Visual Programming Environment for ECA Rules

Mónica Rivera de la Rosa, Oscar Olmedo Aguirre

Computer Science Section, Department of Electrical Engineering, Cinvestav
Av Instituto Politécnico Nacional 2508 México 07360 D. F.
mrivera@computacion.cs.cinvestav.mx, oolmedo@delta.cs.cinvestav.mx

Abstract. System software practitioners have largely recognized the difficulties of specifying complex program behavior in the design of distributed systems. In this paper, we present **Moon**, an experimental programming environment that uses UML sequence diagrams to describe and integrate the ECA rules that govern distributed systems behavior. The programming environment provides the means to interactively describe ECA rules and to visualize their effect. The rules are compiled into common concurrent programming abstractions including basic (i.e. send and receive) and structured (i.e. sequence and parallel) constructs to facilitate program execution and visualization. Among the contributions of this work, we emphasize the π-calculus foundation of the UML sequence diagrams which reduces the complexity of rule-based descriptions by introducing structuring notions of object encapsulation, sequential, parallel and conditional composition of simpler rules. We also believe that Moon may provide the means to develop applications ranging from algorithmic visualization, UML-based system development and also to prepare educational material of computer science courses centered on standard UML diagrams and rule-based reasoning

Keywords. UML, Visual Programming, Rule-based Programming, Programming Environment

1 Introduction

The design and construction of concurrent programs in a distributed setting has been largely recognized a complex undertaking. In this respect, rule-based programming has been proposed to better our understanding of complex program behavior. Rule-based specifications are used in reactive systems that are monitored for the occurrence of events that may signal critical conditions. In rule-based systems, once an event is detected, if the event parameters satisfy a condition, a specified action is performed to handle the situation. The language construct that corresponds to this model of interaction is called *ECA-rule*. The use of ECA-rules in distributed systems arise from the fact that a distributed system is a reactive system whose behavior can be conducted by the messages received (events) and by the messages sent as response (actions). ECA-rules are the basis of ADM, an Active-Deductive Model that uses XML as data representation and message exchange format in distributed applications [11]. ADM is a

© *A. Gelbukh, C. Yáñez Márquez, O. Camacho Nieto (Eds.)*
Advances in Artificial Intelligence and Computer Science
Research on Computing Science 14, 2005, pp. 253-264

programming language aimed to coordinate software agents with proactive, rational and social behavior. Tough ADM-rules constitute a powerful programming paradigm, their interactions are very often difficult to understand because they are generally not designed to be applied in a structured or hierarchical manner.

With the purpose of making rule interactions easier to understand, in this paper we propose to use UML sequence diagrams to visualize rule definition, selection and execution. We believe that rule definition can be easily understood by means of a short UML sequence diagram that expose the event-condition-action structure of the rule. Furthermore, rule selection and instantiation becomes more coherent as the collection of sequence diagrams are identified by the events they handle. Finally, rule execution may become more understandable because rules are grouped around objects in a sequential or parallel manner.

Unfortunately, the programming environments that have adopted UML diagramming as the basis for its design methodology are used at best for the generation of program skeletons in programming languages like Java and C++ because UML diagrams cannot be directly executed. Although replacing the textual representation of a programming model for a visual one represents a substantial step in abstracting algorithm design from program coding, program behavior still cannot be visualized directly from UML diagrams as the program execution is not integrated into the environment. On the other hand, algorithm visualization systems do not use standard UML diagrams to visualize program behavior.

In this work, we propose the construction of a programming environment that addresses the aforementioned problems. By integrating the visualization of program edition and execution, the programming environment offers a number of outstanding advantages from previous work:

- Priming algorithmic design over program coding.
- Abstracting algorithm description from programming languages and platforms
- Visualizing program behavior using standard, widely known and accepted UML diagrams

Next, we outline the content of this paper. In section 2, we compare our proposal with some related work in the areas of program visualization and UML diagramming tools. In section 3, we briefly describe the Moon programming environment including its overall architecture. In section 4, we present the visual and textual forms of the programming language along with their formal specification. This section also shows the translation schemes defined between the representations. In section 5, we informally present the subset of the π-calculus used in this work as a specification language and also discuss how it is used to describe ECA rules. In section 6, we comment about the translation schemes, and finally, in section 7, we present our concluding remarks.

2 Related Work

Knowlton's video "Sorting out Sorting" was probably the first dynamic animation of a data structure algorithm produced in 1981. Since then, a number of algorithm visualization systems have been developed. According to their interaction, they can be classified in two groups. The first group allows only algorithm visualization with no

user interaction (BALSA[2], TANGO[2], JAWAA[1], GAIGS[2]), while the second group characterizes by their pleasant user guidance (Leonardo Web[1], ALICE[3]). Both groups use available Web technology to display scenarios ranging in sophistication. However, none of them use standard UML diagrams to describe the algorithmic concepts involved.

Visualization of concurrent programs is more complicated than visualization of sequential programs, due to the presence of multiple threads that communicate, compete for resources, and periodically synchronize. The misunderstanding of concurrent programs behavior may result in unexpected interactions and non-deterministic executions. Some visualization tools have been realized to overcome these issues. In the Gthreads library [16], a program graph is built as threads are forked and functions are called, where the vertices of the graph represent program entities and events, while the arcs represent temporal orderings between them. Message passing views are supported by the Conch system [17] where processes appear outside of a ring and messages are exchanged among them by traversing the ring. In this way undelivered messages can be detected, as they remain within the ring. The Hence system [16] offers animated views of the program graph obtained from execution of PVM programs. Once again, none of them use standard UML diagrams to describe process interactions.

There are also currently available a number of CASE (Computer Aided Software Engineering) tools that facilitates the edition of UML diagrams. However, once created the diagrams, there are no means to completely transform the model into an executable program code whose results can in turn be visualized in the diagrams (Argo UML[9], Rational Rose[12], Magic Draw). Despite the fact that Flash from Macromedia has been largely used in Web-based applications, it does not conform to the WWW standards. After recognizing the benefits of using vector graphics, the WWW consortium launched an initiative to define the SVG (Scalable Vector Graphics) dialect of XML [8]. However, both SVG and UML have independently been defined in separate standardization efforts.

3 The Visual Programming Environment Moon

With the aim of constructing a programming environment for a direct visualization, we advocate to the design of the main modules of the environment:

- UML sequence diagram editor intended to verify that the UML diagram is well formed, producing in such case meta-data about the model. It is built upon the available Internet browsers and enabled by Adobe SVG plug-ins by means of scripting.
- Front-end compiler intended to translate UML diagrams into ADM rules, as explained in section 5. From the meta-data produced by the editor, the front-end compiler uses translation schemes to transform the graphic and textual annotations of the UML sequence diagram into the constructs of the ADM rule language.
- Back-end compiler aimed to translate ADM rules into Java programs. ADM rules are transformed into executable Java programs according to the intended meaning of the model.

- Run-time mediator aimed to coordinate program execution with the diagram editor. The mediator consists of a minimal coordination and communication infrastructure between the viewer layer and the run-time JVM that executes the program. During the program execution, the mediator synchronizes both layers to exchange messages containing the data produced by the running program and by the user interactions received through the viewer.
- Program viewer intended to display program execution as UML diagrams. The viewer receives the incoming data and transforms them into SVG graphical elements to be rendered in the Web browser.

The relationships among the components are shown in Fig. 1.

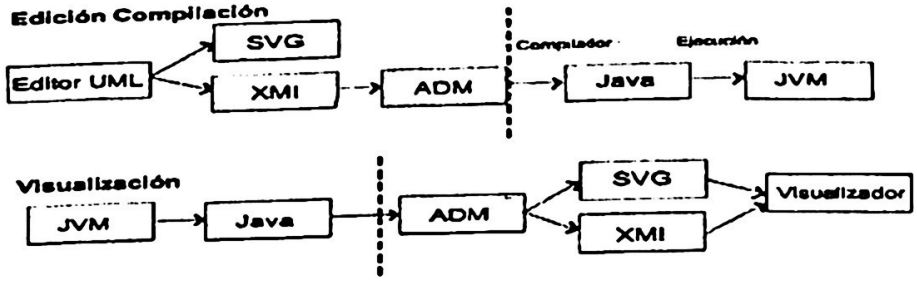


Fig. 1. Architecture of the Moon programing environment

The graphical editor, written in JavaScript, transforms the elements of the UML sequence diagram into SVG elements. Being an XML dialect, the SVG representation of diagram brings in the available standard tools (SAX or DOM-based) to store them in a stable storage or transfer them through a network. In particular, a DOM API is used to compile de diagrams into ADM rules by means of XML transformations. Fig. 2 shows a view of the UML editor.

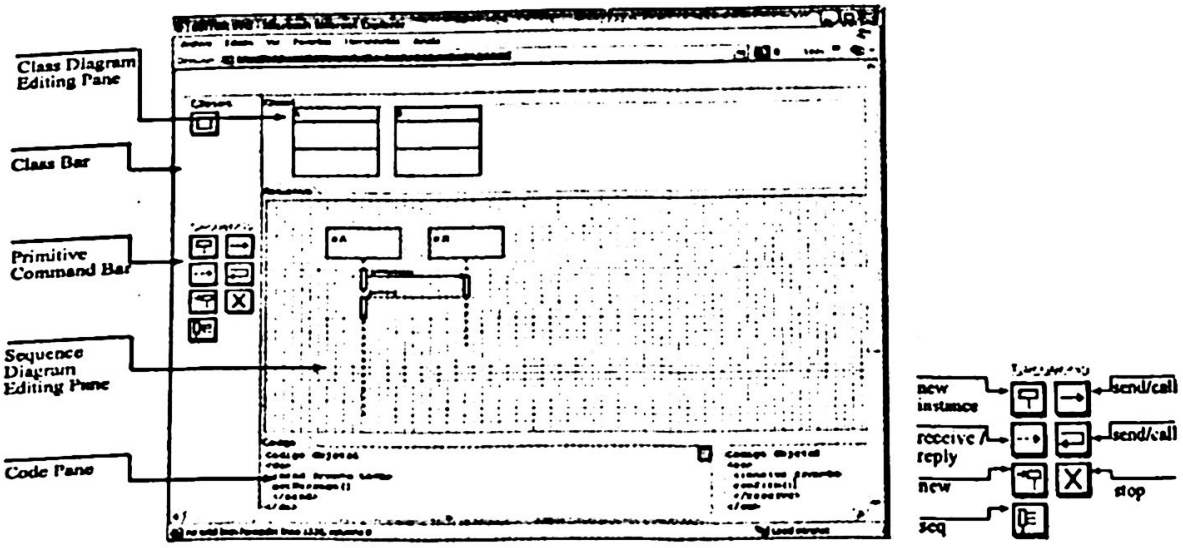


Fig. 2. Moon Programming Environment

The Moon programming environment is organized in four panes: action buttons, class diagram editing, sequence diagram editing, and ADM code generation. The action button pane contains control buttons for introducing the graphical elements for each element of the language into the class diagram and the sequence diagram panes.

A typical session in the Moon programming environment follows a three-phase cycle: graphical edition, code generation and program execution. Graphical edition begins by placing the classes of all participants in the class diagram pane, whose instances are in turn placed in the sequence diagram pane. Then, the interactions and collaborations of participants are edited according to their roles and the intended behavior of the system. Code generation begins after concluding the edition phase: the UML diagrams produced are compiled into ADM rules according to the translation schemes explained in section 4. Then, a π-calculus specification is generated from the ADM rules produced. The π-calculus specification is interpreted upon a virtual machine written in Java. The program execution phase is responsible of running the program on the coordination infrastructure provided by the virtual machine. During the program execution, the mediator helps to show the results back into the UML diagrams edited. The programming session may repeat this cycle until the program specification can be visually validated from the UML diagrams.

In the next section, we describe the three forms of representation used in the environment along with the translation schemes that related them.

4 Visual and Textual Forms of the Programming Language

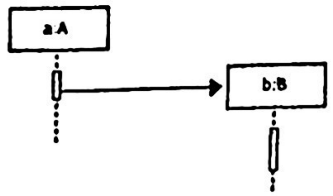
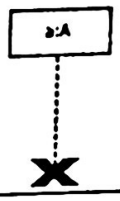
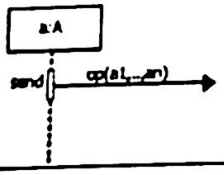
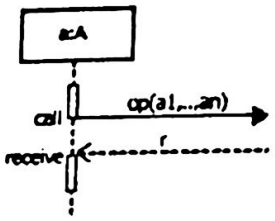
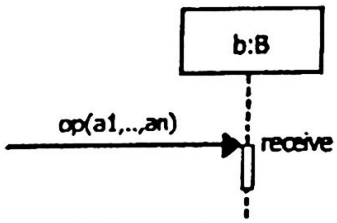
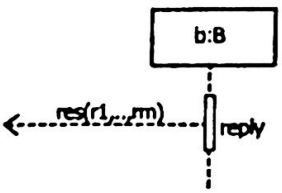
Sequence diagrams describe the behavior of a system developed by the interaction of the participants. These diagrams contain objects (classes) that exchange messages arranged in a time sequence. They are defined by identifying the following elements [15]:

1. Class roles, denoted by rectangles surrounding the role-name and the class-name, specify the type of objects that may participate with interactions and collaborations
2. Lifelines, denoted by vertical dashed lines, represent the existence of class roles over a period of time, since the object is created until it is destroyed. They may split into two or more concurrent lifelines to show concurrency or conditionality, each lifeline corresponding to a thread or a conditional branch and they may merge together at some subsequent point.
3. Activations, denoted by thin vertical rectangles arranged along lifelines, represent the time during which a class role is performing an action (operation) or when it is active and has focus of control.
4. Messages, denoted by labeled horizontal arrows between lifelines, define the information content of a communication that is exchanged in interactions and collaborations. The message instance has a sender, a receiver and possibly other information according to the characteristics of the request. Messages may be either synchronous (solid arrow heads), with explicit or implied return (stick arrow heads), or asynchronous (half stick arrow heads).

A concurrent programming model provides meaning to the graphical elements of a UML sequence diagram by means of the notion of process. *Process* represent independent threads of control, each of which executes sequential code [14]. A process is an active object that in addition of developing its own behavior (by executing its code), it may provide services to other objects by accepting calls to the methods the active object offers. Processes can be created dynamically and can communicate with

each other by means of a variety of mechanisms: message passing, local and remote method calls and rendezvous.

In the textual form of the programming language, a process is described by the abstract syntax shown in Fig. 3.The figure consists of a three-column table that relates a UML diagram fragment with the corresponding ADM rule and π-calculus expression.

	<var name="b"> <new> <B> <p1>a_1 </p1>... </B> </new> </var>	$b : B(a_1,...,a_n)$ (create and initialize object b of class B)
	<stop/>	stop (cease object activity)
	<send to="b"> <op> <p1> a_1 </p1>... </op> </send>	b ! $op(a_1,...,a_n)$ (send message op to object address b)
	<call to="b"> <op> <p1> a_1 </p1>... </op> </call> <receive from="b"> r </receive>	b ! $op(a_1,...,a_n)$; b ? r (send message op to object b, then wait for response to be bound to r)
	<receive from="a"> <op> <p1> a_1 </p1>... </op> </receive>	a ? $op(a_1,...,a_n)$ (receive message op at object b)
	<reply to="a"> <res> <p1> r_1 </p1>... </res> </reply>	a ! r (send response back to the caller process and resume execution)

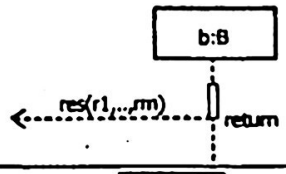
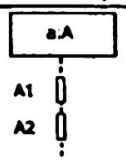
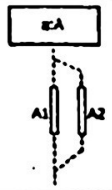
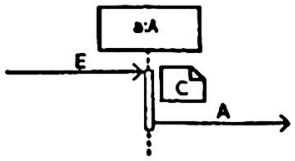
	<return to="*a*"> <res> <p1> r_1 </p1>... </res> </return>	a ! r ; stop (send response to the caller process before process normally terminates)
	<seq> A_1 A_2 </seq>	A_1 ; A_2 (perform A_1 then perform A_2)
	<par> A_1 A_2 </par> <alt> A_1 A_2 </alt>	$A_1 \mid A_2$ (perform both A_1 and A_2) $A_1 + A_2$ (perform either A_1 or A_2)
	<rule> <on> E </on> <if> C </if> <do> A </do> </rule>	b ? E; C; A (after receiving message E, test if its contents satisfies condition C and if it does, perform action A; otherwise, abort)

Fig. 3. Textual and graphical language constructs.

The ECA rules generated from the editor are then compiled into a Java program. The code is generated according to the following programming model that describes in a precise and unambiguous manner, the meaning of each programming notion.

5 Programming Model

The programming model provides meaning to the syntactical constructs shown before. It is inspired in the π-calculus[13] whose abstract syntax is shown in Fig. 4. The operational semantics of the programming model is presented in this section by means of transition rules.

P	::=	Stop	normal process termination
	|	Abort	abnormal process termination
	|	c ? x	receive from channel c a value to be bound to x
	|	c ! v	send value v through channel c
	|	new c	create an unique channel named c
	|	P_1 ; P_2	perform P_1 and upon its termination perform P_2
	|	$P_1 \mid P_2$	perform concurrently P_1 and P_2
	|	$P_1 + P_2$	perform only one of P_1 or P_2

Fig. 4. Abstract syntax of process expressions in the π-calculus

As in the π-calculus, a notion of address space is rendered by a collection of process accessed through dynamically created channels. Thus a channel identify uniquely a process instance (address of an active object) and can be used as the unidirectional communication medium through which the process instance receives requests for the operations it supports and sends back the responses.

Process definition. Processes are defined by means of equations. In the programming language, a typical process definition has the form $P(c,v) = E$, where P is the class constructor with communication channel c, list v of initial values and process expression E describing the object behavior. For example, a typical process definition is:

$$P(c,v) = c?m_1(x_1); P_1 + ... + c?m_k(x_k); P_k$$

where c denotes the address of the process instance and v denotes the (possibly empty) list of initial values of the instance. The right-hand side displays the set of k methods that the process supports. Method $c?m_i(x_i); P_i$ is called when message $m_i(x_i)$ is received at channel c. Method body P_i is performed possibly returning a message as result of the invocation.

Although process are defined by equations, in practice they are used as left to right rewrite rules. However, we prefer to maintain the equational form of processes definitions to clearly distinguish them from the transition rules of the operational semantics. Process definitions are used in the instantiation of processes. However, in order to avoid uncontrolled creation of process instances, they are created on demand when a channel name is passed to the class constructor, as explained next.

Process creation. An instance of class A is created by sending a new communication channel a to the class constructor:

$$a :A(v) \rightarrow \text{new } a; A!a; a!m(v) \mid A?c^*P(c,u)$$

$$P(c,v) = c?m_1(x_1);P_1 + ... + c?m_k(x_k);P_k$$

In the left-hand side of this transition ($a :A(v)$), the channel name a is created as the communication medium with the process instance. Name A denotes not only the class name but also (by abuse of notation) the channel name where the class constructor accepts requests for creating a new instance with a list v of initial values.

In the right-hand side of the transition appears the π-calculus expression that actually creates the process instance. The channel a is created by the new operator (new a), ensuring the uniqueness of a. Process creation uses the guarded form of the replication operator (i.e. $A?a * P$, where P represents the expression in parenthesis). Its purpose is to repeatedly create an instance $P[c/a]$ of P (obtained from the substitution of c by a in P) only after a channel name is received at the class constructor channel A, according to the transition:

$$A!a \mid A?c * P \quad P[c/a] \mid A?c * P$$

As the transition shows, the guarded replication expression $A?c * P$ is left unchanged to create new instances. The parallel composition operator | (bar) denotes the concurrent execution of the process instances.

Finally, the expression $c?m_1(x_1);P_1[c/a] + ... + a?m_k(x_n);P_k[c/a]$ consists of the sum of k method definitions $P_i[c/a]$ guarded by the corresponding method invocation (message reception) $m_i(x_i)$ containing a list x_i of parameters. A sum of k processes denotes the selection of exactly one process for execution (after applying substitution $[c/a]$). Method definition $P_i[c/a]$ is selected only after receiving method invocation $m_i(v)$ along with its parameter list v, according to the transition:

$$a!m_i(v);\ Q \mid a?m_1(x_1);\ P_1[c/a] + ... + a?m_i(x_i);\ P_i[c/a] + ... + a?m_k(x_k);\ P_k[c/a]$$
$$Q \mid P_i[c/a,\ x_i/v_i]$$

Because data flows from sending $a!m_i(v)$ to receiving $a?m_i(x_i)$ expressions, sums are restricted to contain only receiving expressions to simplify program language design. It is important to remark that each process instance has its own copy of the methods to guarantee that each instance evolves independently and that no conflict may arise from the data they hold.

Variables. Processes in the same address space can communicate to each other by sharing variables. Variables can be seen as processes that only accept operations *get* and *set* to retrieve and modify, respectively, the value that a variable holds. In the transition:

$$x : Variable(v) \quad \text{new } x\ ;\ Variable!x\ ;\ x!set(v) \mid Variable?x\ {}^*Var(x,z)$$

after creating name x (new x), the name is sent to the process constructor of class *Variable* (*Variable*!x) and then its initial value is set to v ($x!set(v)$).

The guarded replication *Variable*?x **Var*(x,z) creates variable *Var*(x,z), after receiving the channel name of a new variable at the process constructor (*Variable*?x). Note that though the starting value z of a variable is always undefined, it is immediately replaced by the initial value v received from action $x!set(v)$.

The behavior of *Var*(x,v) is defined by equation:

$$Var(x,v) = x?get();\ x!v;\ Var(x,v) + x?set(y);Var(x,y)$$

that precisely defines the methods that variables offer.

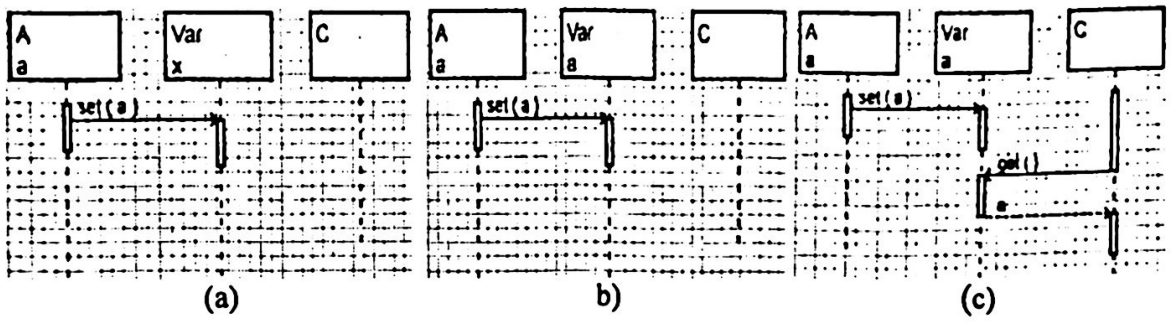


Fig. 5. Variable Example

Although, the Moon environment does not display the channel associated with the objects created, it is handled internally a made explicit in the process expressions. Fig. 5 shows a sequence of three screenshots taken from a Moon programming session of

objects of classes A and C that interact with object $Var(x,z)$. As shown in Fig. 5a, if variable $Var(x,z)$ receives message $set(a)$ from object A, incoming value a is bound to y and $Var(x,z)$ becomes $Var(x,a)$, holding new value a which will be used in further requests. Fig. 5b shows how the Moon environment immediately updates the box of the variable with the new value. Then, as shown in Fig 5c, when $Var(x,z)$ receives message $get()$ at x from object C, a is returned to C, and then $Var(x,v)$ behaves as before.

Method invocation and rendezvous. Though in a method invocation the caller process and the called process may reside on the different machines, the called process creates an attending process to execute the method code for the invocation, while the caller is delayed for the response. Once the invocation is completed, the caller process resumes its execution, while the attending process is destroyed. According to this behavior, method definition begins receiving the message including the operation name and its parameters and terminates before sending the response back to the caller. The following expression sketches the receive-send pair described:

$$P = c\,?\,m(x)\,;M\,;c\,!\,r + \ldots$$

where, m is the method head and M is the method body. In the caller side, the complementary pattern required consists of sending the requests and then waiting for the response:

$$P = c\,!\,m(v)\,;c\,?\,x\,;\ldots$$

There is some flexibility on the mechanisms offered in the way results are passed back to the caller process. In the remote method invocation approach the attending process is immediately destroyed after sending back the results:

$$P = c\,!\,m(v)\,;c\,?\,x\,;\,\text{stop}$$

whereas the randezvous approach there is no attending process so, in general, it cannot be destroyed.

Event-driven behavior. Simple ADM rules can be encoded in the π-calculus. Process receiving messages invoking operations constitute basic events (E). Testing logical conditions upon message content, including equality, arithmetic relations and usual logical connectives, correspond to condition checking (C), whereas sending messages correspond to the actions performed (A). These three elements are composed in the sequence $b\,?\,E;\,C;\,A$. When evaluated a condition, it is reduced either to skip if the condition is true or to fail if the condition is false, determining the flow of control in the process, according to the transitions:

$$\text{skip};A \quad A \qquad\qquad \text{abort};A \quad \text{abort} \qquad\qquad \text{abort} + A \quad A$$

The last transition selects only one course of control among many possible if all of them have mutually exclusive conditions. For example, in a sorting algorithm, a process $S(m, c, d)$ in a chain of processes always keeps the minimal value:

$S(m, c, d) = c\ ?\ x\ ;\ (x < m;\ d\ !\ m;\ S(x, c, d) + x'\ m;\ d\ !\ x;\ S(m, c, d))$

Being both conditions $x < m$ and $x'\ m$, mutually exclusive, only one branch is selected for execution.

6 Compiling Rules in the Programming Model

The table shown in Fig. 4 also shows the translation schemes used by the front-end and back-end compilers. The front-end compiler translates UML diagrams into ADM rules, whereas the back-end compiler translates ADM rules into π-calculus formal specifications. So far, we have implemented a library of Java classes that conforms to a subset of the π-calculus model of concurrency with the purpose of executing the programs specified by the UML diagrams. The ADM programs are then compiled into a semantically equivalent Java program that uses our π-calculus library to produce and executable version of the UML diagrams provided.

7 Conclusions

In this paper, the Moon visual programming environment was presented. The environment facilitates the design, analysis and construction of UML sequence diagrams. The environment uses the ADM rule-based language as a textual representation for the diagrams provided, that are amenable for execution due to the well-defined semantics of the language constructs. Among the contributions of this work, we emphasize the π-calculus foundation of the ADM syntactic constructs which reduces the complexity of rule-based descriptions by introducing structuring notions of object encapsulation, sequential, parallel and conditional composition of simpler rules. We foresee a number of applications ranging from algorithmic visualization, UML-based system design and program development. We also believe that Moon will provide the means for preparing educational material of distributed computing courses centered on standard UML diagrams and rule-based reasoning.

References

1. B.A. Colombo, C. Demetrescu, I. Finocchi, and L. Laura. "A Java-based System for Building Animated Presentations over the Web", Journal paper accepted for publication in Elsevier Science of Computer Programming (SCP), special issue on "Practice and Experience with Java in Education". An extended abstract appears in the Proceedings of the AICCSA'03 Workshop on Practice and Experience with Java Programming in Education, Tunis (July 2003)

2. Baker, R., M. Boilen, M. Goodrich, R. Tamassia, and B. Stibel. "Testers and Visualizers for Teaching Data Structures", In Proc. 1999 ACM SIGCSE Symp., ACM (1999) 261-265
3. Dann, W., Cooper, S., Pausch, R. "Making the connection: programming with animated small world", in Proceedings of the Conference Integrating Technology into Computer Science Education, (2000) 41-44
4. Thomas L. Naps, Guido Rößling. "Evaluating the Educational Impact of Visualization". inroads - Paving the Way Towards Excellence in Computing Education, volume 35, Number 4. pp. 124-136, ACM Press, New York (2003)
5. R. Fleischer and L. Kucera. "Algorithm animation for teaching". In Software Visualization, State-of-the-Art Survey, Stephan Diehl (ed.). Springer LNCS 2269 (2002) 113-128
6. Hansen, S. R., Narayanan N.H. & Hegarty, M. "Designing educationally effective algorithm visualizations". ,lnl. of Visual Languages and Computing (2002) 291-317.
7. G. Booch, J Rumbaugh, I. Jacobson. "The Unified Modeling Language User Guide". Addison Wesley Longman Inc. (2000)
8. C. Lilley, D. Jackso. "Scalable Vector Graphics (SVG) XML Graphics for the Web". http://www.w3.org/Graphics/SVG/, W3C (2004)
9. ArgoUML Quick Guide.http://argouml.tigris.org/documentation/defaulthtml/quick-guide/index.html, Tigris org (2004)
10. L. Quinn. "Extensible Markup Language (XML)". http://www.w3.org/XML/, W3, 2004
11. O. Olmedo, K. Escobar, G. Alor, G. Morales. "ADM: An Active Deductive XML Database System", Springer, LNAI 2972:139-148 (2004)
12. "Rational Rose XDE Developer" http://www-306.ibm.com/ software/awdtools/developer/rosexde/features/ IBM (2004)
13. R. Milner, *Communicating and mobile systems: the π calculus*. Cambridge University Press (1999)
14. G. Andrews, "The distributed programming language SR-mechanisms, design and implementation". *Software-Practice and Experience* 12,8:719-754 (1992)
15. Sinan Si Alhir, *UML in a nutshell. A desktop quick reference*. O'Reilly (1998)
16. A. Begeulin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing'91* (1991) 435-444
17. B. Topol, J. Stasko and V. Sunderam. Integrating visualization support into distributed computing systems. In *Proceedings of the 15-th International Conference on Distributed Computing Systems* (1995) 19-26