Specification and Modelling of Distributed Systems

R. Maye, A. S. Bavan and G. Abeysinghe
School of Computing Science
Hendon Campus
Middlesex University
The Burroughs
London NW4 4BT
United Kingdom
{r.maye, s.bavan, g.abeysinghe}@mdx.ac.uk

Abstract

The paper describes a new approach to specify distributed programs using a diagrammatic notation (NMDS) and implementing them using an asynchronous message passing language (LIPS). The approach adopted here is to define a network of dedicated communication channels for data exchange in LIPS-compliant programs; thus promoting portability between different computing platforms. The NMDS expresses distributed specifications without ambiguity in a clear and concise manner. Apart from providing a clear picture of the intended system it also fits in with the syntax and semantics of the LIPS language thus making it easier to translate the specification into LIPS programs. One of the major advantages of using NMDS as a specification tool with LIPS is that reverse engineering of LIPS programs can be achieved without significant overheads.

Keywords:
Distributed Systems, Guarded Processes, Role Activity Diagrams, Process Modeling

I. Introduction

Successful implementation of any software demands that the design and implementation satisfy the user requirements reliably. This means that the choice of design tools and the implementation language must be carefully chosen in such a way that the design tool is able to clearly express the system and the language is able to translate the design unambiguously. Ideally, there should be a clear correspondence between the design and the implementation so that there is no compromise when translating the design into code. This will not only guarantee a correct system but also ease the burden on the maintenance process that follows it. A reliable design technique is the one that unambiguously and clearly represents the intended system in such a way that it lends to verification and modifications. In addition one would expect this representation to promote elicitation so that one can learn more about the system with the view to refine it to produce a more efficient representation as well as ease the task of maintenance. On the other hand, if the chosen implementation language is compatible in style, structural characteristics, and functional characteristics, the implementation task will not only be easier but also more likely to be efficient and less prone to errors. In the case of distributed systems, even though there are a number of good implementation languages [1, 2, 3, 4] there is a severe shortage of design techniques outside the mathematically based formal techniques such as CSP [5, 6] and CCS [7] and other similar systems. Since these formal techniques require a good understanding of mathematics most of the designers are discouraged by them. This paper presents possible solutions to both of these problems by presenting a diagrammatic notation for specifying distributed processes, NMDS, and a language for implementing parallel and distributed systems (LIPS) which easily translates the diagrammatic notation into implementation code.

Amongst the available design tools based on graphical notations, Petri nets [8, 9, 10] and RADs [11, 12] can be considered to be superior in their ability when it comes to modelling / expressing distributed systems. These two methods are very efficient at process modelling, but they are not good at expressing distributed computations, where the system consists of multiple communicating processes. This has motivated us to develop a graphical notation based on Role Activity Diagrams that has the required features for expressing distributed processing in a manner that is not only unambiguous and clear but can also be translated into formal specifications and LIPS programs, thus lending itself to verification.

In section 2 we present an overview of the LIPS language which inspired the development of NMDS. In section 3, we describe the NMDS notation and illustrate its use with examples. Finally, we present our conclusion in section 4.

II. Overview of LIPS (Language for Implementing Parallel and distributed Systems)

A LIPS program consists of interconnected nodes, where a node refers to an instantiation of a node definition. A program in LIPS is specified in two parts:

- a network definition part that describes the topology of the network of nodes
- a node definition part that defines the computational steps in a node

A. Network Definition

The network definition describes the topology of the program by naming each node in the program and its relationships (in terms of input and output data) to other nodes in the system. This is achieved through the use of *link* statements. There are two types of link statements, namely connect and rep; which are available for specifying the connectivity of a node in a program. The rep statement is purely a short hand way of describing similar link statements and is therefore omitted from this paper for brevity. The syntax of the connect statement is as follows:

```
[node_label]:connect node_name (input
list) -> (output list)
```

An example of a connect statement that links two input channels, a and b to a node that executes process P with a node label 7 and produces one output on channel c is as follows (see figure 2):

```
[7]:connect P ([a, b]) -> ([c])
```

The square brackets are used to group data belonging to a particular data type or category.

Using connect statements, one can build networks of any complexity. The connect statements allow a dataflow graph to be built for a problem without having to worry about the specification of computation at network level. This promotes good design practice by motivating the programmer to produce the target solution model at a higher level of abstraction using communication as the framework; thus divorcing computation from it.

B. Node Definition

In giving a definition of a node we appeal to the notion of a guarded process which we shall define shortly. A node, in the LIPS language, consists of one or more guarded processes and is specified as a function with the following syntax.

```
node node_name (inputs) -> (outputs)
{
  declarations;
    guarded_process_1
        :
    guarded_process_n
}
```

The elements of the input/output list represent a set of virtual input/output channels through which the messages are sent and received by the nodes. declaration used within a given function is local to function.

A guarded process consists of a guard and associated body. The guard is a list of channels zero or more status flags. The body uses data defined in the guards and processes them according to given specification. There are 2 other special types guards, an initialising guard which only gives values to status flags and variables, and a start guard which marks the entry point of execution for system. In the case of two or more guarded processes being eligible for execution, only one is selected randomly. The input channels in different guards mutually exclusive. This allows the avoidance ambiguity in selecting a process for execution. typical guarded process has the following syntax.

```
(input_condition) => (statements)
```

For example, the following guarded process evaluates y only when all the input channels, x1, x2, and contain new and unused data.

```
[x1, x2, x3] => (y = 2*x1 + 3*x2 - x3;)
```

Complex input conditions can be constructed logical and special operators. A process is defined by using C language statements. Transfer of data via channels is achieved by simple assignment of values to variables that represent virtual channels. divorces the communication from computation takes away the burden of dealing with deadlock the programmer.

III. Notation for Modelling Distributed Systems (NMDS)

Traditionally, diagrammatic design tools are preferred instead of formal specification techniques such as CSP [5, 6] and CCS [7] for modelling purposes. This mainly because of the fact that the latter requires good understanding of mathematics and process algebra. This naturally lead to the development number of diagrammatic techniques for modelling of data, flow of control and various relationships between objects. Most software engineers find it easier to construct their design and able to convey information about structure and logic of the system using these diagrammatic approaches. On the hand, when modelling distributed systems and algorithms, most diagrams lack the ability to succinctly reveal the concurrency that is inherent in these systems and show the communications between processes. Exceptions to this belief are two candidates, Role Activity Diagrams (RADs) [11, 12] and Petri nets [8, 9] which provide a clear way of modelling processes and interactions between processes.

Role Activity Diagrams have a number of characteristics that makes it particularly suited for modelling distributed processes compared to other existing diagrammatic representations we have studied. These include:

- Relatively few symbols and relaxed rules in RADs enable us to construct accurate diagrams with relative ease.
- They are state based diagrams and so have a formal background. Mapping RADs to CSP has already been demonstrated [13].
- They have the ability to express the concept of interactions between processes and thus allow the designer to specify the communications between processes clearly.
- Concurrent processes can be modelled using the dangling and simultaneously executing threads.

Despite their benefits, Role Activity Diagrams have a number of weaknesses when used for modelling distributed processes based on lips. These include:

- All interactions are synchronous (blocking send).
- Have no facilities for specifying detailed message passing as the interaction is specified at abstract level. This can hide important details of the communications between processes.
- There is no concept of a guard as represented in LIPS, and modelling mutually exclusive threads can make the model complex.

Because of the aforementioned weaknesses, we initially incorporated extensions to tighten the rules for using interactions. This exercise not only produced a set of confusing notations but also removed the intuitive trait of RADs. This lead to the development of our own notation, NMDS, using the principles and symbols of RADs as the basis. NMDS was specifically developed to model the main concepts of distributed, guarded processes, with much stricter rules.

A node is represented by a rounded box, as are roles in RADs. Guards within a node are represented by a rectangle containing all the prerequisites for that guard to become active as shown in Figure 1, with the adjoining process leading from it. The symbol (□ represents a buffered receiver, and Ⓢ represents a status flag.

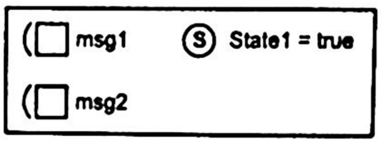


Figure 1: A guard

This can be perceived as being similar to the mathematical representation of a guard using set theory, in which the same guard given in Figure 1 may be given as the unordered set: {msg1, msg2, state1=true}

The start and initialising guards are represented by labelled boxes as shown in Figure 2.

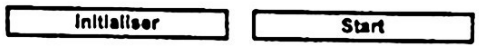
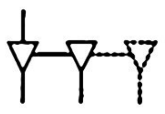


Figure 2: Initialiser and start guard notation

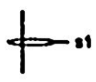
A process associated with a guard may consist of actions, sending of messages and the changing of status flag values and can be specified using the following symbols:

■ Perform some action	Performs actions described by the accompanying label
▨ Send a message	Sends a message. This is accompanied by a label explaining what or why.
(□	A buffered receiver
Ⓢ State1 = false	A status flag and a label with its value.

To specify the control flow, we use the following constructs:



Choice operator. Identical to the RAD notation. There must be a minimum of two possibilities.



State Label. Identical to the RAD notation. State labels are used to label any state along the process flow.

Although it is only necessary to limit the scope of a named state to the corresponding guarded process, it has been found that it is less confusing to the user if the scope spans the entire node, thus each state label should be unique to the corresponding node.

○ s1

Jump. The flow of execution is redirected to the state given. It allows for the construction of iterations.

⊗

Terminate. This indicates the completion of the entire system. Not all systems do terminate, and so this symbol is not mandatory.

◆

End of a guarded process. The execution of the current guarded process ends, and thus the next eligible guard is chosen.

To demonstrate the notation, we present an example that is designed using the RAD notation and the NMDS notation to show the improvements that can be achieved by using our notation. In our example, we use the Simpson's rule to calculate the area under the curve: (4 / (1+x2))(π) within an interval of [0-1]. The network consists of three types of nodes, "*Host*" which issues the work to the distributed nodes and collects the result; "*Area*" which calculates the area of a slice using its width (*ww*) and its height –(calculated using the segment number *k[x]*) and "*Sum*" which adds all the areas of the segments to produce the total area. The topology of the system is as shown in Figure 3.

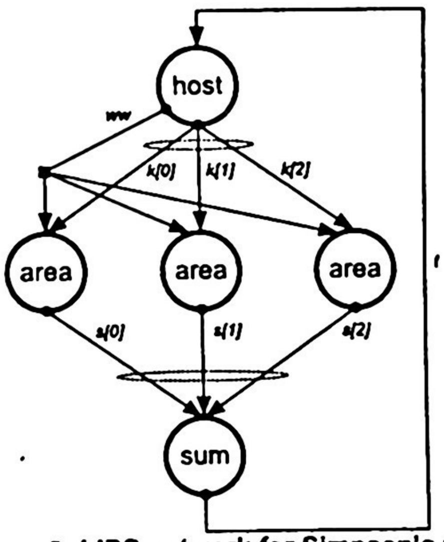


Figure 3: LIPS network for Simpson's rule

The RAD representation given in
Figure 4 uses two small additional symbols that are not normally found on Role Activity Diagrams, but have been included to clarify the situation:

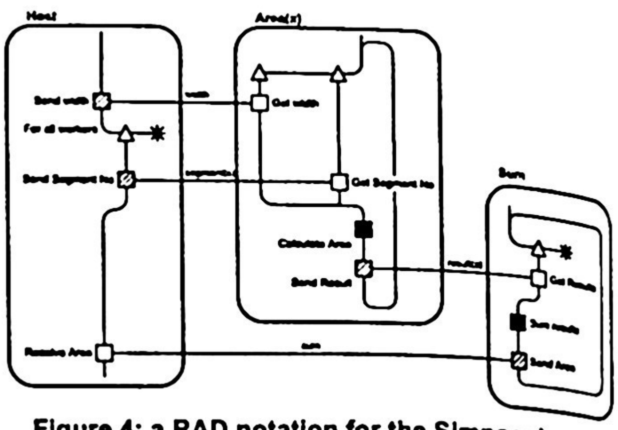


Figure 4: a RAD notation for the Simpson's rule

- The role "Area" has been parameterised, it is named "Area(x)". In each real instance of the Area(x), x is replaced with a unique value. So, example, if there were three instances of Area, then they could be named Area(1), Area(2) Area(3) respectively.
- Interactions may be parameterised if a number unique messages are being sent or received. example, segment(x) indicates that unique segment value is sent to each instance of node area, and result(x) indicates that the calculated by each instance of node area received by node sum.

Even without these nuances, Role Activity Diagrams would seem perfectly capable of modelling distributed processes. Guarded processes, however, are more than just distributed processes.

Figure 5 shows the NMDS notation of the Simpson's rule. Although similar to
Figure 4, we believe the new notation provides a much less ambiguous representation of the system. It makes a very clear distinction between the guards' prerequisites and the guarded processes themselves.

Note also the use of the dot notation: the *fan-out* ▨— and the *fan-in* receive —<□. Fanning-out fanning-in is the process of sending or receiving number of unique messages, typically achieved through iteration, as shown by the for-each construct is in the RAD example given in
Figure 4. This is different to the normal send, such the sending of the width message, as although

message may have multiple recipients, only 1 message is broadcast.

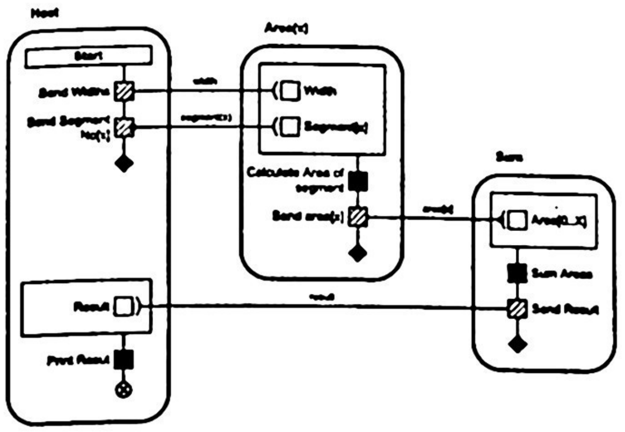


Figure 5: NMDS notation for Simpson's rule.

IV. The need for NMDS

NMDS was developed after unsuccessful attempts to model guarded distributed systems using Role Activity Diagrams (RADs). RADs is not designed to express mutually exclusive processes and in order to model such a system requires the inclusion of a framework to ensure the mutual exclusiveness. Essentially, this implies that the modeller has to concem themselves with the runtime system rather than being able to abstract purely to the logic of the program itself.

One of the major difficulties was in the representation of the guard. It is important to note that a guard is a set of preconditions such as the precondition concept in Z notation [14] and does not form part of the process logic. RADs offer no way to divorce the two concepts, which makes models hard to understand.

Although RADs provide ways to represent interactions involving multi-processes it is implicit that these interactions are synchronous. This counteracts to the concurrent behaviour we aim to achieve in distributed systems creating bottlenecks. In guarded distributed systems, we send and receive asynchronously. For example, a sender of a message should not be bottle necked by the behaviour of the receiver. Such behaviour can be modelled with RADs by introducing an unbounded buffer at each receiver, but this makes diagrams cumbersome.

Our first attempt to rectify the problems was to extend the RAD notation, by introducing a way to model the guarded process as dangling threads and impose stricter rules on the use of the send and receive symbols. We also created new symbols to represent buffered receivers, but the results were ambiguous and complicated.

NMDS was created as a new notation that would be based upon the notation of RADs, but be significantly different in order to specifically model distributed guarded processes.

V. Conclusion

This paper has attempted to describe a diagrammatical design notation called NMDS and the parallel and distributed language LIPS, on which the NMDS is based. NMDS is also based on the principles of RADs and was specifically designed to design parallel and distributed systems. Using an example in section 3, we have demonstrated that the NMDS approach is better at expressing concurrency and modelling guarded processes than RADs. The main motivation behind the development of NMDS is to produce a design tool that will not only express concurrency unambiguously, but also downward compatible with the LIPS language so that implementation of the design can be achieved easily without losing any information in the design. As seen in our example, NMDS also managed to capture the structural and logical characteristics that are built in to the LIPS language.

It is intended to extend our work in this area by producing an animated version of NMDS for verification purposes prior to implementing in LIPS code. The work is also underway to produce software that would translate NMDS based design into CCS and CSP to aid verification.

References

[1] G. R. Andrews and R. A. Olssen, "The SR Programming Language: Concurrency in Practice", Benjamin-Cummings. 1993.
[2] H. E. Bal and M. F. Kaashoek, "ORCA: A Language for Parallel Programming of Distributed Systems", *IEEE Transaction on Software Engineering*, 18(3), pp190-205.1992.
[3] J. Kerridge, "OCCAM programming: A Practical Apprcach", Blackwell Scientific Publication.1987.
[4] W. Gropp, E. Lusk and A. Skjellum, "USING MPI Portable Parallel Programming with Message Passing Interface", Pub MIT Press, 1994.
[5] C. A. R. Hoare, "Communicating Sequential Processes", Prentice-Hall. 1985.
[6] A. W. Roscoe, "Theory and Practice of Concurrency", Prentice Hall. 1998.
[7] R. Milner, "Calculus of Communicating Systems", *Proceedings of the LNCS*, Springer Verlag. 1992.

[8] J. L. Peterson, "Petri Net Theory and the Modelling of Systems". Englewood Cliffs: Prentice-Hall. 1981.
[9] W. Reisig, "Petri Nets: An Introduction", *Volume 4 of EATCS Monographs in Theoretical Computer Science*. Berlin, Springer-Verlag, 1985.
[10] W. Reisig and G. Rozenberg, "Editors Lectures on Petri Nets II: Applications", *Volume 1492 of Lecture Notes in Computer Science*. Berlin: Springer-Verlag, 1998.
[11] M. A. Ould, "Business Processes, Modelling and Analysis for Re-engineering and Improvement", John Wiley & Sons, 1995.
[12] M. A. Ould and C. Roberts, "Modelling Iteration in the Software Process", *Procs. Third International Software Process Workshop*, Breckenridge, Colorado, USA. IEEE Computer Society Press. 1986.
[13] G. K. Abeysinghe and K. T. Phalp, "Combining Process Modelling Methods, Information and Software Technology". Elsevier Science. 1997.
[14] A. Z. Diller, "Z An introduction to formal methods", John Wiley & Sons, 1990