A Distributed Reactive Synchronous Programming Language

GIOVANI RUBERT LIBRELOTTO[1]
JONAS BULEGON GASSEN[2]
ROGÉRIO CORRÊA TURCHETTI[1]
SIMÃO SIRINEO TOSCANI[3]

UNIFRA - Centro Universitário Franciscano
Rua dos Andradas, 1614,
Santa Maria, RS, 97010-032, Brasil
[1](giovani,turchetti)@unifra.br
[2]jbgassen@gmail.com [2]sstoscani@gmail.com

Resumo. RS is a reactive synchronous programming language suited to the implementation of the control part of reactive systems. The RS source programs are compiled to finite automata, which are very fast at execution time. This paper describes: (1) the distribution of the RS language, (2) the design of an MDX kernel that implements the communication facilities for the distributed automata, and (3) the implementation of the resulting distributed model using the C language.

Palavras-Chave: Reactive systems, finite automata, reactive synchronous programming language, MDX, distributed automata.

Uma Linguagem de Programação Síncrona Reativa Distribuída

Abstract. RS é uma linguagem de programação síncrona reativa projetada para a implementação da parte de controle de um sistema reativo. Os códigos-fontes RS são compilados para autômatos finitos, os quais possuem um rápido tempo de execução. Este artigo descreve: (1) a distribuição da linguagem RS, (2) o projeto de um núcleo MDX que implementa as facilidades de comunicação entre os autômatos distribuídos e (3) a implementação do modelo distribuído resultante usando a linguagem C.

Keywords: Sistemas reativos, autômatos finitos, linguagem de programação reativa, MDX, autômatos distribuídos.

(Received December 29, 2007 / Accepted March 26, 2008)

1 Introduction

The RS language [11] is intended for the programming of reactive kernels, which are the central and most difficult part of a reactive system [1]. A reactive kernel takes care of all logic of a reactive system, handling input signals, performing reactions and generating output signals [5]. In its original version, the RS language dealt only with centralized kernels, since each program had been compiled to a single automaton [9].

This article describes the changes that allow the RS language to deal with distributed control. The model tries to be general enough, without binding definitions to languages or environments. A communication environment based on the MDX protocol [4] is also described. The environment offers the communication services for the RS distributed automata. The new MDX environment, called MDX_RS, offers a fast communication service to the automata.

The paper is organized as follows: section 2 gives a brief introduction of the RS language; section 3 defines the RS distribution model; section 4 introduces the

MDX system; section 5 describes MDX_RS communication kernel; section 6 describes the implementation of the distributed model, using the C language; section 7 presents the conclusions of the work.

2 The RS Language

The RS language adopts the *synchrony hypothesis*, that is, it assumes that each reaction is performed in zero time. Therefore, the output signals are synchronous with the input ones, and the time only goes by during the external environment activity. This assumption simplifies the language semantics and allows the programs to be compiled to finite automata [10].

The RS compiler translates the source programs to a set of tables that describe a state machine similar to the *Mealy* machine [6]. As the object code is not an executable file, the system needs an interpreter for the automata execution. In addition to the control kernel, a reactive application requires the implementation of an I/O interface, to receive the input signals and to conduct the output signals, and a set of procedures, to handle the application data.

As it occurs with Esterel [3, 2], Lustre [5], and other synchronous languages, RS is not a self-sufficient language; the I/O interface and the data handling components should be provided by a host language or by the execution environment. An RS source program is composed by a set of *modules*, each module is composed by a set of *rule boxes* and each rule box is composed by a set of *reaction rules*. Despite having these 3 levels, every source program can be seen as composed by only a single set of *reaction rules*. Each rule has the form $C \rightarrow action$, where C is a set of signals, called the *firing condition*, and *action* is a sequence of statements. A firing condition C is true when all of its signals are on.

The execution of an RS program is accomplished in a sequence of steps, where each step consists of the *parallel execution* of all rules with firing condition *true* [8]. The first action of a step is an implicit action that turns *off* all the signals contained in the true conditions. As the execution of a rule can turn *on* signals, this originates a sequence of steps that only finishes when the set of *on* signals is not enough to fire any reaction rule. In this situation, the program waits until some external signal arrives to start a new reaction.

3 The distribution of the RS language

The distribution of control is currently used in many environments, such as plant floors, domestic automation, and robotics. In these environments, a component reaction can depend on another component behavior. This section defines a model that allows the translation of an RS program into a set of automata and, afterwards, the execution of these automata on different machines.

3.1 Syntactic modifications

The RS distribution requires new declarations to allow: (a) to compile a program for a set of automata; (b) to place these automata in different processors. The adopted approach allows to specify, in the same source code, the processes to be distributed as well the machines in which they will run.

3.1.1 The machine declaration

The *machine* declaration introduces a new level in the syntactic hierarchy of the language. Beyond the three levels of the original structure, i.e., *modules*, *rule boxes*, and *reaction rules*, the machine declaration adds a new level. By the way, this level is the same as that of a program, i.e., the machine declaration may contain all the old structuring levels.

A distributed program begins with the declaration *rsd_prog* and is composed by several machine declarations. Another small syntactic change was the inclusion of the external interface definition, after the header *rsd_prog*. The *external interface* declaration allows the identification of the *input* and *output* external signals. This declaration is intended only to make the program more clear, because the information it contains is redundant and could have been gotten directly by the compiler. Now, each *machine* declaration originates a single independent automaton for a specific machine. This new syntactic hierarchy of the RS language can be seen in next subsection.

3.1.2 An example of a distributed program

To illustrate the new syntax, a distributed RS program is presented in figure 1. Although there is no interest at the moment, neither the program has any practical importance, what the program verifies if a mouse button was pressed with a *double* or a *single* click. The program has two *input* signals: *tick*, that means a clock impulse, and *click*, that means the pressing of the mouse button, and two *output* signals: *single* and *double*. The program is executed in two machines, called *sinope* and *pan*.

```
rsd prog mouseD:
  external interface:
    [ input : [click, tick], output : [single, double] ].
  machine sinope :
    [ input : [click, tick], output : [start, relax]
      module timer:
      [ input : [click, tick], output : [start, relax], p signal : [a,b],
        var : [delta], initially : [up(a)], tick#[a] ===> [up(a)],
        click#[a] ===> [delta:=3, emit(start), up(b)],
        tick#[b] ===> case [delta>0 > [delta:=delta-1, up(b)],
                      else > [emit(relax), up(a)] ] ]
  ].
  machine pan:
    [ input : [click, start, relax], output : [single, double],
      module emitter:
      [ input  : [click, start, relax], output : [single, double], var : [count],
        start ===> [count:=0], click ===> [count:=count+1],
        relax ===> case [count=0 > [emit(single)],
                   else > [emit(double)] ]
  ]
].
```

Figura 1: A distributed RS program for a mouse button

3.2 Distributed automata generation

In the following, a *machine* [1] declaration is called an UD (Unit of Distribution). The steps to compile each UD and later to transfer the generated automaton to the corresponding machine is now summarized:

1. Put each UD in a proper file, keeping the identification of the machine and substituting "*rs_prog name#n*"for "*machine machineID*", where *name* is the name of the distributed program and *#n* is a counter value (the initial value of the counter is zero and it is increased by 1 for each occurrence of *machine*).

2. Compile each UD (using the original RS compiler) and send the automaton to the corresponding machine.

In the previous example, the name of the distributed program is *mouseD* and it has two UDs. The corresponding automata to these UDs are called *mouseD1* and *mouseD2*. The generated code for each UD is placed in two files, one with extension *.aut* and another with extension *.rul*. The first one describes the automaton and the second one contains the reaction rules for this automaton (from which the actions are obtained at execution time). For the distributed mouse control program, the files in figure 2 will be generated.

The compiler also generates another file, with extension *.iod*, called Archive of Distributed Information (AID), which specifies in which machine each automaton will run, which are the signals coming from the external environment, and which are the signals that must be sent to other automata. Coming back to the distributed mouse example, the AID file is:

```
IOFILE mouseD
  External input:  [click, tick]
  External output: [single, double]
AUTOMATON mouseD1 (machine sinope)
  External input : [click, tick]
  External output: [start, relax]
  Input from env.: [click, tick]
AUTOMATON mouseD2 (machine pan)
  External input : [click, start, relax]
  External output: [single, double]
  Input from env.: [click]
```

3.3 Execution environment for the distributed automata

The system has a master–slave organization, where the master is the RS_Main process and the slaves are the automata. The master communicates with the user and sends signals and commands to the slaves. The slaves carry out the reactions and send the results to the master, which shows them on the screen. To free the master from the blocking data entry activity, a special process named RS_IO is used. This process reads the input data from the keyboard. The current prototype is useful during the debugging phase of a distributed system. In the final version of this system, the RS_IO will be replaced by the external I/O interface, which will implement the real communication with the external environment.

The processes that compose the RS distributed environment are:

- *RS_IO*: receives input signals and commands entered by the user and sends them to the RS_Main process.

[1] Each machine declaration specifies a complete original RS program.

mouseD1.aut	**mouseD1.rul**
$AUTOMATON\ \ mouseD1:$	Rules for mouseD1 :
$init\ \ -\ \ [1,*,go\_to(1)]$	Module timer:
$1\ \ click\ \ [2,*,go\_to(2)]$	1. [ ] ===> [ ]
$1\ \ tick\ \ [1,*,go\_to(1)]$	2. [ ] ===> [delta:=3, emit(start)]
$2\ \ tick\ \ [[3-1,*,go\_to(2)],$	3. Case:
$[3-2,*,go\_to(1)]]$	3-1. [ ] delta>0—>[delta:=delta-1]
	3-2. [ ] else —> [emit(relax)]
mouseD2.aut	**mouseD2.rul**
$AUTOMATON\ \ mouseD2:$	Rules for mouseD2 :
$init\ \ -\ \ [1,*,go\_to(1)]$	Module emmiter:
$1\ \ start\ \ [2,*,go\_to(1)]$	1. [ ] ===> [ ]
$1\ \ click\ \ [3,*,go\_to(1)]$	2. [ ] ===> [count:=0]
$1\ \ relax\ \ [[4-1,*,go\_to(1)],$	3. [ ] ===> [count:=count+1]
$[4-2,*,go\_to(1)]]$	4. Case:
	4-1. [ ] count = 0 —> [emit(single]
	4-2. [ ] else —> [emit(double)]

Figura 2: Generated automata for distributed mouse

- *RS_Main*: initialize and finish the automata, transfer each signal received from RS_IO to the ARSD that treat this signal, and shows results and error messages in the screen.

- *ARSDs*: the distributed RS automata. They carry out the commands received from the RS_Main process, and react to the received signals.

4 The MDX parallel programming environment

The MDX environment[2] [4] executes over an heterogeneous workstation network, and implements a parallel virtual machine with shared memory. The idea is to allow the programmer to write multithreaded programs where the threads are dynamically created in the network, and in which sharing of data is done through the distributed virtual memory. Each network node is a computer with one or more processing units and a local memory, interconnected by a physical network, under some operating system such as Windows, OS/2, Linux, and Solaris, among others.

In an application, the same program is replicated in all the nodes of the network, however in each node only a subset of threads is executed. Thread synchronization is carried through semaphores and barriers. The MDX system is based on the client/server model, and it uses RPC as the communication mechanism. The system is composed by a name server, a distributed virtual memory manager, a compiler manager, an execution manager, and a synchronization manager. These components use the services of a communication kernel. The basic architecture of the MDX system over a network of workstations is shown in figure 3.

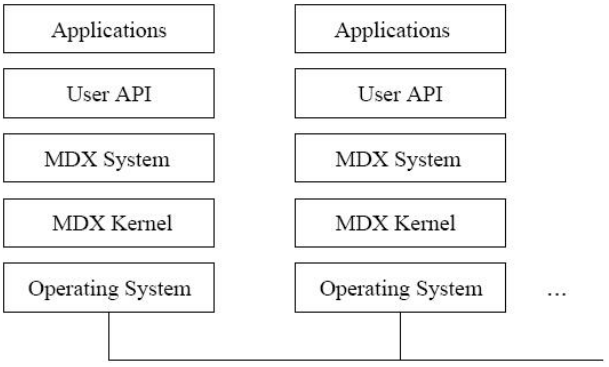

Figura 3: The basic architecture of the MDX system over a network

The communication kernel is intended to allow the communication between the client and server processes, regardless of their location, in a fast, trustworthy, and transparent way [4]. To accomplish this duty the kernel executes the following three tasks: (a) message examination to identify if it is a request or a reply and to whom it must be delivered; (b) location of the involved client or server process; (c) sending of the messages to the right place.

[2]The MDX system is being investigated as part of a project involving the following Brazilian universities: UNICRUZ, PUCRS, and UFRGS.

5 MDX_RS — A communication kernel for RS automata

Communication requirements of the RS environment are small and may be provided by a simple kernel, which will be called MDX_RS. This kernel is presented in the coming sections.

5.1 Functions required by the RS environment

There is no need of shared memory, for the RS automata; instead, some message interchanging mechanism. In the reaction rules source code (file *.rul*), the occurrence of the statement *emit(s)* means that the automaton will emit the signal s to either another automaton or to the external environment. The destination is defined in the file AID (extension *.iod*). For example, in the *mouseD* distributed program, the statement *emit(start)* sends the *start* signal from the machine *sinope* to the machine *pan*, because in the AID file it is indicated that *start* is an output signal of*mouseD1* (machine *sinope*) and is an input signal for *mouseD2* (machine *pan*).

In the translation of an ARSD to a C program, the *emit* command is substituted by the communication primitive *MDX_send( )*, which sends a given signal to a specific automaton. For this to happen, a previous connection between the two automata must have already been established. The establishment of the connection creates one socket and, through this *socket*, the two automata will communicate until the end of the distributed program execution. The *MDX_send( )* command for the above described example would be the following: *MDX_send (mouseD2, "start");* where *mouseD2* is the automaton that will receive the message and start is the signal to be sent.

For signal reception, every automata has a thread that uses the *MDX_recv( )* command, which has the following format: *MDX_recv (int Socket, char *message);* where the parameters specify the connection *socket* and the *message*, which will be later treated according to the RS protocol. It must be remarked that an ARSD is very different from an MDX server. When an automaton receives a signal, it can perform only a local computation, or even it can execute nothing, if the signal is not expected in the current automaton state.

5.2 Structure of the MDX_RS communication kernel

The original MDX system is based on the client/server model, where several clients ask for services to one or more servers. As an automaton is neither a client nor a server, it was necessary to define a new structure for the communication kernel.

The MDX_RS kernel uses a NLT table (Name Local Table) that registers the network nodes that are used by the distributed system.

5.2.1 The establishment of connections

The procedure that establishes the ARSD's connections has a parameter that receives a communication port identification, which will be used to wait for connections with other automata. After receiving this port specification, the automaton waits, in an *accept* command, until another automaton establishes a connection with it. When this happens, the information about the connection is added to the NLT, and a *thread* for the intercommunication of these two automata is created.

This *thread* repeats a *loop* until the automaton receives a message asking for its termination. Inside the loop, the automaton waits for a message using the *socket* created in the previous procedure. When a message is received, the message is treated in according to the RS protocol. If, after the handling of the message, it is necessary to send a signal to an automaton A, then a seek operation is made in the NLT to find the socket belonging to A and the message is sent to this *socket*, through an *MDX_send( )* operation.

5.2.2 Structure of the new kernel

In the RSD (distributed RS) system, an *Init( )* process that waits for new connections between ARSDs will always be active. From this *Init( )*, for each established connection, a thread is created (procedure *Recv( )*) which makes the communication between the local ARSD and the remote one. This structure is shown in figure 4.

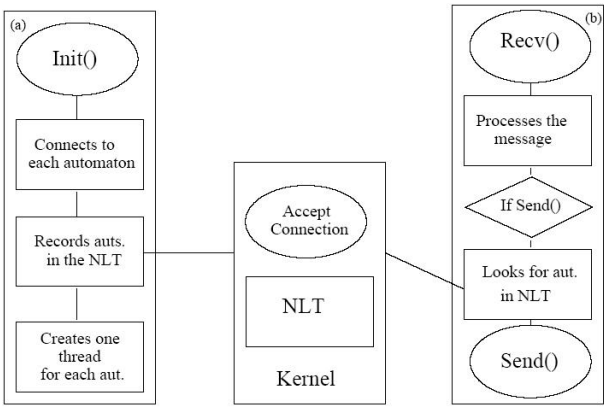

Figura 4: Structure of the MDX_RS communication kernel

5.3 Structure of the MDX_RS system

The structure of the MDX_RS system has just three layers:

- *Automaton*: It is a C program that implements an ARSD and uses the MDX_RS communication and synchronization primitives.
- *MDX_RS Kernel*: It is the main part of the system. It supplies the communication support to the ARSD and manages the NLT.
- *Operating System*: Until now it has only been tested on Linux.

As it can be seen, the user API and the specialized servers have been eliminated. The API could be removed because the kernel, together with the RSD model functionality, already guarantee the required transparency, giving the illusion that the automata executed in a single machine.

As to the specialized servers, it can be stated that the execution server is not necessary because the actual prototype requires the manual loading and execution of each automaton (later the system will be improved in this aspect). The shared memory server plays no role, because the RS automata do not share memory. The synchronization server is not needed because the synchronization is made by the RSD system itself. Similarly, the compilation is done directly by the user. Finally, the name server is not necessary because each automaton has a copy of the NLT in its local kernel.

6 CRSD — The RSD compiler

The objective of the CRSD is not to substitute the original RS compiler, but to complement it. The idea is to use the original compiler generated code (an automaton) to generate the C code that implements the automaton. The CRSD uses only the information of the files that describe the automata.

`Code generation.` The code generation is accomplished in two phases: *action generation* and *automaton generation*. In the first phase, the program variables are identified and the RSD commands and attributions are translated. In the second phase, the action execution order is defined. From the RS automaton definition, the *automaton* routine is constructed; basically, this routine contains calls to the previously generated procedures and functions.

`Handling of parallel actions.` In the representation of an ARSD, the asterisks are used to separate actions that can be executed in parallel. During the execution of these actions, the signal values must remain frozen (that is demanded by the semantics of RS language). That is, the new value of a signal can be attributed only after the execution of all the parallel actions. To solve this problem the same solution of the old RS system is used: every signal possesses an original value and a copy, the value is always read from the original signal and written in the copy. When the parallel action finishes, the copy value is used to bring up to date the original signal.

`Internal exception handling.` An internal exception means a communication error or even an execution error. In such situation the ARSD emits a exception message to the RS_Main process and waits for a reply, which can be either an order to finish or an order to continue the processing. The waiting for reply is synchronous; in such a way the ARSD does not execute any command until the reply arrives.

`RS protocol.` The RS protocol [7] standardizes the information exchanged between the following pair of processes: RS_Main-ARSD, ARSD-ARSD and RS_IO-RS_Main. For the total ordering of messages the system uses *logical clocks* and *time stamps*, according to Lamport's algorithm [9]. The messages for which the ordering is not important do not receive time-stamps. The main reason for the option of centralized control was maintenance easiness.

7 Conclusions

In the RSD language, all the components to be distributed are specified in a single source program, which improves the programmer view of the structure of the system as a whole. Each component is compiled for an independent automaton and, for each component, the programmer specifies the machine where it will run.

The distribution of language RS extends the scope of its applications, since the language starts to contemplate distributed controls. One of the advantages of organizing a system as a set of communicating automatons, instead of a single automaton, is the reduction of size (number of states) of the system. The great disadvantage is the overhead introduced by the run-time communication.

Despite the overhead corresponding to the communication between automata running on different machines, it was possible to observe a sensible increase in the execution speed of the distributed programs when compared to the centralized program. This is justified by the fact that the code generated by the CRSD compiler is faster than the interpreted code of the original RS language. In fact, the distributed mouse example showed that the RSD version, executed on the MDX_RS communication kernel, has better performance than the cen-

tralized RS program (this can be detected visually). But this apparent better performance can be due to the low complexity of the example and needs to be better studied. However, as a big distributed system will be composed by several automata, having several operations occurring at the same time in separate machines, hopefully the distributed performance will be higher than the centralized one.

As RSD is still a prototype in a test phase, it is natural that it has improvements to be introduced. With respect to security, for example, the monitoring of the master process must be implemented, therefore. In relation to the MDX_RS kernel, an execution manager must be created to make possible the execution of the kernel and the automata in remote machines, with only one command. The experimentation of the RS system in real distributed applications will probably indicate other improvements to be introduced in both the model and in its implementation. These improvements will be introduced in future versions of the system.

Referências

[1] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and de Simone, R. "The Synchronous Languages 12 Years Later". *Proceedings of the IEEE*, 91(1):223–252, January 2003.

[2] Berry, G. "The foundations of Esterel". Proof Language and Interaction: Essays in Honor of Robin Milner, 2000. Cambridge, MA: MIT Press.

[3] Berry, G. and Gonthier, G. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] Costa, C., Dotti, F., Copetti, A., and Preuss, E. MDX-A Parallel Programming Environment Supporting Distributed Shared Memory an message Passing, 2000. Simposio Argentino de Tecnología, Tandil, Argentina.

[5] Halbwachs, N. Synchronous Programming of Reactive Systems, 1993. Dordrecht: Klumer Academic Publisher.

[6] Hopcroft, J. and Ullman, J. Introduction to Automata Theory, Languages and Computation, 1979. Reading, Massachussetts: Addison-Wesley.

[7] Librelotto, G., Toscani, S., and Monteiro, L. Distribution of the RS language in the MDX environment, 2000. SBLP 2000, Recife - PE. p. 120-133.

[8] Librelotto, G. R., Cassal, M. L., Turchetti, R., Dhein, G., and Toscani, S. S. "The RS Language for Distributed Automata". In *Proceedings of Conferencia Latinoamericana en Informática*. Santiago, Chile, 2006.

[9] Peterson, J. and Silberschartz, A. Operating Systems Concepts, 1985. New York, Addison-Wesley Publishing Company.

[10] Raymond, P., Jahier, E., and Roux, Y. Describing and Executing Random Reactive Systems. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 216–225, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Toscani, S. RS: Uma Linguagem para Programação de Núcleos Reactivos, 1993. Depto de Informática, UNL, Lisboa, Portugal.