural, Parallel & Scientific Computations Dynamic Publishers, Atlanta 5(4):511–534
25. Luckham DC, Kenney JJ, Augustin LM, Vera J, Bryan D, Mann W (1995) Specification and analysis of system architecture using Rapide. IEEE Trans. on Software Engineering, pp 336–355
26. Magee J, Kramer J (1996) Dynamic structure in software architectures. In: Proc. of the 4th Symposium on the Foundations of Software Engineering (FSE4). San Francisco, CA, pp 3–14O
27. Garlan D, Allen R, Ockerbloom J (1994) Exploiting style in architectural design environments. In: Proc. of SIGSOFT'94: Foundations of Software Engineering. New Orleans, Louisiana, USA, pp 175–188
28. Shaw M, DeLine R, Klein DV, Ross TL, Young DM, Zelesnik G (1995) Abstractions for software architecture and tools to support them. IEEE Trans. On Software Engineering pp 314–335
29. Allen R, Garlan G (1994) Formal connectors. Technical Report, CMU-CS-94-115, Carnegie Mellon University
30. Garlan D, Monroe R, Wile D (1997) ACME: an architecture description interchange language. In: Proc. of CASCON 97. Toronto, pp 169–183.
31. Faulkner S, Kolp M (2002) (Isys), Towards an agent architectural description language for information systems. School of Management, the Catholic University of Louvain (UCL), Technical Report
32. Kokar MM, Baclawski K, Eracar Y (1999) Control theory-based foundations of self-controlling software. IEEE Intelligent Systems, pp 37–45