From Specification to Implementation: A Method for Designing Multi-Agent Systems in a Transformational Style

Hong Lin, University of Houston-Downtown, 1 Main Street, Houston, TX 77002, USA; E-mail: linh@uhd.edu

ABSTRACT

The suitability of using the Chemical Reaction Metaphor (CRM) to model multi-agent systems (MASs) is justified by CRM's capacity in specifying dynamic features of multi-agent systems. This paper presents a module language that facilitates a transformational method for implementing the specified multi-agent systems. A computation model with a tree-structured architecture is proposed to support the module language. The computational model is a straightforward abstraction of networked computing sources with minimum assumptions. In this model, the multicast network functionality pragmatizes the implementation of communications and synchronization among distributed agents. The transformational method is a rewriting process that translates the CRM specification into a program in the module language.

Keywords: Multi-agent systems, the chemical reaction models, program specification, very high-level languages, distributed systems, software architecture

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 (Yu, 2001). 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.

The modeling issue in the abstract computing machine level has been studied in (Banâtre, Fradet, & Radenac, 2004), where the chemical reaction model (Banatre & Le Metayer, 1990 & 1993, Banatre, Fradet, & Radenac 2005a, Le Metayer, 1994) is used to model an autonomic system. Given the dynamic and concurrent nature of multi-agent systems, we find that the chemical reaction metaphor provides a mechanism for describing the overall architecture of the 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 (Lin, 2004; Lin & Yang, 2006).

As pointed out in (Banâtre, Fradet, & Radenac, 2005b), however, a direct implementation of a CRM specification is unlikely to be efficient and the authors also pointed out that this is another exciting research direction. The difficulty in reaching an efficient implementation of CRM specifications is caused by the use of multisets as the basic data structures and that a direct implementation of the selection operations in the reaction rules requires a brute force testing of the data. We observe that implementation of CRM specifications in the system architecture level, e.g., the architectural specification of an MAS, and that in the programming level can be handled in different ways. By using network communication functions to facilitate reaction testing, we can implement the specifications without brute force testing. This implementation allows further refinement of node-specific programs using proprietary techniques.

The presentation of our method will be in the following organization: In the second Section, we present a brief description of the Chemical Reaction Metaphor; In the third Section, we describe the proposed method for implementing CRM specifications of MASs. Discussions and Conclusions are drawn in the last section.

THE CHEMICAL REACTION MODEL

Based on the computation model of CRM, The Gamma language (Banatre & Le Metayer, 1990 & 1993) 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 distributed entities that execute and interact with one another asynchronously and that are added into the system or deleted from the system dynamically. Follow-up researches revealed that the Gamma language can successfully address the architectural design issues since its computation model captures the dynamic characteristics of a distributed system (Inverardi & Wolf, 1995; Banatre & Le Metayer, 1996; Le Metayer, 1998). For instance, it is a distinguished language for the architectural design in coordination programming (Holzbacher, 1996), configuration programming (Kramer, 1990), and software architecture (Allen & Garlan, 1994; Garlan & Perry, 1995).

The basic term of a Gamma program is molecules (or γ-expressions), which can be simple data or programs (γ-abstractions). The execution of the Gamma program can be seen as the evolution of a solution of molecules, which react until the solution becomes inert. Molecules are recursively defined as constants, γ-abstractions, multisets or solution of molecules. The following is their syntax:

$M ::=$	$0 \mid 1 \mid \ldots \mid$ 'a' $\mid$ 'b' $\mid \ldots$	; *constants*
$\mid$	$\gamma P[C].M$	; *γ-abstraction*
$\mid$	M_1, M_2	; *multiset*
$\mid$	$<M>$	; *solution*

The multiset constructor "," is associative and commutative (AC rule). Solutions encapsulate molecules. Molecules can move within solutions but not across solutions. γ-abstractions are elements of multisets, just like other elements. They can be applied to other elements of the same solution if a match to pattern P is found and condition C evaluates to true and therefore facilitate the chemical reaction. The pattern has the following syntax:

$$P ::= x \mid P, P \mid <P>$$

where x is a variable. In addition, we allow for the use of tuples (written $x_1 : \ldots : x_n$) and names of types. For example, γ-abstraction

$$\gamma(x: \textbf{Int}, y: \textbf{Int})[x \geq y].x$$

can be interpreted as: replace x, y by x if $x \geq y$, which is equivalent to finding the maximum of two integers.

The semantics of γ-Calculus is defined as the following:

$(\gamma p[c].m_1), m_2$;*γ-conversion*	$= \phi m_1$	if match$(p/ m_2) = \phi$ **and** ϕc
m_1, m_2 ;*commutativity*	$= m_2, m_1$	

$m_1, (m_2, m_3) = (m_1, m_2), m_3$
; *associativity*
$E_1 = E_2 \Rightarrow E[E_1] = E[E_2]$
; *chemical law*

The γ-conversion describes the reaction mechanism. When the pattern p matches m_2, a substitution ϕ is yielded. If the condition ϕc holds, the reactive molecules $\gamma p[c].m_1$ and m_2 are consumed and a new molecule ϕm_1 is produced. $match(p/m)$ returns the substitution corresponding to the unification of variables if the matching succeeds, otherwise it returns **fail**.

Chemical law formalizes the locality of reactions. $E[E_1]$ denotes the molecule obtained by replacing holes in the context $E[\ ]$ (denoted by $[\ ]$) by the molecule E_1. A molecule is said to be *inert* if no reaction can be made within:

$$Inert(m) \Leftrightarrow (m \equiv m'[(\gamma p[c].m_1), m_2] \Rightarrow match(p/m_2) = \textbf{fail})$$

A solution is inert if all molecules within are inert and *normal forms* of chemical reactions are inert γ-expression. Elements inside a solution can be matched only if the solution is inert. Therefore, a pattern cannot match an active solution. This ensures that solutions cannot be decomposed before they reach their normal form and therefore permits the sequentialization of reactions. The following inference rule governs the evolution of γ-expressions:

$$\frac{E_1 \rightarrow E_2 \quad E \equiv C[E_1] \quad E' \equiv C[E_2]}{E \rightarrow E'}$$

For example, assume M, N, and R are names of three types, and f and g are two molecules that transform an element of M into an element of N, and from N into R, respectively, a producer can be defined as the following γ-abstraction:

$$prod = \gamma(x: M)[\textbf{true}].\ \langle f, x\rangle: N, prod$$

and a consumer can be defined as:

$$cons = \gamma(x: N)[\textbf{true}].\ \langle g, x\rangle: R, cons$$

A producer-consumer program that allows stream processing in which the producer and the consumer work concurrently can be written:

$$PC\ M_0 = \langle M_0, prod, cons\rangle$$

where M_0 is the initial set of values of M type.

IMPLEMENTING ARCHITECTURE SPECIFICATIONS

Although there were discussions about implementing the Gamma language on parallel computers (Creveuil, 1991; Gladitz & Kuchen, 1996; Lin, Chen, & Wang, 1997), 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. In a 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 the design on proprietary platforms is deemed even more necessary for dealing with the complexity of the system (Lin, Lin, & Holt, 2003). 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.

Computation Model

The computation model on which we discuss the implementation of a Gamma specification is a multi-process system, with processes 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 network. That is to say that multiple processes can run on a single node or on distributed nodes. The hierarchy of the multi-processes is a tree structure, in which processes have full control over the creation/deletion of their descendent processes in the lower level, but not vice versa. A process can "lock/unlock" its activities. Locking means the freeze of all local computations and unlocking is the reverse operation. However, locking/unlocking does not apply to communications between the manipulated processes and other nodes. Communications among nodes are performed through communication channels which support unicast and multicast communications. However, these communication functions are process based instead of IP address based. That is to say, for example, multicast involves a set of processes instead of a set of nodes with distinct IP addresses.

Module Specification

We propose a language for specifying processes that run on an execution environment that supports the above computation model. Processes are specified by *modules* in the module language. A module is composed of a name, a parameter list used to take initial data when the process starts running, and a body block consisting of sequentially executed statements.

```
module name(parameter-list)
begin
        statement-sequence
end
```

First-class values are stored in a data pool named pool(*mid*), where mid is the name of the module. We leave the data structures of the data pool unspecified to maintain high-level abstraction. Their implementation is left to the stage of coding in a concrete programming language, which is subject to proprietary platform technologies. We do assume, however, that data items are addressed in the data pool so that we can locate particular data items and delete them.

Each module is associated with two multicast groups: sync(*mid*) and dist(*mid*). sync(*mid*) is the multicast group used to implement atomic captures of molecules. It involves synchronization using *syn* and *ack* messages, as described above. dist(*mid*) is the multicast group for distributing produced molecules. We use *mid* as an argument to identify the multicast group to which *mid* belongs, such as sync(*mid*) and dist(*mid*). We may use sync or dist along if there is no ambiguity. Note that both sync and dist can be either syntactically (statically) or semantically (dynamically) checked. The syntactic check is used in this paper. We will show how to do the syntactic check in Section 3.4.

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

- Add(data): add data into the pool
- Delete(data): delete data from pool
- Select(): select a set of element in pool that may match the pattern. The set of selected elements is returned by the function if the selection is successful, or **fail** otherwise.
- Release(): release the selected elements and return them to pool.

two communication operations:

- Send(type, sid, data): send a message. type can be syn, desyn, ack, or dis. Semantics of Send operation differs with different types. A syn/desyn message is sent to all processes in multicast group sync, while an ack message is unicast to the sender of the syn message identified by sid. In addition, data field of the ack message is empty. A dis message is sent to all processes in multicast group dist and sid is insignificant.
- Recv(type, sids, data): Probe the message queue, return the first message (through parameters type, sids, and data), or **fail** (through function return

value) otherwise. Recv() function can return data sent by multiple senders identified by ids, e.g., Recv(ack, sync,) checks whether the syn message has been acknowledged by all processes in sync.

and five process control operations:

- Create(id, arg1, arg2, …, argn): create a module with the given name and argument list. The argument list is used to pass initial data set to the created module.
- Delete(id): delete the module whose id is specified in the parameter and collect the resulting multiset in pool of module id and join it with the local pool. Also, add id into the local pool. If module id is not inert, Delete(id) does not have effect and return immediately.
- Inert(): This is an overloaded operation. The default version (no argument) freezes local activities in the local module if no more actions can occur, and return **true**, or **false** otherwise; while the version with a module id (Inert(id)) tests whether module id is inert. Inert() function does not affect communications between the module and other modules. In addition, once a module is inert, a message is sent automatically to the solution module in the immediate upper level. If Inert() is called by a solution module, it automatically checks whether there are still active modules in the lower level by checking the received messages. A solution module will not become inert until all lower level modules are inert.
- React(): resume local activities in the local module. Also, once a module is re-activated, a message is sent automatically to the solution module in the immediate upper level.
- Thread(abs, arg1, arg2, …, argn) : create a thread, which runs the program represented by the abs parameter. This feature is used to support mobile agents, codes sent by other modules and run on the environment of the module that receives it. The mobile agent is run as a thread so that it shares the data in the execution environment of the receiving module. The result of the thread, if any, can be retrieved by <abs>.

The body block of a module consists of a sequence of statements separated by ";". A statement is either a call to the one of above operations, a conditional statement, or a looping statement. The following is the BNF definition:

block ::= begin *statement-sequence* end

statement-sequence ::= *statement* ; *statement-sequence* | empty

statement ::= *operations* | *conditional* | *looping*

conditional ::= if *reactions* fi

reactions ::= *reaction* ; *reactions* | empty

reaction ::= *cond* -> *actions*

cond ::= *actions*, *bool-expression*

actions ::= *action* , *actions* | empty

action ::= *statement*

looping ::= do *reactions* od

bool-expression ::= … ; an expression returning either **true** or **false**

operations ::= … ; the operations described above

The conditional statement has the following semantics: conditions are tested and one of the statements whose corresponding conditions test to **true** is executed. If none of the conditions tests to **true**, the control falls through the if statement and continues to execute the statement that follows it. Its semantics of the looping statement is: in each iteration, conditions are tested and one of the statements whose corresponding conditions test to **true** is executed. This process is repeated until none of the conditions evaluates to **true**. This semantics of the conditional and looping statements is non-deterministic since no rule is set to govern how to select the statement to execute when multiple conditions are evaluated to **true**.

Note that even the module language is still a high level specification language, e.g., it leaves data structures and underlying communication mechanisms unspecified and has nondeterministic control structures, it is a language based on the realistic computation model. No higher-order operations exist in a program in the module language.

For example, the three modules designed for the producer-consumer problem are in the following. PC is the solution module, which creates the producer module (prod) and consumer module (cons). In this particular program, sync is empty because the patterns of prod and cons do not intersect. Dist(prod) = {cons} and Dist(cons) = Φ, which means that the produced molecules are sent from prod to cons but not vice versa.

```
module PC(M0)
begin
   Create(prod, M0); Create(cons);
   do
      !Inert() ->                    ; polling the status of both prod and cons
   od
   Delete(prod); Delete(cons)
end

module prod(N)
begin
   do
      x: M = Select() -> Thread(f, x), Delete(x), Send(dist, cons, <f>);
   od
end

module cons()
begin
   do
      x: N = Select() -> Thread(g, x), Delete(x);
   od
end
```

The modules designed for the nth element program follow.

```
module nth(M0 n)
begin
    Create(sigma, M0);
    do
       Inert(sigma) -> Delete(sigma), Create(extr, pool, n);
    od
    Delete(extr);
end

module sigma(M)
begin
    do
       {(a, i): M, (b, j): M} = Select() and i < j and a > b
-> Delete((a, i): M, (b, j): M)), Add((b, i): M, (a, j): M);
    od
end

module extr(M, n)
begin
    do
       {(a, i): M, n: Int} = Select() and i = n -> Delete((a, i): M, n: Int),
    Add(a);
    od
end
```

The above module specifications have artificial simplifications that make them different from the module specifications obtained in an automatic transformation (depicted in Section 3.4). For example, in a reaction, there should be a test whether the sync group is empty before replacing elements. If the sync group is not empty, a syn message must be sent and acknowledged before performing the replacement. Similarly, after the reaction, there should be a test to the dist group.

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.

We refer the readers to (Lin, 2006) for case studies of the proposed method. (Lin, 2006) uses a higher-order Gamma formalism proposed by Le Metayer (1994). More current studies including case studies and applications in Gamma Calculus is being prepared for publication in the near future.

DISCUSSIONS AND CONCLUDING REMARKS

The chemical reaction models were proposed years ago to address high-level design issues of large distributed systems. Our work shows that it can be used to design multi-agent systems in a top-down fashion and benefits the design methodology with the reasoning capability of a formal system. While have shown that implementing such a system is feasible on a network computing environment, we would like to point out that our method can only be exerted to the module level, i.e., we can only derive the system to the specifications of module interfaces and operations. The implementation of individual modules will rely on conventional software engineering technologies. Further studies are needed to address the issues concerning module implementation and, perhaps, module abstraction, if we are to follow a "bottom-up" approach to design the multi-agent system, i.e., build the system on top of a set of existing functional units that coordinate over networks.

We present a method for implementing multi-agent system specifications in Gamma Calculus using a transformational style. Our existing work has demonstrated that Gamma Calculus, the newest formalism of the chemical reaction models, is suitable to describe high-level architectural properties of multi-agent systems and allows for systematic derivation and implementation of the systems. In this paper, we present a set of rules that can be used to derive the specified system into a module language, which is an intermediate language that do not include any higher-order operations as those in Gamma Calculus and is supported by most common networking execution environment. This study paves the way for implementing the specified system by using a sequence of program transformation and offers a new method for multi-agent system design.

ACKNOWLEDGMENT

This work is partially supported by the National Science Foundation award (grant# 0619312) and the U.S. Army Research Office Award (#W911NF-04-1-0024) through Scholars Academy of University of Houston-Downtown.

REFERENCES

Allen R., & Garlan, D. (1994). "Formalising architectural connection," Proc. of the IEEE 16th International Conference on Software Engineering, 71-80.

Banatre, J.-P., & Le Metayer, D. (1990). "The Gamma model and its discipline of programming," Science of Computer Programming, (15), 55-77.

Banatre J.-P., & Le Metayer, D. (1993). "Programming by multiset transformation," CACM, (36:1), 98-111.

Banatre, J.-P., & Le Metayer, D. (1996). "GAMMA and the chemical reaction model: Ten years after," In J-M. Andreoli, C. Hankin, and D. Le Metayer, editors, Coordination Programming: Mechanisms, Models and Semantics. Imperial College Press.

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).

Banâtre, J.-P., Fradet, P., & Radenac, Y. (2005a). "Principles of chemical programming," In S. Abdennadher and C. Ringeissen (eds.): Proc. of the 5th International Workshop on Rule-Based Programming (RULE'04), Vol. 124, ENTCS, 133-147.

Banâtre, J.-P., Fradet, P., & Radenac, Y. (2005b). "Higher-order Chemical Programming Style," In *Proceedings of Unconventional Programming Paradigms*, Springer-Verlag, LNCS, (3566), 84-98.

Creveuil, C. (1991). "Implementation of Gamma on the Connection Machine," Proc. Workshop on Research Directions in High-Level Parallel Programming Languages, Mont-Saint Michel, 1991, Springer-Verlag, LNCS 574, 219-230.

Garlan D., & Perry, D. (1995). Editor's Introduction, IEEE Trans. on Software Engineering, Special Issue on Software Architectures.

Gladitz, K., & Kuchen, H. (1996). "Shared memory implementation of the Gamma-operation," Journal of Symbolic Computation 21, 577-591.

Holzbacher, A.A. (1996). "A software environment for concurrent coordinated programming," Proc. of the 1st Int. Conf. on Coordination Models, Languages and Applications, Springer-Verlag, LNCS 1061, 249-266.

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.

Kramer, J. (1990). "Configuration programming, A framework for the development of distributable systems," Proc. COMPEURO'90, IEEE, 374-384.

Le Metayer, D. (1994). "Higher-order multiset processing," DIMACS Series in Discrete Mathematics and Theoretical Computer Science, (18), 179-200.

Le Metayer, D. (1998). "**Describing software architecture styles using graph grammars**," IEEE Transactions on Software Engineering, (24:7), **521-533**.

Lin, H., Chen, G., & Wang, M. (1997). "Program transformation between Unity and Gamma," Neural, Parallel & Scientific Computations, (5:4), Dynamic Publishers, Atlanta, 511-534.

Lin, F.O., Lin, H., & Holt, P. (2003). "A method for implementing distributed learning environments," Proc. 2003 Information Resources Management Association International Conference, Philadelphia, Pennsylvania, USA, 484-487.

Lin, H. (2004). "A language for specifying agent systems in E-Learning environments," in: F.O. Lin (ed.), Designing Distributed Learning Environments with Intelligent Software Agents, 242-272.

Lin, H., & Yang, C. (2006). "Specifying Distributed Multi-Agent Systems in Chemical Reaction Metaphor," The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies, Springer-Verlag, Vol. 24, No. 2, pp.155-168.

Yu, E. (2001). "Agent-oriented modelling: software versus the world," Agent-Oriented Software Engineering AOSE-2001 Workshop Proceedings, LNCS 2222, Springer Verlag, 206-225.