MODELLING AND DISCRETE-EVENT SIMULATION OF COMPLEX SYSTEMS USING RAINBOW

Angelo Furfaro, Libero Nigro, Francesco Pupo

Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, I-87036 Rende (CS) - Italy
Email: a.furfaro@deis.unical.it {l.nigro,f.pupo}@unical.it

Abstract: This paper describes a modelling language –Rainbow- based on Coloured Petri Nets, which was designed for modelling and simulation of complex systems. The formalism uses Java as the net annotating language. The timing model permits different policies to be associated with places which affect the token binding process. A graphical tool was achieved in Java which supports editing, debugging and simulation of CPN models. Large models can be simulated on top of a Time Warp based distributed executor. The practical use of Rainbow is demonstrated through a scalable simulation model.

Keywords: Modelling, simulation, complex systems, coloured Petri nets, Java

1. INTRODUCTION

Coloured Petri (CP) nets (Jensen 1992-98) are a well-known class of high-level nets that extend ordinary Petri nets (Murata, 1988) by allowing tokens to carry arbitrarily complex data, and arcs to be annotated with input predicates (influencing the enabling of a transition) or output functions (stating the production rule of tokens when a transition fires). Declarations and net inscriptions can be expressed by means of mathematical notations or by using an ordinary high-level programming language.

The work described in this paper focuses on the development of a CP-net dialect -Rainbow– which was especially designed for supporting modelling and simulation of large systems, in a centralized or distributed setting. Key features of Rainbow are:

- the use of Java as the net annotating language. Colour sets of places, arc inscriptions and guards (of arcs and transitions) can directly be programmed in Java
- a timing model which accommodates both unordered and ordered places. Unordered places support classical non deterministic token selection. Ordered places can work with different token selection policies, e.g., FIFO-strict and FIFO-random, which restrict the choice of tokens during the binding process, on the basis of colours and time.

A totally portable Java-based graphical tool was achieved which enables editing, debugging and simulation of Rainbow models on a single workstation. A distributed executor based on a Time Warp mechanism (Beraldi and Nigro, 2001)(Beraldi *et al*. 2002) was implemented which supports distributed simulation over a networked system. Details of the distributed executor are described in a recent paper (Furfaro *et al*., 2002b).

This paper summarises the Rainbow modelling language and associated general timing model. The implementation status of the project is then clarified. After that, a scalable simulation model, together with some experimental results, are presented to demonstrate the practical use of Rainbow. Finally, directions of on-going work are outlined in the conclusions.

2. THE RAINBOW MODELLING LANGUAGE

The following provides a brief and informal description of Rainbow. The formalism relies on Java as the net programming language. With respect to similar modelling languages and tools (e.g. Renew (Kummer *et al*., 2002)), types (classes), functions ad so forth are expressed in Java and not using a syntax which requires mapping and translation in Java. Rainbow hosts only basic net constructs and focuses on time management.

2.1 Places

To each place is associated a class (*colour set*) whose instances are the admitted tokens (or colours). Place classes are extensions of the ColourSet base class. A few colour set classes, corresponding to primitive data types, are predefined so as for them to be immediately reused: ColourInt, ColourFloat, etc. Tokens in a place form a *multi-set*. A parameterless *initialization function* can be assigned to a place to provide its initial marking.

2.2 Arcs

Can be input or output. Input arcs connect places to transitions. The input places of a transition constitute the transition *preset*. Output arcs connect transitions to output places (transition *postset*). Both input and output arcs can be annotated by *arc inscriptions*, e.g. a *variable* or a *function*. More in particular, input arcs are normally decorated by a variable, which will be bound to a colour from the emanating place. In alternative, a function can be attached to an input arc, checking for the existence of suitable tokens in the relevant place of the preset. Input arc functions can be replaced by arc *guards*. A guard is a function which returns true if the token bound to the arc variable satisfies a certain selection criterion. By default, guards evaluate to true if missing. Output arc inscriptions regulate the generation of tokens at transition firing. An output arc inscription can be the same variable of an input arc, or a function which generates specific output tokens.

2.3 Transitions

A *binding element* is a pair *(t,b)* consisting of a transition *t* and a *binding b*. A binding is an assignment of values to all the variables involved with the transition, i.e., the variables used in the arc inscriptions relevant to the transition. Transition *t* is enabled in a marking *M* if there exists at least a binding for *t*. A *guard* can be associated with a transition for controlling the binding/enabling process. For the transition to be enabled, all the input arc and transition guards must evaluate to true. An enabled transition can fire. Firing a binding element *(t,b)* withdraws tokens from the preset of *t* according to the binding *b*, and generates tokens in the postset according to the *t* output arc inscriptions.

Proceedings 17th European Simulation Multiconference
(c) SCS Europe BVBA, 2003 ISBN 3-936150-25-7

2.4 Timing aspects

A Rainbow model has a time notion (Jensen 1992-98) expressed by the value of a global clock (*model time*). In addition, tokens (i.e., colours) are time stamped. The time stamp of a colour reflects its generation time. Time stamped colours are components of *timed* multi-sets. The following is an example of a timed multi-set:

$$4'[a]\{1@49 \quad 2@50 \quad 1@52\} + 2'[b]\{1@49 \quad 1@50\}$$

The multi-set has four tokens of colour *a*, one with time stamp 49, two with time stamp 50 and the last one with time stamp 52, and two tokens of colour *b* respectively with time stamps 49 and 50.

To be acceptable, a binding element must be *time enabled*. A binding element is time enabled if it is composed of *ready tokens*. A token is ready if the global clock is greater than or equal to the token time stamp. Normally, the choice among ready tokens in a place is non deterministic (*unordered place*). Would there been multiple ready tokens for a given binding element, any one such a tokens can be selected to participate in the binding element. A time enabled binding element is characterized by its *enabling time*, i.e., the maximum value of the time stamps of the tokens involved in the binding element.

The global clock is automatically advanced when no binding element is time enabled at current time. In these cases, a binding element with minimum enabling time is chosen and the global clock adjusted to this value to ensure progress in model behaviour.

As in Generalized Stochastic Petri Nets (Marsan et al. 1984), Rainbow permits both *timed* and *untimed* (or immediate) *transitions* to be used in a model. Binding elements involving immediate transitions are always selected before binding elements of timed transitions. Immediate transitions can be assigned priority and probability values useful for conflict resolution (Marsan et al. 1987)(Ferscha 1994). The set of binding elements of immediate transitions having the highest priority is determined in the first place. Then, the actual binding element is selected in the set by a random choice according to transition probabilities.

Timed transitions are associated with a *delay* which affects the generation of tokens at transition firing. Firing a transition *t* at time τ is an instantaneous event whose effect is the creation of tokens in the postset of *t*, all time stamped with the value τ+*delay*. The delay of a timed transition can be deterministic or stochastic. A *delay function* can be attached to a timed transition in order to constrain the delay value on the basis of the selected binding.

2.5 Token selection policies

The set *P* of places of a Rainbow model is partitioned in two subsets: $P = rP \cup oP$, where *rP* denotes a set of classical unordered (or random) places, *oP* is a set of *ordered* (or *queue*) places (Bause 1993)(Poses). In an ordered place colours are ranked by ascending time stamps.

One of different *token selection policies* can be associated to an ordered place: FIFO-strict, FIFO-random, LIFO-strict, LIFO-random. According to FIFO-strict, a binding element with a queue place can only occur with tokens at the queue's head (oldest tokens). FIFO-random flexibly allows a binding to occur with the first matching colour starting from the head of the ordered token list. In a similar way are defined the LIFO-strict and LIFO-random policies which visit the token ordered list starting from the youngest tokens.

FIFO policies are the most natural in many simulation models. Figure 1 shows a typical scenario. Place W contains tokens representing units of work, which are assigned for processing to a given machine. Each machine can process one unit of work at time. For simplicity, the colour of a unit of work coincides with the corresponding machine number. Place M holds the machines available at current time. Transition $t_{process}$ models the task of a machine which processes a unit of work. Near to each place is indicated its current marking. The global clock is assumed to be 10. Finding a binding for $t_{process}$ means binding a machine colour to variable m so that the function f(m) returns a colour which is contained in W. For the purposes of the example f(m) can be the identity function: it just returns a colour from W equal to its argument. Function f(m) could be replaced by annotating the arc W-$t_{process}$ with a variable, e.g. n, and introducing a transition guard which checks that n and m are corresponding colours. Table 1 depicts the available binding elements at current time under FIFO and/or Random policies.

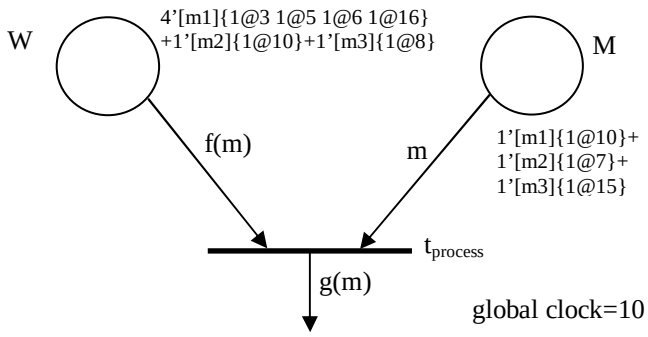

Figure 1. A typical scenario for token selection policies

W-Rand, M-Rand	b1: $\{m1@3\}_W$ $\{m1@10\}_M$, enabling time: 10 b2: $\{m1@5\}_W$ $\{m1@10\}_M$, enabling time: 10 b3: $\{m1@6\}_W$ $\{m1@10\}_M$, enabling time: 10 b4: $\{m2@10\}_W$ $\{m2@7\}_M$, enabling time: 10
W-FIFO-Strict, M-Rand (or FIFO-Rand)	b1
W-FIFO-Rand, M-Rand (or FIFO-Rand)	b1, b4

Table 1. Binding effects when applying different token selection policies

If the Random policy is adopted for both W and M places, four binding elements as possible for $t_{process}$. In particular, two bindings can be fired, one at a time and in any order: one chosen among b1, b2 or b3, and the other being b4. Generated tokens are controlled by the arc function g(m). All such tokens are time stamped by $10+\delta(t_{process})$, where $\delta(t_{process})$ is the (estimated) delay of $t_{process}$. The two firings occur at the same time horizon (global clock=10) to express the parallelism (infinite server semantics (Ferscha, 1994) of $t_{process}$) with which physical machines (e.g., m1 and m2) process distinct units of work.

The random policy does not force tokens in W or M to be processed according to their arrival time. Constraining work units to be processed in the arrival order is the responsibility of FIFO policies. However, FIFO-Strict for W would forbid, at current time, to fire other bindings except but b1. In addition, if the m1 colour in M is ready at a time greater than the global clock, no binding would then be available at current time for $t_{process}$, although machine m2 is ready from time 7 and b4 is potentially ready for firing. FIFO-Random for W and Random or FIFO-Random for M, would constrain machines m1 and m2 to process the available units of work having minimum time stamp (see b1 and b4 bindings).

The design of the timing model of Rainbow purposely separates time management from functional aspects of a net model captured by arc inscriptions. From this point of view, an input arc

inscription can only express requirements for colour selection. The use of token time stamps and the system time advancement rule are under implicit control of the underlying executor which has responsibility in applying place selection policies.

3. IMPLEMENTATION STATUS

An implementation of Rainbow was achieved in Java through a graphical tool. The following are some points of the developed tool:

- it allows editing, debugging, simulation and analysis of CPN models. Both step-by-step execution and checkpoints (e.g., desired markings in selected places) are supported
- it hosts both coloured and non coloured nets. Non coloured nets rest on tokens which consist of the time stamp only
- it allows graphically to distinguish between unordered (default) and ordered places (split circles). A property of an ordered place concerns its selection policy
- it hosts an executor which is devoted to sequential simulation of a model. The executor uses Java reflection for accessing and invoking user-defined model functions.

Distributed simulation of a Rainbow model can be required by the computationally very expensive (in time and space) task involved with binding element processing. The critical factor is *binding calculation*. Building the bindings corresponding to a transition *t* requires in general exhaustively enumerating all the possible assignments of values (according to colours and time stamps available in the preset of t) to variables involved with *t*. A variable can be used alone on an arc or as a function parameter of an input or output arc of *t*. The same value of the variable must consistently be used in all its occurrences in a binding. Binding calculation is responsible for identifying all the candidate bindings existing at current time for any transition. Among alternative bindings, a random choice eventually selects the binding to fire. Ordered places and associated token selection policies obviously can speedup the relevant binding calculation process, since they restrict the possible proposed bindings.
Distributed simulation is currently dealt with externally to the Rainbow graphical tool and depends on a specialized version of the executor built on top of an agent-based Time Warp mechanism (Furfaro *et al*., 2002b). Key points of the distributed executor are the following:

- it allows a large model to be partitioned into a collection of subnets/LPs allocated for execution one per physical processor of a networked system. The Rainbow tool makes it possible to visually decompose a net model into cuts and to save them on disk as part of the model data representation. Actually, model data representation can be archived according either to standard Java serialization or XML and associated DTD. The model data representation is parsed by a *director* agent which configures and controls the distributed simulator
- it benefits from the features of Temporal Uncertainty Time Warp –TUTW- (Beraldi and Nigro, 2001)(Beraldi *et al*., 2002) which permits temporal uncertainty to be exploited in general distributed simulations. TUTW adopts an event delivery strategy where the occurrence time of an event is specified by a time interval and not a punctual timestamp. All of this augments the model event parallelism (events having overlapping time intervals are concurrent) and has the potential of improving the simulation performance since the control engine is given some flexibility in the event resolution, i.e., choosing the actual time stamp of events at dispatch time. Temporal uncertainty allows to relax in part the synchronization constraints. TUTW, though, is able to keep causality among concurrent events using Lamport "happens-before" relationship. For many simulation applications, experiments have shown that TUTW is capable of improving performance of the distributed simulator with respect to the case temporal uncertainty isn't used, without necessarily compromising the accuracy of the results.

4. A SIMULATION MODEL

The following describes a complex and scalable simulation model with the goal of illustrating the practical use of Rainbow and its graphical tool. The model is based on a non coloured TPN model proposed by Zuberek (Zuberek, 1999)(Zuberek, 2002) for studying the influence of long-latency memory accesses in distributed-memory multithreaded multiprocessors (DM-MM). The simulation model was actually experimented for exploring the effects on the cpu utilization of component heterogeneity vs locality of memory references. All of this can be accomplished without changes in the model topology.

4.1 A multiprocessor multithreaded model

A DM-MM system with *nxn* processors (or nodes) interconnected by a bi-dimensional torus-like switching network is assumed (see Figure 2). Each processor can communicate directly with its four neighbours. An outline of the node architecture is portrayed in Figure 3).

Each node has a local memory and two network interfaces allowing concurrent send/receive operations. Any processor can issue a memory request which can be directed to local memory or to the memory module of some remote node, which can be reached through the interconnecting network according to a suitable path, e.g., one with shortest distance. Through the outbound interface is routed all the outgoing traffic concerning remote memory requests originated in this node, or the results of memory operations asked by remote processors to the memory module in this node. Through the inbound interface occurs all the incoming traffic consisting of remote originated requests to the memory of this node, as well as the results of remote request operations which come back to the originating nodes.

Each processor has a queue of ready threads. Whenever a long-latency memory operation is started at this processor, a *context switch* is accomplished as follows: first the current thread is suspended, then the memory operation is forwarded to the relevant memory module (local or remote); finally, processor execution is resumed by selecting another thread, if there are any, from the ready queue, and transferring the control to its next instruction. When the result of this memory request is received, the corresponding thread changes its status from "suspended" to "ready" and it is added to the ready queue waiting for dispatch.

4.1.1 Model parameters

The runlength of a thread, ℓ_t, represents the number of instructions executed, on the average, between context switches. This parameter is directly related to the probability that an instruction raises a long-latency memory access. Two other important parameters are p_ℓ and $p_r=1-p_\ell$, that is respectively the probability that a memory access is to local memory or is directed to a remote memory node.

The values of p_ℓ and p_r control the amount of switching network traffic and congestion vs local node memory accesses. Finally, the (average) number of available threads, n_t, influences the utilization of system components and the overall system performance. In the Rainbow model of Figure 4, the value of this parameter is assumed to not change with time.

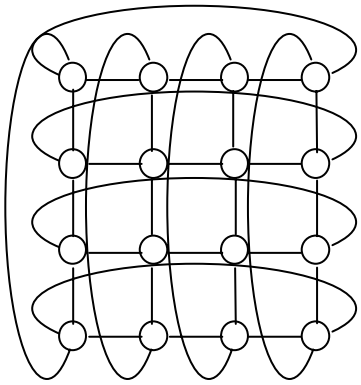

Figure 2. Switching network

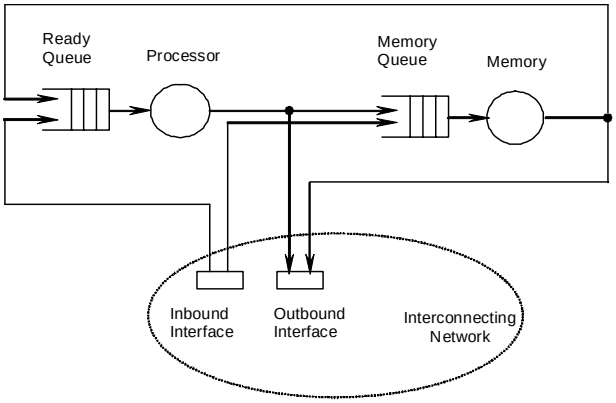

Figure 3. Node architecture

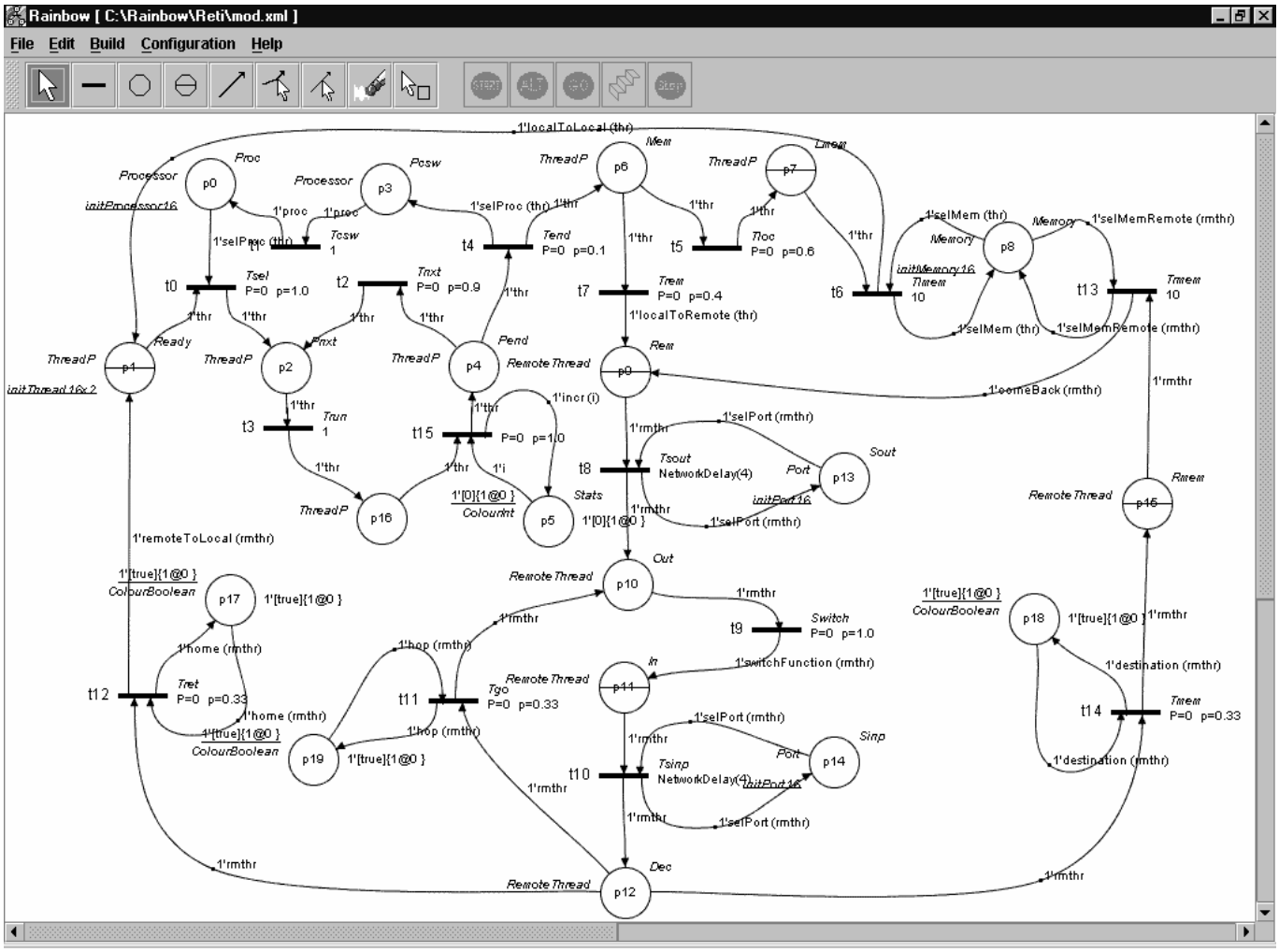

Figure 4. A DM-MM Rainbow model

4.1.2 A DM-MM Rainbow model

The model shown in Figure 4 is logically organized into four sections: a processing subnet (places p0 to p4 and p16, transitions t0 to t4), a memory subnet (places p6 to p9 and p15, transitions t5, t6 and t13), a switch subnet (places p9 to p14, p7 to 19, transitions t7 to t12, t14), a statistics subnet (place p15 and transition t15). Component replications in the physical system (Figure 2) are achieved by colour replications in the fixed model topology. For instance, all the available processors are initially represented by colours fed to place p0 (also named *Proc*). Similarly, all the threads in the ready queues of the various processors are mapped on to colours in the FIFO-Random place p4 (or *Ready*), all the switch board colours are kept in p13 and p14 places and all the memory modules are represented by colours in the p8 (*Memory*) place. Scalability of the model is automatically ensured by adjusting the initial number of colours in places p0, p4, p8, p13 and p14.

When a processor is available in p0 for processing a thread from p1, a thread token is generated in p2. Transition t3 (*Trun*) simulates the execution of one instruction. Its delay is 1. All other delays in the model are expressed in terms of number of instructions. Thread execution is simulated by the loop p2-p16-p4 and transitions *Trun*, t15 and *Tnxt*. Transition t15 serves only for statistical purposes. At each firing of t15 (or equivalently of *Trun*) the counter in p5 gets incremented. Place p4 is a free-choice. Immediate transitions *Tnxt* and *Tend* represent respectively the

execution of a non memory accessing instruction or a memory request which implies a context switch. In the latter case, a processor token is deposited in *Pcsw* (or p3) with the timed transition *Tcsw* modelling the actual thread context switch. At the end of the context switch the processor becomes again available for processing the next thread from its ready queue. Probability of *Tnxt* (and then of *Tend*) mirrors the (average) runlength of a thread.

Place *Mem* (p6) is a free-choice. Immediate transitions *Tloc* and *Trem* represent respectively an access to local memory or an access to the memory module of a remote processor. The probability of *Tloc* (and then of *Trem*) captures the locality of memory accesses vs remote memory accesses. Local requests are held in the FIFO-Random place *Lmem* waiting for the memory (place *Memory* or p8) to be available. An actual memory operation is modelled by *Tlmem* timed transition. Remote thread requests are maintained in the *Rmem* place and served by *Trmem* timed transition. Firing frequencies of *Tlmem* and *Trmem* are function of token multiplicity respectively in places *Lmem* and *Rmem*. After being served, a remote thread request is routed into *Rem* for it to engage the coming back to home path through the interconnection network.

The outbound/inbound interfaces of processor nodes are respectively modelled by *Tsout* and *Sout* and *Tsinp* and *Sinp*. The switching network is represented by places *Out* and *In* and transition *Switch*. A remote memory request which has been transmitted through the switch is received in the *Dec* place from which it can proceed (next *hop*) in the network (transition *Tgo*), or it just arrived at the destination node (*Tmem* transition) or at its home node (transition *Tret*). In the physical system such decision depends on the current position within the transmission path toward the target node which is the remote node during forward movement or the home node during backward movement.

The design of the DM-MM model was driven by the desire to reproduce "as close as possible" the behaviour of the actual system. This in turn motivated the adoption of FIFO-Random policy for ordered places *Ready*, *Rem*, *In*, *Lmem* and *Rmem*, and the introduction of suitable colour sets and arc inscription functions. For brevity, the following only provides an informal description of colour sets and arc functions. The colour set is indicated at the left of a place in Figure 3.

A processor is encoded by its *id* (an int colour). A thread keeps the processor number to which it is assigned (*ThreadP* colour set) and the time at which it enters the ready queue. A *Memory* colour is identified by its processor number too. A long-latency memory request is modelled by a *RemoteThread* colour. A remote thread moves along the network and carries such information as: the originating processor, the destination processor, shortest_path to destination, and current processor position in the switching network. The path to destination is concretely expressed by the number of hops to be taken respectively at North, East, South and Ovest from current position. The *localToRemote* function of arc *Trem-Rem* transforms a local thread into a remote thread by choosing a destination node and a path to follow for reaching it. The *remoteToLocal* function on the arc *Tret-Ready* converts a remote thread to its local representation. The *selPort* function selects the in/out switch port of the current processor a remote thread is passing through. It is ensured that *Tsout* and *Tsinp* have a single server semantics.

The choice among *Tmem*, *Tgo* and *Tret* transitions from *Dec*, is made by checking current position of the remote thread. In the case of an intermediate position, the *hop* function on the arc p19-*Tgo* enables *Tgo* and the thread proceeds for a next hop in the interconnection network. Similarly, *destination* function on p18-*Tmem* and *home* function on p17-*Tret* respectively enable *Tmem* or *Tret* in the case the current position of the remote thread coincides with the target node (destination or home node). Only one transition among *Tret*, *Tgo* and *Tmem* can be enabled at a same time. Places p17, p18 and p19 hold a boolean colour which always is true. Only in the case the corresponding checking function (*home*, *hop* or *destination*) returns the true colour is the transition enabled. After a firing of *Tret*, *Tgo* or *Tmem*, the true colour is reconstructed in its corresponding place.

The *switch* function on the arc *Switch-In* is responsible of updating the current position of a remote thread after a hop.

The *localToLocal* function on the arc *Tlmem-Ready*, saves in the thread token the time when it was generated (its time stamp) by *Tlmem*. Such information allow estimating thread waiting time in the ready queue.

Functions *selMem* on arc *Memory-Tlmem* and *selMemRemote* on arc *Trmem-Memory* select the memory colour of the processor respectively specified by local or remote thread. It is ensured that a memory module is always handled according to the single server semantics.

The *NetworkDelay* function attached to *Tsout* and *Tsinp* transitions is an example of a function which can set the delay of a timed transition according to model parameters. The function receives as an argument the number of switching boards which are assumed to be of low speed. *NetworkDelay* provides the context for a customization of the built-in *getDelay* function which in the case of *Tsout* and *Tsinp* was redefined in order to receive the remote thread selected from *Rem*. *getDelay* can thus regulate the actual delay of *Tsout* or *Tsinp* according to the managed remote thread, and its current position.

4.1.3 Simulation experiments

The DM-MM model was experimented in a case for evaluating the influence of component heterogeneity on the cpu utilization vs the probability of local/remote memory accesses. Heterogeneity can occur in the computing power, switching board and memory module of processors. In the following, for demonstration purposes, some executions carried out by considering only switch heterogeneity are reported. Different runs refer to varying the amount of node switches which introduce additional communication delays. A low speed switching board is assumed to double the transmission delay.

Figures 5 to 7 depict the estimated model cpu utilization for a DM-MM with 16 processors, and a probability p_ℓ that a memory access is local (see transition *Tloc* in Figure 3) respectively of 0.4, 0.6 and 0.8. In the experiments, the simulation time was 10^4, the thread number n_t was varied from 2 to 10, and the number of low speed switching boards was varied from 0 (homogeneous case of high speed boards) to 16 (homogeneous case of low speed boards). Moreover, the runlength of threads was set to 10 (probability of *Tnxt* 0.9 and of *Tend* 0.1).

Figures 5 to 7 confirm that the critical factor on the cpu utilization is the p_ℓ value. When p_ℓ is 0.8 a good cpu utilization is achieved even when processors are loaded with a small number of threads.

5. CONCLUSIONS

Rainbow is a formalism based on Coloured Petri Nets (Jensen, 1992-98). It was designed for supporting modelling and simulation of complex systems. Prototyping tools were achieved which allow to experiment with simulation models both in a centralised and a distributed framework on top of a Time Warp mechanism (Beraldi and Nigro, 2001)(Beraldi *et al*., 2002). A key factor of the Rainbow project is the adoption of Java both as the net annotating language and as the tools implementation language. All of this simplifies the use of the modelling language and makes the achieved tools totally portable almost on every platform.

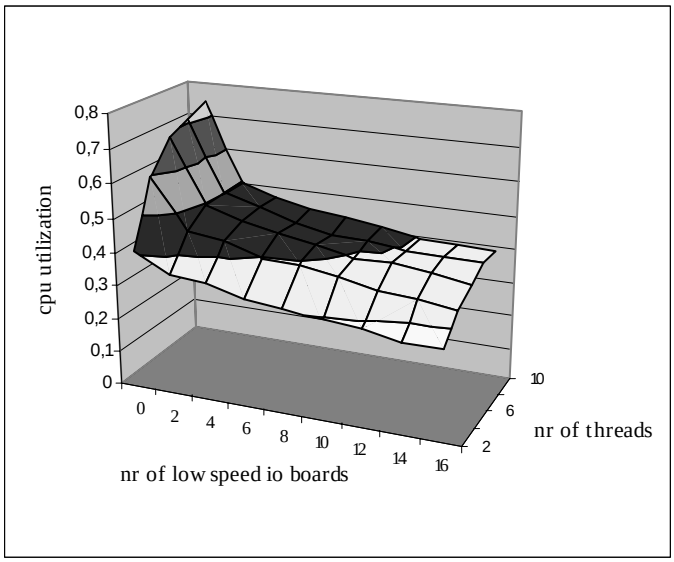

Figure 5. Cpu utilization, 16 processors, p_ℓ =0.4

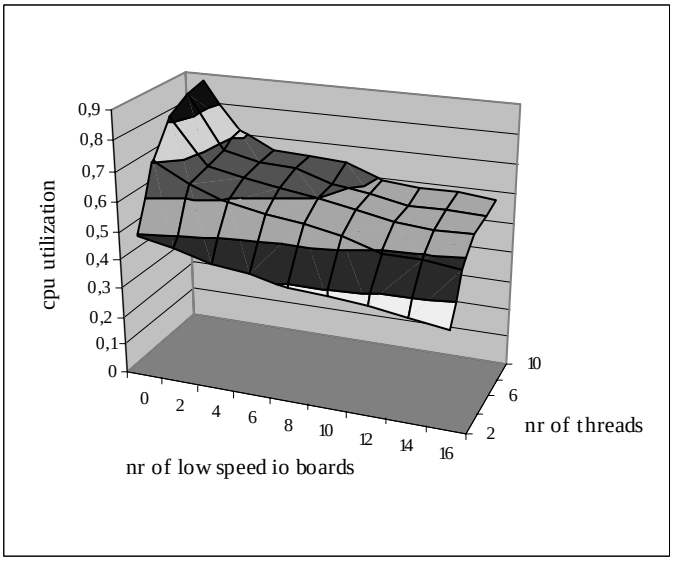

Figure 6. Cpu utilization, 16 processors, p_ℓ =0.6

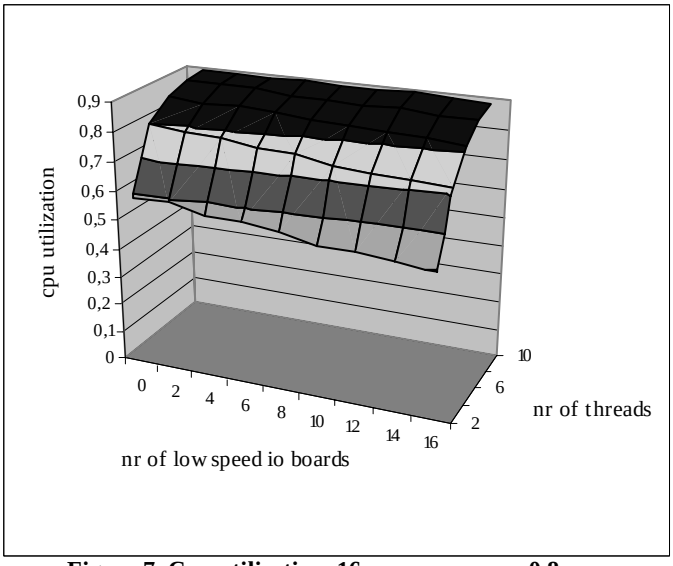

Figure 7. Cpu utilization, 16 processors, p_ℓ =0.8

On going and future work is geared at

- optimising the Rainbow executor by improving the binding calculation process which critically affects the simulation performance. From this point of view the aim is to replace the actual linked-list representation of colour multi-sets in places by more efficient data-structures and algorithms (Mortensen, 2001)
- extending the graphical tool with aspect-oriented features (CACM, 2001)(Furfaro *et al*., 2002a), i.e., the possibility of adding to a model a crosscutting specification (*monitor*) useful for monitoring and analysing the simulation (Wells, 2002). A monitor would catch selected event occurrences in the model and make necessary book-keepings for statistics computation. Aspect-oriented monitors would be transparently attached to a model by avoiding explicit subnets for statistical computations to be introduced

6. REFERENCES

(Bause, 1993) F. Bause. Queuing Petri Nets: A formalism for the combined qualitative and quantitative analysis of systems. *Proc. of the Int. Workshop on Petri Nets and Performance Models*, pp. 14-23, Tolouse, October, 1993.

(Beraldi and Nigro, 2001) R. Beraldi, L. Nigro. A time warp mechanism based on temporal uncertainty. *Transactions of the Society for Modelling and Simulation International*, **18**(2), June, pp. 60-72, 2001.

(Beraldi et al., 2002) R. Beraldi, L. Nigro, A. Orlando, F. Pupo. Temporal Uncertainty Time Warp: An agent-based implementation. *Proc. of 35th Annual Simulation Symposium*, 14-18 April, San Diego, CA, pp. 72-79, 2002.

(CACM, 2001) *Communications of the ACM*. Aspect-oriented programming. **44**(10), pp. 29-99, October, 2001.

(Ferscha 1994) A. Ferscha. Concurrent execution of timed Petri nets. *Proc. of 1994 Winter Simulation Conference (WSC94)*, Lake Buena Vista, Florida, USA, pp. 229-236, 1994.

(Furfaro et al., 2002a) A. Furfaro, L. Nigro, F. Pupo. Aspect oriented programming using actors. *Proc. of 22nd IEEE Int. Conference on Distributed Computing Systems Workshops, Aspect Oriented Programming for Distributed Computing Systems (AOPDCS 2002)*, Vienna, Austria, 2-5 July, pp. 493-498, 2002.

(Furfaro et al., 2002b) A. Furfaro, L. Nigro, F. Pupo. Distributed simulation of Timed Coloured Petri Nets. *Proc. of Sixth IEEE Int. Workshop on Distributed Simulation and Real-Time Applications (DS-RT 2002)*, 11-13 October, Fort Worth (Texas), IEEE Comp. Society, pp. 159-166, 2002.

(Jensen et al., 1996) K. Jensen, S. Christensen, P. Huber and M. Holla. (1996). Design/CPN. A reference manual. Computer Science Department, *University of Aarhus*. Online: http://www.daimi.aau.dk/designCPN/, 1996.

(Jensen, 1992-98) K. Jensen. *Coloured Petri Nets - Basic concepts, analysis methods and practical use*. Vol. **1, 2, 3**. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992-98.

(Jensen et al., 1996) K. Jensen, S. Christensen, P. Huber and M. Holla. Design/CPN. A reference manual. Computer Science Department, *University of Aarhus*. Online: http://www.daimi.aau.dk/designCPN/.

(Kummer et al., 2002) O. Kummer, F. Wienberg, M. Duvigneau (2002). Renew-User Guide. http://www.informatik.uni-hamburg.de/TGI/ renew/renew.html.

(Marsan et al., 1984) M.A. Marsan, G. Balbo, G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of systems. *ACM Transactions on Computer Systems*, **2**(2), pp. 93-122, 1984.

(Marsan et al., 1987) M.A. Marsan, G. Balbo, G. Chiola and G. Conte. Generalised Stochastic Petri Nets revisited: random switches and priorities. In *Proc. of the 2nd Int. Workshop on Petri Nets and Performance Models*, pp. 44-53, IEEE-CS Press, 1987.

(Mortensen, 2001) K. H. Mortensen. Efficient data structures and algorithms for a coloured Petri nets simulator. In: Kurt Jensen (Ed.): *3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pp. 57-74. DAIMI PB-554, University of Aarhus, August 2001.

(Murata, 1989) T. Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, **77**(4), pp. 541-580, 1989.

(Poses) Poses on-line: http://www.gpc.de

(Wells, 2002) L. Wells. Performance analysis using Coloured Petri Nets. *Proc. of MASCOTS 2002*, 11-16 October, Fort Worth (Texas), pp. 217-221, 2002.

(Zuberek, 1999) W.M. Zuberek. Performance modeling of multithreaded distributed memory architectures. *Proc. of 2nd Workshop on Hardware Design and Petri Nets*, Williamsburg, VA, pp. 63-82, 1999.

(Zuberek, 2002) W.M. Zuberek. Approximate simulation of distributed-memory multithreaded multiprocessors. *Proc. of 35th Annual Simulation Symposium*, 14-18 April, San Diego, CA, pp. 107-114, 2002.