Genetic Regulatory Mechanisms by means of Extended Interactive Petri Nets

António Pina, João Miguel Fernandes, Ricardo Jorge Machado
Email: {pina,miguel,rmac}@di.uminho.pt
Dep. Informática, Universidade do Minho
Largo do Paço, 4709 Braga codex, Portugal

ABSTRACT

In our work we have chosen to integrate formalism for knowledge representation with formalism for process representation as a way to specify and regulate the overall activity of a multi-cellular agent. The result of this approach is XP_iN, another formalism, wherein a distributed system can be modeled as a collection of interrelated sub-nets sharing a common explicit control structure. Each sub-net represents a system of asynchronous concurrent threads modeled by a set of transitions. XP_iN combines local state and control with interaction and hierarchy to achieve a high-level abstraction and to model the complex relationships between all the components of a distributed system. Viewed as a tool XP_iN provides a carefully devised conflict resolution strategy that intentionally mimics the genetic regulatory mechanism used in an organic cell to select the next genes to process.

1. INTRODUCTION

We believe that the application of biological ideas to novel software and hardware design can possibly lead to a non conventional style of approach on the design of distributed systems. Following this perspective we developed MC^2 [1] – cellular computational model – to explore what seems to be central to the concept of network, typically consisting of a large number of elements, similar or identical, mutually interconnected by a homogeneous pattern of interconnections.
In order to preserve the integrity of the informational structure required to our representation of a distributed system, as well as to explicitly control the activities at the component level, we present XP_iN – an Interactive Extended Petri Net model. It combines production systems formalism, to encode the knowledge of a certain problem domain, with Petri Net formalism to model choice and coordination among alternative actions and concurrent activities.
In our programming methodology, XP_iN may be viewed as providing a general *genetic regulatory mechanism* suitable to represent and coordinate the overall structure and the dynamic properties of a distributed system [2]. It may be used as a mechanism that allows each cell to determine the next sections of code to run [3] and as a way to influence the code executed by the totallity of the cells in a multi-agent system.
Next we attempt to give a short description of the novel model discussing some of the basic characteristics of a distributed system viewed as an agent, the analog to a multi-cellular biological organism.

Cellular Computation Model

An agent in MC^2 is a distributed system modeled as a set of independent cells, or elementary agents, which may communicate through asynchronous messages to produce a global behavior. This set of cells may dynamically evolve by creating new cells, or building new structures of communication channels.
Within a single cell, generally defined by a large number of independent states and actions, several executions (processes) may simultaneously occur, being each process represented as a single address space with one or more threads of control.

Knowledge representation

Due to the event-driven nature of interaction between cells, the dynamic properties of an agent information-processing capacity are better encoded declaratively as opposed to procedurally.
Taking this approach, we can view each process in a cell as being modeled by a production system [4] that determines the proper structure for representing the knowledge applicable to a certain problem domain. Each production system consists of a set of rules designed to selectively respond to different combinations of inputs and data to produce an adaptive behavior in diverse environments.
As in natural biological systems where the genome appears to carry all the information needed to describe the structural and dynamic properties of an individual, in MC^2, to code the overall properties of an individual cell or multi-cellular agent, we use production systems. This suggests a functional equivalence between the rules of a production system and the genes of a genome.

Control representation

Since some of the agent's behaviors will include long sequence of actions, it must be able to coordinate the firing of several rules by explicitly evoking its successors. But if this path is taken, production rules autonomy is lost, which limits the ability of an agent to

grow by augmenting and refining its knowledge about the environment.
The most obvious way to mitigate the difficulties of using production systems without loosing their advantages, is to find a mechanism that can preserve the agent capability to be responsive to demands of its environment, maintaining at the same time the continuity in its behavior without sacrificing rules autonomy and modularity. Adding control structure to a production system by appending to each rule some *a priori* information may give some clue as which rules are likely to be activated.

2. OBJECTIVES OF THE WORK

In MC^2 we have chosen to integrate a formalism for knowledge representation – production systems – with a formalism for process representation – Petri Nets – as a way to specify and regulate the overall activity of multi-cellular agents. The result of this approach is XP_iN, another formalism, wherein a multi-cellular agent can be modeled with a collection of interrelated sub-nets which are production systems sharing a common explicit control structure. Each *sub-net* represents a process, a system of asynchronous concurrent threads modeled by a set of transitions. Since a process is the equivalent of a production system, a transition is really a "home" for the rule describing a gene.
XP_iN provides a carefully devised conflict resolution strategy that intentionally mimics the *genetic regulatory mechanism* used in an organic cell to select the next genes to process. The approach will ensure the correctness and efficiency of the agent program largely dependent on the number of rules to fire and the policy used to choose which rules to evaluate and in what order. However, in the context of a distributed system a single control construct used in isolation provides little power to model other cellular phenomenon.
An agent as a distributed system needs the ability to create hierarchies of control that allow many distinct processes divided among several cells to mutually influence the system global behavior. XP_iN offers a solution to this problem based on the perspective that the flow of control of a multi-cellular agent may also be represented and coordinated, by using the basic regulatory mechanism.
Following this perspective we may view a distributed system as being represented by a collection of several sub-nets linked together through a general interaction mechanism. The sub-nets represent local control and the activity at each elementary site.
The interaction mechanism is based in a message-passing model of communication that allows any pair of threads to communicate, whether they reside in the same cell or in distinguished cells.

3. THE XP_iN MODEL

Distributed systems tend to be complex, with many distinct components that may communicate to establish dynamic relationships. In this context, support for a programming methodology that facilitates distributed programming is especially important.
Programming using the process-based message-passing paradigm, while straightforward in principle, typically forces the programmer to explicitly manage all task creation and synchronization in a program. These tasks can be unmanageable in large systems with hundreds or thousands of concurrent activities.
The design of XP_iN has pursued two main objectives:

- to enhance performance and to simplify the development of distributed programs, by combining in a uniform approach the principles of multithreading, data-flow computing and message-passing.

- to model the complex relationships between all the components of a distributed system, by combining local state and control with interaction and hierarchy in a high-level abstraction

In systems characterized by a strong interaction with the environment, due to the event-driven nature of program applications, workload allocation and scheduling are critical when resources are not dedicated or the demands are not predictable. XP_iN attempts to automate thread instantiation and message passing in order to address these issues by providing a scheme that delays binding of work and scheduling only when data is available. This also enables a more optimal assignment of resources to tasks.
The use of data-driven, to mask some of the inherent complexity of programming with asynchronously-communicating sequential entities, essentially requires entities to be declared with named data dependencies. When the data dependencies for a certain entity are satisfied, the underlying software automatically schedules and executes an entity of that type.
In our work the principles of data-driven have been converted into an equivalent representation by means of Petri nets through the use of XP_iN.
The conversion is easy to obtain if we consider that the entities in a sub-net may specify its data dependencies in terms of other entities – i.e. they indicate that their activation requires the availability of a set of specific messages (data or control) issued upstream.
All the running entities of an application inform the scheduling sub-system when they generate those messages. When a complete set of messages, bounded to an entity, has been formed, the corresponding thread is spawned at the most suitable location, as determined by the state of the system.
With Petri nets, the programmer is not limited to sequential code, but can also describe parallel processes, fork & join mechanisms, synchronization between tasks and so on.
The model proposed by XP_iN benefits from PNs, being at the same time straightforward and comprehensive enough to represent distributed application programs.
Programming with XP_iN does not require explicit thread creation or message passing, thus simplifying the complex state management task required by parallel distributed applications. In order to achieve a high-level abstraction and modularity it supports the construction of composite program elements by considering the design of a parallel application as being processed at two different levels of representation:

- microscopic - here we decide the functions to execute when the transitions in a sub-net are selected to "run". Afterwards we describe, in terms of XP_iN, the existence of data dependencies between all the transitions in a local sub-net.
- macroscopic – here we define a separate and explicit control structure by means of a hierarchy of sub-nets. At this level, the activation of a local transition will be propagated to the global system, indirectly influencing or controlling the activities of other sub-nets.

Basic definitions

The design of XP_iN retains and is compatible with the essence of Petri nets, both including the basic concepts of transition, place, token and arc.
Places represent queues, whereas the *transitions* expect the arriving of synchronization data elements to become active. *Tokens* may be either signals with only control significance or events with associated pieces of data. *Arcs* are direct edges into places and transitions representing the channels for the exchange of tokens between the two types of nodes.
The XP_iN has a structural restriction: each place has only one input transition and one output transition, what suggest that XP_iNs may be regarded as *marked-graphs* [5].
As a result of our work, a more in-depth analysis reveals extended capacities, some already introduced by other PN models and others only proposed, to our knowledge, within the XP_iN model. To ease the task of understanding our work, next we present several basic definitions illustrated by an overloaded graphical representation of a XP_iN transition (see Fig. 1).

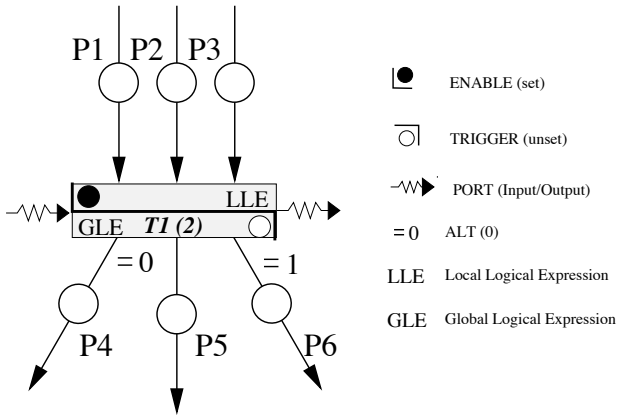

Figure 1: A transition in XP_iN.

Running/Inactive state - a transition is fired when a set of named conditions are satisfied; its action does not occur instantaneously, it remains in a `running state` until it terminates returning to the `inactive state`.

Firing - before entering the `running state` the transition removes one token from each marked input place; if an input place is empty, it remains empty.

TEA - a local variable composed by three binary data fields (`TRIGGER`, `ENABLE`, and `ALT`) whose value is the completion code returned by the transition's action.

Trigger - the value of the `TRIGGER` data field may be used to allow/prevent the generation of tokens after the transition completion; thus by setting/unsetting the value of the `TRIGGER` data field we enable/disable the evolving of the net.

Guard - a constant expression attribute associated to an arc that imposes a special restriction to the passage of tokens; when the expression matches the value coming from the `ALT` data field of the input transition the correspondent output place is filled; used to dynamically select the evolving direction of the net.

Local trigger - the value of the `ENABLE` data field offers the possibility to overcome the restriction to fire; when the `ENABLE` field is set the transition is defined as `Candidate` irrespectively of the evaluation of any other local conditions; it also generates a local trigger, enabling its own scheduling, even when the `TRIGGER` data field is unset.

Port - a low-level communication mechanism that uses a one-way data channel to link a transition to any place in a local or a remote sub-net; it may be used to asynchronously send/receive pieces of data or control events through the established connection.
Data sent through a channel is put into a buffer located at the reception transition by the underlying system; the data in the buffer must be explicitly received during the `running state` by executing an appropriate primitive function. A transition may have at most one `input port`/`output port` and associate to it a `SYNC_IN`/`SYNC_OUT` attribute. Setting a `SYNC` attribute forces the correspondent `port` to be registered as an element of the sub-net. When a port is registered the system guarantees the automatic generation of the control tokens in both sides of the established communication channel. The restriction of using marked graphs does not apply for input/output `ports`.

The above definitions resume the most important features of the model, however it includes other characteristics already presented in fig. 1 which it is worth examining.
Besides the usual graphical representation of the XP_iN entities on the net, we use a special notation based on bullets (black and white) to facilitate the understanding of the net behavior and to specify the functionality associated with transitions. The bullets may only be placed at the top-left (enable bullet) and bottom-right (trigger bullet) of a transition, in a direct correspondence with the data fields `ENABLE` and `TRIGER`. The presence of a black/white bullet indicates that the respective data field should always be interpreted as if it were set/unset; the absence of a bullet means that its data field maintains the value returned in the last completion code.

Scheduling definitions

The extensions we made to standard Petri net strongly affect the rules that determine the firing of transitions. Therefore, to determine the transition activation requirements it is necessary to introduce the following definitions:

Accepted/Rejected - a transition is said to be accepted/rejected when all the conditions needed to fire

including local and global dependencies are successfully/unsuccessfully accomplished.

LLE - an optional `local logical expression` refines the standard rules used to decide the firing of a transition; the terms in the expression correspond to the existence or absence of tokens in its input places.

GLE - an optional `global logical expression` to be satisfied by a transition before being accepted; the terms represent the running state of other transitions.

Candidate - used to classify a transition that evaluates to true the named conditions or its `LLE` (if present); alternatively a transition may become a `Candidate` whenever the `ENABLE` data field is set by its last returned completion code.

Priority - a constant numeric value associated with each transition used as a global condition to suspend the firing of an accepted transition, until no other transition with a higher priority lasts in a running state.

Optional conditions – additional conditions may be used to restrict the number of accepted transition; used to improve the selectivity response of a system or to support other conflict resolution strategies.

Activation of transition
A transition to be activated must pass through several filters until being classified as an `accepted` transition; it then removes tokens from their input places and unsets the `ENABLE` data field.
The transition remains in a `running state` for an indefinite period always returning with a completion code value, used to update the local `TEA` variable.
The standard firing mechanism imposes that after the firing of a transition all its output places are filled with control tokens locally generated; when the value of the `TRIGGER` data field is unset the mechanism is disable; consequently no control tokens are generated, preventing the net to evolve.
The tokens generated after the triggering of an input transition are not automatically put in their output places. For each output place, the `Guard` attribute must be tested against the value of the `ALT` data field of the input transition. The occurrence of a match enables the correspondent output place to be filled; the opposite situation (mismatch) disables the passage of tokens into the respective places. The use of a `Guard` expression permits to decide the evolving direction of the net to be influenced at run-time by the `ALT` data field values of the input transitions. In the absence of `Guards` the passage of tokens after triggering retains the standard interpretation.
In order to define a systematic approach to the rules governing the activation of transitions we next describe the basic algorithm using the flowgraph in fig. 2; boxes represent the successive states associated with a transition and diamonds the conditions to evaluate.

4. XP_iN VERSUS PN

In an attempt to explain the importance of the extensions we made to the standard PN, we summarize below some of the most relevant contributions

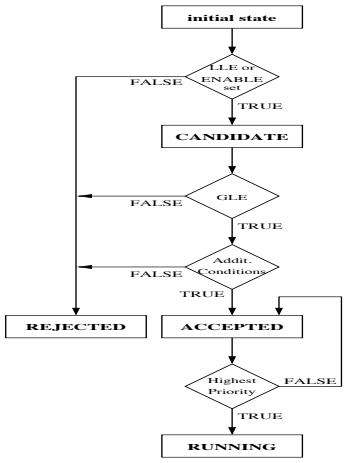

Figure 2: Activation of a transition.

of XP_iN. Some of the novelties being equivalent to other already proposed extensions made by other authors are not presented.
The completion code returned by a transition is copied into `TEA` and it may be used to enrich PNs with a dynamic mechanism for structural reconfiguration. The extension allows the same XP_iN to be simulated under different structural configurations by tuning the values of the data fields, without introducing or deleting arcs, places and transitions.
The `ALT` data field adds the capability to dynamically select alternative paths for the evolving of the net; it is equivalent to the arc expressions of Colored Petri Nets (CPN) [6]; the moment for evolving the net may also be decided at run-time by setting the `TRIGGER` data field.
At last, a transition whose `ENABLE` data field is permanently set will run forever irrespectively of the marking of its input places.
Based on an XP_iN transition described by one input place and a left output place with `guard` = 0 and a right output place with `guard` = 1, we show in fig. 3 how its structural configuration changes whenever the value of TEA changes.

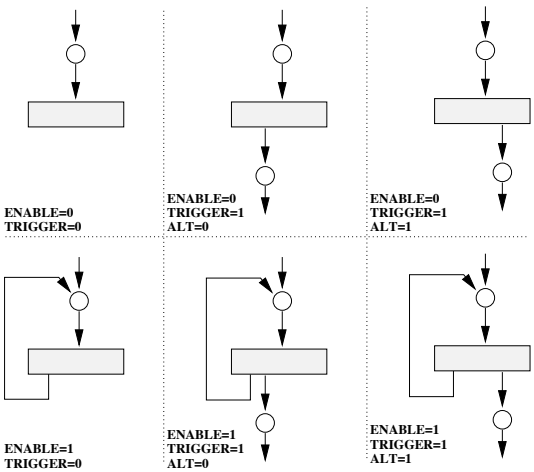

Figure 3: TEA data fields converted into PN.

The structural restriction of using marked graphs does not limit the description power of XP_iN. The use of `LLE` simplifies the representation of the connections between transitions and the respective input places, avoiding in some situations the use of complex sub-nets to represent those connections.
We show in fig. 4 how the expression $LLE = P1.\overline{P2} + P3$ applied to the single extended transition in fig. 1 is represented by a PN using inhibitor arcs between place `P2` and transitions `T11` and `T12`.

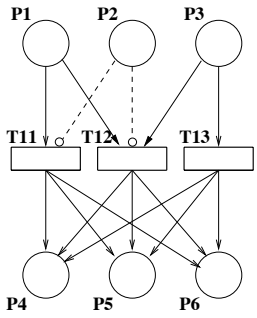

Figure 4: LLE converted into PN.

The conversion was made considering that, if the `LLE` of a given transition contains a negated place as a literal in its constitution then equivalent representation in PN have to be implicitly designed with an inhibitor arc between the place and the transition. In this case we also have to decide which transition to fire if a conflict occurs; transition `T12` must be forced to fire when all the transitions are enabled.
Since a PN with inhibitor arcs gives the capability to test for zero, XP_iN can model any system with the computational capacity of a Turing machine. In addition to this property the `LLE` extension also offers the possibility to specify fireable transitions that would not be fireable in PNs and to reduce the total number of nodes.
A `Candidate` transition may not be accepted to fire (`rejected`) when its `GLE` is evaluated to false; the terms of `GLE` represent the running states of the transitions. `GLE` is typically used to restrict the firing of transitions whose combination of `running states` is not allowed to occur simultaneously; as it happens with `LLE`, in some situations it also avoids the use of complex sub-nets to represent the same expression in PN.
`Priorities` are used to delay the firing of `accepted` transitions until all higher-priority `running` transition return to the `inactive state`; typically used to order the access to shared resources such as devices and variables, or to favor the execution of privileged functions, such as a system call. If correctly used, priorities allow the space-state explosion (the major problem of PNs) to be reduced and solve some non-deterministic situations; lower priority `accepted` transitions are not taken into account when defining the combinations for the state of the reachability graph.

CPN Representation

The design of XP_iN as an extension of PNs could presumably benefit from a well-established conceptual model for dealing with concurrency and take advantage of all the related analysis tools already in existence. However some of the proposed extensions can not be directly converted into the standard PN model. We need a more complex description model, powerful enough to represent data fields and guards; in addition it would facilitate the acceptance of our methodology by programmers familiar with PNs.
Once a distributed parallel program is described using XP_iN we may further convert it into a CPN, using `colored tokens` to interpret the behavior of the extended transitions.
Fig. 5 shows the XP_iN transition of fig. 1 represented as an equivalent CPN. The extended transition is split into one input transition (`T1'`) and one output transition (`T1''`). The sub-net between these two transitions represents the function to be executed.
The function returns a colored token `x`, to be tested in the output arcs of the `T1''` transition.
The place `Enbl` and the two connected arcs correspond to an upper-level self-loop, to implement in the lower level the behavior forced by the existence of the `ENABLE` field. This changes the expression associated with `T1'` to be (LLE + Enbl).GLE.

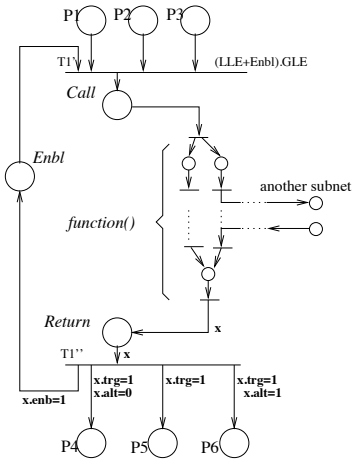

Figure 5: Convertion to CPN.

5. DISCUSSION

Programming with XP_iN does not require explicit thread creation or message passing, thus simplifying the complex management task required by parallel distributed applications. In addition, to achieve a high-level abstraction and modularity it supports the construction of composite programs.
By considering the design of a parallel application as being processed at two different levels of representation, will give the programmer greater flexibility and will make it easy to manage the overall design of a complex program to be executed in a distributed system.
The basic components (sub-nets) of a program are designed at the `microscopic level` of representation. At this level, we describe the functions to execute, using a conventional programming language such as `C`, and we specify their data dependencies, using XP_iN as the representation model.
Once the complete set of sub-nets required for the application are developed we may further continue the process design linking the components through a

definable explicit control structure, also specified by means of XP_iN.

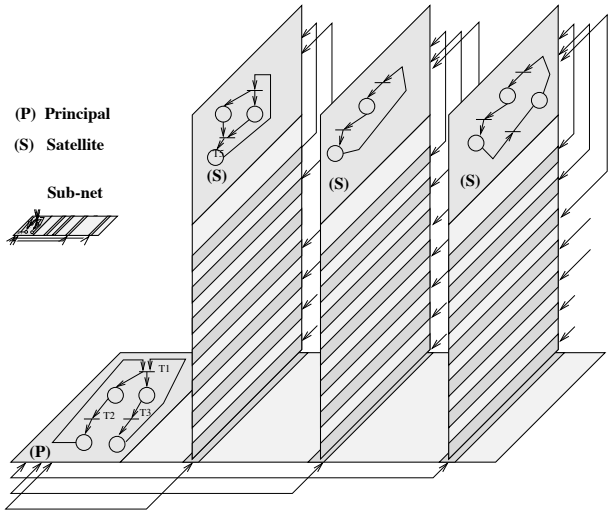

Figure 6: Interactive sub-nets.

To illustrate the advantages of the methodology, next we describe how it has been successfully applied in the implementation of MC^2.
The objective was to design and implement every `agent` (as defined in MC^2) viewed as a set of elementary agents or `cells` that may dynamically evolve, concurrently executing in a distributed system. In order to meet the required computing power and the modularity of the implementation we took the following approach:
At the microscopic level of representation the sub-net is equivalent to an `operon`[1] and the functions to execute when transitions are fired represent the elementary actions produced by `genes`.
At this level the `operon` is a basic functional unit and we use XP_iN as an elementary regulatory circuit that controls the firing or inhibiting of the genes.
At the macroscopic level each `operon` may be regarded both as a `functional unit` or a `regulatory unit` that cooperates to produce a common task; here we use XP_iN to integrate and govern the activity of the `operon`.
The complete functionality of a cell is represented by a `macro-net` that organizes and coordinates the totallity of the definable `sub-net` to produce a global behavior. Fig. 6 shows a representation of a cell as it has been modeled and implemented in XP_iN. To link the `sub-nets` that constitute a cell we use `Ports`, the low-level interaction mechanism included in XP_iN. Using the proposed extension we may establish a communication channel between the `regulatory units` and the `functional units` to obtain a higher level organization. Data and control tokens generated in both types of units may be sent and received through channels allowing any running transition to influence or be influenced by the actions being executed by other transitions in distinguished sub-nets.
A full representation of an agent is obtained by following the same approach we use to model a cell. The substitution is straightforward: at the microscopic level, instead of `genes` we take an `operon` and at the macroscopic level we substitute the `operon` by `cells`. The overall behavior of the `macro-net` that represents the agent is obtained again using the interaction mechanism provided by `ports`.

[1] **Operon**, *a group of neighbors genes in a chromosome that cooperate to accomplish a certain cellular function.*

Conclusions and future work

In this paper we presented XP_iN a model that benefits from PNs, being at the same time straightforward and comprehensive enough to represent distributed application programs.
XP_iN allow us to combine, in two different levels of representation, control with interaction and hierarchy to achieve a high-level abstraction and to model the complex relationships between all the components of a distributed system. XP_iN offers to the programmer a great flexibility and eases the management of the overall design of a parallel program.
Presently XP_iN is being implemented as a native software layer on top of DoTS [7], an experimental message-passing parallel programming environment for distributed computation that uses domains, groups and threads to obtain structured fine-grained computation and communication with control and shared memory.
We plan in the near future to extend the use of this layer to construct a programmable independent user-level package that may serve as an alternative thread scheduler to use under Unix based computational platforms.

6. REFERENCES

[1] A. Pina. MC^2 — *Origem e Evolução de um Modelo de Computação Celular*. PhD thesis, Dep. Informática, U. Minho, Braga, Portugal, 1997.

[2] T. Guan, T. Alukaidey, and S. Barros. An MPP reconfigurable architecture using free-space optical interconnects and Petri net configuring. *Journal of Systems Architecture*, 43(6&7):391–402, April 1997.

[3] T. S. Ray. An Evolutionary Approach to Synthetic Biology: Zen and the Art of Creating Life. *Artificial Life*, 1(1/2):195–226, 1996.

[4] M. D. Zisman. *Use of Production Systems for Modeling Asynchronous, Concurrent Processes*, pages 53–68. Academic Press, Inc., 1978.

[5] T. Murata. Circuit Theoretic Analysis and Synthesis of Marked Graphs. *IEEE Trans. on Circuits and Systems*, CAS-24(7):400–5, July 1977.

[6] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume I. Springer-Verlag, Berlin, Germany, 1992.

[7] A. Pina, C. Moreira, and V. Oliveira. DoTS: Domain and Shared Memory in a Message-passing Threads Environment. Tech. report, Dep. Informática, U. Minho, Braga, Portugal, May 1997.