Balancing Performance, Flexibility and Scalability in a Parallel Computing Platform for Membrane Computing Applications

Van Nguyen, David Kearney, Gianpaolo Gioiosa

School of Computer and Information Science
University of South Australia
{Van.Nguyen, David.Kearney, Gianpaolo.Gioiosa}@unisa.edu.au

Summary. Membrane computing investigates models of computation inspired by certain features of biological cells. To exploit the performance advantage of the large-scale parallelism of membrane computing models, it is necessary to execute them on a parallel computing platform. However, it is an open question whether it is feasible to develop a parallel computing platform for membrane computing applications that significantly outperforms equivalent sequential computing platforms while still achieving acceptable flexibility and scalability. To move closer to an answer to this question, we have investigated a novel approach to the development of a parallel computing platform for membrane computing applications that has the potential to deliver a good balance between performance, flexibility and scalability. This approach involves the use of reconfigurable hardware and an intelligent software component that is able to configure the hardware to suit the specific properties of the membrane computing model to be executed. We have developed a prototype computing platform called Reconfig-P based on the approach. Reconfig-P is the first computing platform of its type to implement parallelism at both the system and region levels. In this paper, we describe Reconfig-P and evaluate its performance, flexibility and scalability. Theoretical and empirical results suggest that the implementation approach on which Reconfig-P is based is a viable means of attaining a good balance between performance, flexibility and scalability in a parallel computing platform for membrane computing applications.

1 Introduction

Membrane computing investigates models of computation inspired by the structural and functional properties of biological cells. Such models have been applied in a variety of domains. To exploit the performance advantage of the large-scale parallelism of membrane computing models, it is necessary to execute them on a parallel computing platform. However, the use of a parallel computing platform instead of a sequential computing platform often comes at the cost of reduced flexibility and scalability.

The first parallel computing platforms for membrane computing applications to be published [15, 27, 30] do not exhibit sufficient flexibility or scalability. Even so, because research in this area is in its early stages, it is still an open question whether it is feasible to develop a parallel computing platform for membrane computing applications that significantly outperforms equivalent sequential computing platforms while still achieving acceptable flexibility and scalability. To move closer to an answer to this question, it is important to investigate the viability of implementation approaches that have the potential to deliver a good balance between performance, flexibility and scalability.

The research presented in this paper involves an investigation of a novel approach to the development of a parallel computing platform for membrane computing applications. This approach involves the use of reconfigurable hardware and an intelligent software component that is able to configure the hardware to suit the specific properties of the membrane computing model to be executed. We have developed a prototype computing platform called Reconfig-P based on the approach. In this paper, we describe Reconfig-P and evaluate its performance, flexibility and scalability.

The paper is organised as follows. In Section 2, we introduce key concepts and previous research associated with parallel computing platforms for membrane computing applications. In Section 3, we describe Reconfig-P. In Section 4, we evaluate the performance, flexibility and scalability of Reconfig-P. And in Section 5 we draw a conclusion regarding the viability of the implementation approach on which Reconfig-P is based.

2 Background

In this section, we introduce key concepts and previous research associated with parallel computing platforms for membrane computing applications. First, we introduce membrane computing and its applications. Second, we define the attributes of performance, flexibility and scalability in the context of a computing platform for membrane computing applications, explain the significance of these attributes, and indicate the connections that exist between them. Third, we describe the general characteristics of sequential computing platforms, software-based parallel computing platforms and hardware-based parallel computing platforms, and discuss the implications of these characteristics for performance, flexibility and scalability. Finally, we briefly describe existing computing platforms for membrane computing applications and evaluate their performance, flexibility and scalability.

2.1 Membrane computing and its applications

Membrane computing [24, 25] investigates models of computation inspired by certain structural and functional features of biological cells, especially features that arise because of the presence and activity of biological membranes.

Biological membranes define compartments inside a cell or separate a cell from its environment. The compartments of a cell contain chemical substances. The substances within a compartment may react with each other or be selectively transported through the membrane surrounding the compartment (e.g., through protein channels) to another compartment as part of the cell's operations.

In a membrane computing model, called a *P system*, multisets of objects (chemical substances) are placed in the regions defined by a hierarchical membrane structure, and the objects evolve by means of reaction rules (chemical reactions) also associated with the regions. The reaction rules are applied in a maximally parallel, nondeterministic manner. The objects can interact with other objects inside the same region or pass through the membrane surrounding the region to neighbouring regions or the cell's environment. These characteristics are used to define transitions between configurations of the system, and sequences of transitions are used to define computations. A computation halts when for every region it is not possible to apply any reaction rule. The input of the computation is defined by the multisets of objects in the initial configuration of the system. The output of the computation may be defined in various ways. For example, the output might be defined as the number of objects located in a particular region in the halting configuration of the system, or as the number of objects emitted to the system's environment during the course of the computation.

Following is a definition of an example P system model. All P systems Π that instantiate the model have the features specified in the definition. The model, which we call the *core P system model*, defines all the essential components of a P system plus two simple and commonly used additional features (namely, catalysts and reaction rule priorities).

$\Pi = (V, T, C, \mu, w_1, ..., w_m, (R_1, \rho_1), ..., (R_m, \rho_m))$, where

- V is an alphabet that contains labels for all the *types of objects* in the system;
- $T \subseteq V$ is the *output alphabet*, which contains labels for all the types of objects that are relevant to the determination of the system output;
- $C \subseteq V - T$ is the alphabet that contains labels for all the *types of catalysts*, which are the types of objects whose multiplicities cannot change through the application of a reaction rule;
- μ is a hierarchical membrane structure consisting of m membranes, with the membranes (and hence the regions defined by the membranes) injectively labelled by the elements of a given set H of m labels (in this paper, $H = \{1, 2, \ldots, m\}$);
- each $w_i, 1 \leq i \leq m$, is a string over V that represents the *multiset of objects* contained in region i of μ in the initial configuration of the system;
- each $R_i, 1 \leq i \leq m$, is a finite *set of reaction rules* over V associated with the region i of μ;
- a *reaction rule* is a pair (r, p), written in the form $r \rightarrow p$, where r is a string over V representing a multiset of reactant objects and p is a string over $\{a_{\text{here}}, a_{\text{out}}, a_{\text{in}} \mid a \in V\}$ representing a multiset of product objects, each

of which either (a) stays in the region to which the rule is associated (the subscript 'here' is usually omitted), (b) travels 'out' into the region that immediately contains the region to which the rule is associated, or (c) travels 'in' to one of the regions that is immediately contained by the region to which the rule is associated; and

- each ρ_i is a partial-order relation over R_i which defines the *relative priorities of the reaction rules* in R_i.

Several P system models have been developed that extend in various ways the core P system model. Examples of additional features found in these extended models include: structured (i.e., non-atomic) objects, membrane creation and dissolution, special inter-region communication rules (e.g., symport and antiport rules), membrane permeability, and electronic charge for objects and membranes. See [24] for a discussion of these features.

P system models have been applied in a variety of domains, including algorithm solving and analysis [1, 19, 20, 23], linguistics [6] and biology [2, 8, 10, 11, 12, 22, 29]. Most existing applications of membrane computing are targeted at the modelling and simulation of biological systems. For example, researchers have modelled the following biological systems as P systems: respiration [10], photosynthesis [22], cell-mediated immunity [12], mechanosensitive channels [2] and protein signalling pathways [29]. In general, biologists are interested in using P systems to perform simulations, rather than to produce computational outputs as in classical computing. That is, they are interested in viewing the configuration-by-configuration evolution of a P system and not only in the final output produced by the P system. Once a biological system has been modelled as a P system, it is possible to simulate the biological system by executing the P system. In many cases, the P system will be executed many times so that the effect of different initial conditions on the evolution of the biological system can be studied.

2.2 Quality attributes of computing platforms for membrane computing applications

The overall quality of a computing platform depends on the extent to which it possesses certain positive attributes, including usability, performance, flexibility, maintainability and scalability. Performance, flexibility and scalability are three of the most important quality attributes for a computing platform for membrane computing applications. Ensuring that a computing platform for membrane computing applications has all three of these attributes to an acceptable degree is a challenge, because a factor that promotes one of the attributes can sometimes demote another one of the attributes. In this section, we define the attributes of performance, flexibility and scalability in the context of a computing platform for membrane computing applications, explain the significance of these attributes, and indicate the connections that exist between them.

Performance

By the performance of a computing platform for membrane computing applications we mean the speed at which it executes P systems; that is, the amount of useful processing it performs per unit time. A suitable measure of the amount of useful processing performed is the number of reaction rule applications performed. Thus the performance of a computing platform for membrane computing applications can be measured in reaction rule applications per unit time.

Flexibility

By the flexibility of a computing platform for membrane computing applications we mean the extent to which it can support the execution of a wide range of P systems. Thus a flexible computing platform for membrane computing applications must be able to adapt to the specific properties of the P system to be executed. The greater the flexibility of the computing platform, the greater the diversity among the P systems in the class of P systems that the computing platform is able to execute.

Scalability

By the scalability of a computing platform for membrane computing applications we mean the extent to which increases in the size of the P system to be executed do not lead to a reduction in the ability of the computing platform to perform its functions or a reduction in the performance of the computing platform. We take the size of a P system as being largely determined by the number of regions and the number of reaction rules it contains.

Connections between performance, flexibility and scalability

The performance of a computing platform for membrane computing applications can be increased by tailoring its implementation to the specific properties of the P systems it is intended to execute. However, the greater the diversity of these P systems, the more difficult it is to efficiently tailor the implementation to their specific properties. Therefore, increasing the performance of the computing platform is likely to come at the cost of reduced flexibility, while increasing the flexibility of the computing platform is likely to come at the cost of reduced performance.

Increasing the flexibility of a computing platform for membrane computing applications involves supporting additional P system features. Naturally, this usually requires the implementation of additional data structures and algorithms. In software-based computing platforms, the implementation of additional data structures is likely to come at the cost of increased memory consumption. In hardware-based computing platforms, the implementation of additional data structures comes at the cost of increased hardware resource consumption, as does the implementation of additional algorithms. Therefore, given that memory resources and hardware resources are limited, implementing additional P system features reduces the maximum size of the P systems that a computing platform for membrane computing applications can execute. Thus increasing the flexibility of a computing platform for membrane computing applications is likely to come at the cost of reduced scalability, while increasing the scalability of such a computing platform is likely to come at the cost of reduced flexibility.

2.3 Types of computing platforms

We identify three major types of computing platforms: sequential computing platforms, software-based parallel computing platforms and hardware-based parallel computing platforms.

Sequential computing platforms are typically based on a software-programmed microprocessor. When such a microprocessor is used, the execution hardware is abstracted by the instruction set architecture, which provides a set of specific instructions that the microprocessor can process to perform computations. This is a very flexible computing solution since it is possible to change the functionality of the computing platform simply by modifying its software — there is no need to modify the hardware configuration. As a result of this flexibility, the same fixed hardware can be used for many applications. However, the flexibility comes at the cost of lower performance. As each instruction needs to be sequentially fetched from memory and decoded before being executed, there is a high execution overhead associated with each individual operation. Furthermore, only one instruction can be executed at a time.

Software-based parallel computing platforms are typically based on a cluster of software-programmed microprocessors. Because the microprocessors execute in parallel, software-based parallel computing platforms can significantly outperform sequential computing platforms for many applications. The microprocessors synchronise their activities by using shared memory or by sending messages to each other (often over a network). Such synchronisation can be very time consuming, and therefore can hinder performance significantly. Increasing the performance of a software-based parallel computing platform involves increasing the amount of parallelism and therefore requires the inclusion of additional microprocessors. However, as the number of microprocessors increases, the overheads associated with synchronisation increase substantially (unless the overall algorithm executed by the computing platform can be neatly partitioned into separate procedures that are largely independent of each other). This fact limits the scalability of software-based parallel computing platforms.

Hardware-based parallel computing platforms execute algorithms that have been directly implemented in hardware. In one approach, an application-specific integrated circuit (ASIC) is used. The design of an ASIC is tailored to a specific algorithm. As a consequence, ASICs usually achieve a higher performance than software-programmed microprocessors when executing the algorithm for which they were designed. However, with this higher performance comes reduced flexibility: as the implemented algorithm is fabricated on a silicon chip, it cannot be altered without creating another chip. In another approach, reconfigurable hardware is used. Unlike ASICs, reconfigurable hardware can be modified. Therefore, by using reconfigurable hardware, it is possible to improve on the performance of software-based computing platforms while retaining some of their flexibility. A field-programmable gate array (FPGA) is a type of reconfigurable hardware device. As shown in Figure 1, an FPGA consists of a matrix of logic blocks which are connected by means of a network of wires. The logic blocks at the periphery of the matrix can perform I/O operations. The functionality of the logic blocks and the connections between them can be modified by loading configuration data from a host computer. In this way, any custom digital circuit can be mapped onto the FPGA, thereby enabling it to execute a variety of applications. The digital circuits used in hardware-based parallel computing platforms are specified in hardware description languages. A very popular hardware description language is VHDL. VHDL allows circuits to be specified either in terms of a structural description of the circuit or in terms of low-level algorithmic behaviours of the circuit. Another popular hardware description language is Handel-C. Unlike VHDL, Handel-C does not support the specification of the structural features of a hardware circuit. However, sharing a syntax similar to that of the C programming language, Handel-C allows algorithms to be specified at a very abstract level, and therefore eases the process of designing a circuit for an application.

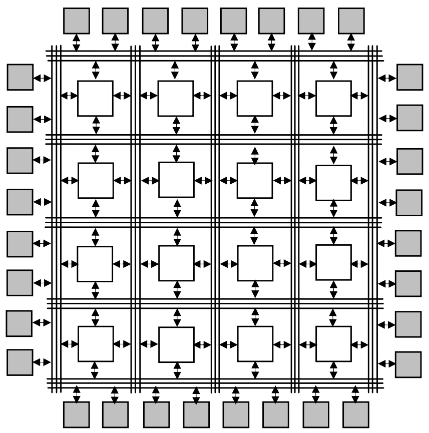


Fig. 1. The basic architecture of an FPGA.

2.4 Existing computing platforms for membrane computing applications

In this section, we provide a brief survey of existing computing platforms for membrane computing applications.

Sequential computing platforms

As P systems are inherently parallel devices, it is not possible to truly implement them on sequential computing platforms. Nevertheless, sequential computing platforms exist that enable one to simulate in a sequential manner the execution of P systems [4, 7, 9, 13, 16, 21, 26, 28]. For example, Nepomuceno-Chamarro's software tool SimCM [21] is able to simulate the execution of P systems that have the features specified in the definition of the core P system model as well as the feature of membrane dissolution. SimCM is written in Java. It provides a graphical user interface that enables the user to specify and view the evolution of a P system in a visual manner.

Software-based parallel computing platforms

Two research groups have created prototypes of software-based parallel computing platforms for membrane computing applications. Ciobanu and Guo [15] have implemented a simulation of P systems on a Linux cluster using C++ and a library of functions for message-passing parallel computation called the Message Passing Interface (MPI), while Syropoulos and colleagues [30] have implemented a distributed simulation of P systems using Java Remote Method Invocation (RMI). We discuss Ciobanu and Guo's computing platform below.

Ciobanu and Guo's computing platform

Ciobanu and Guo's computing platform is a software program written in C++ that is designed to run on a cluster of computers. The communication mechanism for the computing platform is implemented using MPI. In its prototype form, the computing platform consists of a Linux cluster, in which each node has two 1.4GHz Intel Pentium III CPUs and 1GB of memory, and the nodes are connected by gigabit Ethernet.

Ciobanu and Guo's computing platform supports the execution of a class of P systems that is very similar to the class of P systems that instantiate the core P system model. That is, the computing platform implements most of the basic features of P systems, but does not implement additional features such as membrane creation and dissolution. In the computing platform, each region of a P system is modelled as a separate computational process. Such a process implements the application of the reaction rules in its corresponding region. The processes for the regions in the P system execute in parallel. Communication and synchronisation between regions is implemented using MPI. The application of reaction rules is performed in rounds. At the end of each round, each region exchanges messages with its parent region and child regions. The reaction rules associated with a region are implemented as threads. If multiple reaction rules require objects of the same type in a round, then only one of them is allowed to consume objects of that type in that round. If two reaction rules do not have relative priorities, which of the two reaction rules is allowed to consume objects first is determined at random.

The process associated with the outermost region of the P system executes a halting detection algorithm. If it detects that the P system has halted, it broadcasts this information to the processes associated with the other regions in the P system.

As the threads for the reaction rules in a region execute on the same node in the cluster, and there are only two processors per node, it would seem that it is impossible for the computing platform to achieve region-level parallelism for anything other than small P systems. To achieve region-level parallelism for larger P systems, it would be necessary to increase the number of processors in a node from two to a number at least equal to the number of reaction rules in the region corresponding to that node. Thus without the inclusion of additional nodes, the computing platform cannot be said to implement region-level parallelism, although it does implement system-level parallelism. Nevertheless, Ciobanu and Guo's use of multithreading as a means of representing the concurrent application of reaction rules within a region is promising, and could be used to implement region-level parallelism if sufficient hardware resources were available.

Ciobanu and Guo do not provide a detailed evaluation of the performance of their computing platform. However, they do report that the performance of the computing platform is somewhat unpredictable. While the execution times exhibited by the computing platform are often acceptable, some execution times are unacceptably long owing to unexpected network congestion. Ciobanu and Guo indicate that the major problem with their computing platform from the point of view of performance is the overhead associated with communication and cooperation between regions. Such communication and cooperation consumes most of the total execution time.

Ciobanu and Guo do not evaluate the scalability of their computing platform. However, it is clear that the scalability of the computing platform is limited to a large extent by the nature of a cluster-based implementation approach. For example, to execute P systems with a large number of regions, the computing platform would have to include a large number of nodes, since there is a one-to-one correspondence between regions and nodes. As a consequence, there would be very significant overheads associated with communication and synchronisation between regions, and this would have an adverse impact on the performance of the computing platform.

As it implements only a basic P system model, Ciobanu and Guo's computing platform is not capable of executing P systems that have additional features such as symport and antiport rules. This detracts from its flexibility. Nevertheless, since the existing implementation is expressed at a level of abstraction at which the high-level features of a P system are apparent, it seems very feasible that the computing platform could be extended to support additional P system features.

Hardware-based parallel computing platforms

A few researchers have designed digital circuits for particular aspects of P systems (e.g., see [17, 18]). However, to the best of our knowledge, only Petreska and Teuscher [27] have implemented a hardware-based computing platform for membrane computing applications. We discuss Petreska and Teuscher's computing platform below.

Petreska and Teuscher's computing platform

Petreska and Teuscher [27] have developed a full implementation of a particular P system model on reconfigurable hardware. This P system model is similar to the core P system model, except that it also includes the feature of membrane creation and dissolution. The hardware architecture for the specific P system to be executed, which is specified in structural VHDL, is elegant in that it contains only one type of high-level hardware component (a universal component) and interconnections between components of this type.

Petreska and Teuscher have demonstrated the feasibility of implementing some of the important features of membrane computing on reconfigurable hardware. Nevertheless, their computing platform has four main limitations.

First, the computing platform does not exploit the performance advantages of the membrane computing paradigm. This is primarily because it does not implement parallelism at the region level (i.e., the reaction rules in a region are applied sequentially). Achieving region-level parallelism requires the implementation of a scheme for the resolution of conflicts that arise when different reaction rules compete for or produce the same types of objects in the same region at the same time. It is difficult to implement such a scheme efficiently in hardware, especially when a low-level hardware description language is used, and this is perhaps a major reason why Petreska and Teuscher did not attempt to do so. Conflicts do not arise when reaction rules do not compete for or produce the same types of objects in the same region. Nevertheless, in Petreska and Teuscher's computing platform, even such non-conflicting reaction rules must be applied sequentially. Furthermore, if reaction rules from different regions need to update the same multiset, their respective update operations must occur sequentially.

Second, the computing platform is inflexible. As the computing platform uses only one type of high-level hardware component and connects components of this type to build hardware architectures in a fixed manner, the extent to which the hardware architecture for a P system can be tailored to the specific characteristics of the P system is limited.

Third, the computing platform is not extensible. As it is specified at the hardware level in a low-level hardware description language, adding support for additional P system features would require redesigning the hardware for the computing platform directly. This is likely in most cases to be a difficult and time-consuming task, given the dependence of the computing platform on the design of a single universal hardware component. Thus there is limited opportunity to improve the flexibility of the computing platform.

Fourth, the computing platform has limited scalabilty. As there is only a limited ability to tailor the hardware architecture to the specific characteristics of the P system to be executed, the hardware architecture often includes many redundant hardware components. These redundant components unnecessarily consume hardware resources.

As it implements membrane creation and dissolution in addition to the basic P system features included in the core P system model, Petreska and Teuscher's computing platform can execute a wider range of P systems than Ciobanu and Guo's computing platform. So, in this respect, it is more flexible than Ciobanu and Guo's computing platform. However, being specified in a low-level hardware description language, the implementation of Petreska and Teuscher's computing platform is more brittle, and therefore less extensible, than the implementation of Ciobanu and Guo's computing platform. Therefore, unlike in the case of Ciobanu and Guo's computing platform, it seems that it would be very difficult to increase the flexibility of Petreska and Teuscher's computing platform without significantly changing its existing implementation.

3 Description of Reconfig-P

In this section, we describe Reconfig-P, our prototype hardware-based parallel computing platform for membrane computing applications. Being the first computing platform based on reconfigurable hardware to implement parallelism at both the system and region levels, Reconfig-P advances the state-of-the-art in hardware implementations of membrane computing. First, we specify the key features of the novel implementation approach on which Reconfig-P is based, and explain why this implementation approach has the potential to deliver a good balance between performance, flexibility and scalability. Second, we specify the functional requirements of Reconfig-P. Third, we provide an overview of the major components of Reconfig-P and the role of these components in the execution of membrane computing applications. Fourth, we provide an overview of the functionality of P Builder, a software component of Reconfig-P that is responsible for generating customised hardware representations for P systems. Finally, we describe how P Builder represents the fundamental structural and behavioural features of P systems in hardware.

3.1 Implementation approach

Key features of the implementation approach

The implementation approach on which Reconfig-P is based involves

- use of a reconfigurable hardware platform,
- generation of a customised digital circuit for each P system to be executed, and
- use of a hardware description language that allows digital circuits to be specified at a level of abstraction similar to the level of abstraction at which a general-purpose procedural software programming language (such as C) allows algorithms to be specified.

In the approach, a software component of the computing platform is responsible for analysing the structural and behavioural features of the P system to be executed and producing a hardware description for the P system that is tailored to these features. When determining the hardware description for the P system, the software component aims to maximise performance and minimise hardware resource consumption.

Potential of the implementation approach

The use of reconfigurable hardware opens up the possibility of generating custom digital circuits for P systems. The ability to generate a custom circuit for the P system to be executed makes it possible to design this circuit according to the specific structural and behavioural features of the P system, and therefore facilitates the design of circuits that exhibit good performance and economical hardware usage. Therefore the implementation approach facilitates the development of a computing platform that exhibits good performance and economical hardware usage. For example, because the number of reaction rules in the P system to be executed is known before it is executed, the circuit for the P system can be designed in such a way that it includes exactly that number of processing units to implement the reaction rules. Without the possibility of generating a custom circuit, the circuit for the P system would have to include a fixed number of processing units for reaction rules, and therefore would often include redundant hardware components. Also, because it is possible by inspection of the definitions of the reaction rules in a P system to determine for any two regions in the P system whether it is possible for objects to traverse between these regions, the circuit for a P system can be designed in such a way that the logic that implements object traversal is included only for those inter-region connections over which object traversal is possible.

The fact that digital circuits are specified at a level of abstraction similar to that at which a general-purpose procedural software programming language specifies algorithms, rather than at a level of abstraction that reveals the structure or low-level algorithmic behaviour of the circuits, makes it more feasible to develop a software component that is able to flexibly adapt to the specific features of the P system to be executed when generating a circuit for that P system. The greater the ability of the software component to flexibly adapt to the specific features of the P system to be executed, the greater the range of P systems for which it is capable of generating circuits that exhibit good performance and economical hardware usage.

Therefore the implementation approach facilitates the development of a computing platform that exhibits good flexibility. For example, as mentioned in Section 2.4, implementing parallelism at the region level of a P system requires resolving conflicts that may occur when different reaction rules update the same multiplicity values. If a low-level hardware description language were used, it would be very difficult to resolve such conflicts in an efficient manner. The use of a high-level hardware description language makes it more feasible that a solution to the conflict resolution problem can be found.

Because it involves the use of a hardware description language that is incapable of expressing the low-level structure and behaviour of digital circuits, the implementation approach limits the extent to which low-level optimisations of circuits can be carried out. However, it is unlikely that the benefits of customisation and flexibility mentioned above could be achieved if a low-level hardware description language were used.

The above considerations suggest that the implementation approach has the potential to deliver a good balance between performance, flexibility and scalability in a parallel computing platform for membrane computing applications.

3.2 Functional requirements

Reconfig-P is required to execute P systems that instantiate the core P system model on reconfigurable hardware. In addition, to facilitate testing of P system designs, Reconfig-P is required to enable the user to execute a P system in software, and view the configuration-by-configuration evolution of the P system, before generating a hardware circuit for the P system.

It is not a strict requirement that Reconfig-P implement the nondeterminism of P systems. In particular, Reconfig-P is not required to implement the assignment of objects to reaction rules in a nondeterministic manner. Although nondeterminism is very important from a theoretical perspective and can be useful in some applications, many applications of membrane computing do not depend on the nondeterministic aspects of P systems. Therefore we do not regard it as crucial that a computing platform for membrane computing applications implement the nondeterminism of P systems. Even so, it is certainly desirable that a computing platform for membrane computing applications implement the nondeterminism of P systems. We intend to investigate the feasibility of implementing the nondeterminism of P systems in a future version of Reconfig-P.

3.3 System overview

Figure 2 shows the major components of Reconfig-P and the roles of these components in the execution of a P system.

(1) The user begins using Reconfig-P by writing a *P system specification.* This specification defines a P system that is described in terms of the *core P system*

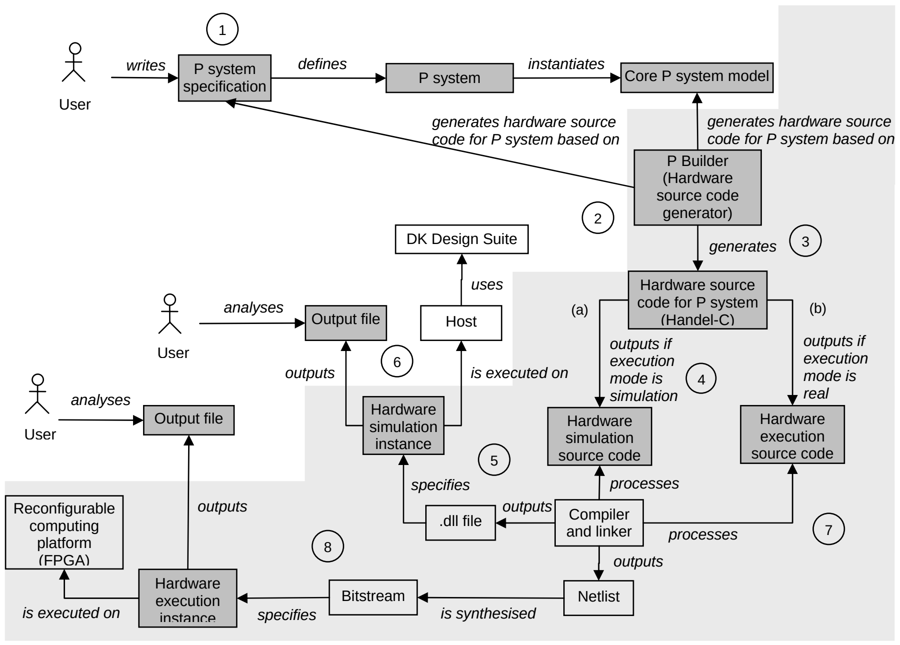

1
User
writes
P system specification
defines
P system
instantiates
Core P system model
generates hardware source code for P system based on
generates hardware source code for P system based on
P Builder (Hardware source code generator)
2
generates
3
DK Design Suite
uses
Hardware source code for P system (Handel-C)
(a)
(b)
User
analyses
Output file
Host
outputs if execution mode is simulation
outputs if execution mode is real
outputs
6
is executed on
4
User
analyses
Output file
Hardware simulation instance
Hardware simulation source code
Hardware execution source code
5
specifies
processes
Reconfigurable computing platform (FPGA)
outputs
outputs
.dll file
Compiler and linker
processes
7
8
outputs
Hardware execution instance
Bitstream
Netlist
is executed on
specifies
is synthesised

Fig. 2. An overview of Reconfig-P. The shaded region covers the components of Reconfig-P that are transparent to the user.

model. (2) The *hardware source code generator* called *P Builder* (which is hidden from the user) processes the input information. (3) P Builder analyses the P system specification, determines a customised hardware representation for the P system, and then generates Handel-C source code that implements the hardware representation. (4) The user can choose to (a) execute the source code in hardware, or (b) simulate the execution of the source code in software. (5) The ability to generate *simulation source code* enables users to examine their P system design before building a corresponding hardware circuit. (6) The *simulation instance* (specified by a DLL file) is executed on a *host* computer. The host computer invokes the simulation feature provided by the Celoxica *DK Design Suite* to allow users to (a) view the evolution of their P system one configuration at a time, or (b) return the output of the simulation in an *output file.* (7) The generation of *hardware execution source code* allows the user, once they have finalised the design of their P system, to build a hardware circuit for the P system. (8) The hardware execution source code is then synthesised into a hardware circuit. A *hardware execution instance* (specified by a bitstream) can then be executed on a *reconfigurable hardware platform* (an FPGA). The FPGA communicates with the host computer via a PCI bus. The output of the execution instance is stored in an *output file*, which can then be analysed by the user.

Much of the process of executing a P system is transparent to the user. The shaded region in Figure 2 covers the components of Reconfig-P that are transparent to the user.

3.4 P Builder

P Builder is responsible for implementing the hardware reconfiguration capability of Reconfig-P. It generates customised Handel-C source code for a P system based on the specific characteristics of the P system.

P Builder interprets a simple declarative language which is used by the user to specify P systems. More specifically, P Builder supports the execution of a P system by

1. converting a text representation of the P system (the P system specification) into software objects (written in Java);
2. converting the object representation into an abstract hardware representation and then into Handel-C source code that implements the abstract hardware representation; and then
3. converting the Handel-C source code that implements the abstract hardware representation into a hardware circuit (by invoking Xilinx tools) or initiating a simulation of the abstract hardware representation in software (by invoking the DK Design Suite).

Since P Builder is hidden from the user, the mechanics of the conversion process it performs are transparent to the user. The conversion process is illustrated in Figure 3. The innovation of P Builder lies in the way it converts a P system specification into an abstract hardware representation. In the next section, we describe how P Builder represents the core structural and behavioural features of P systems in hardware.

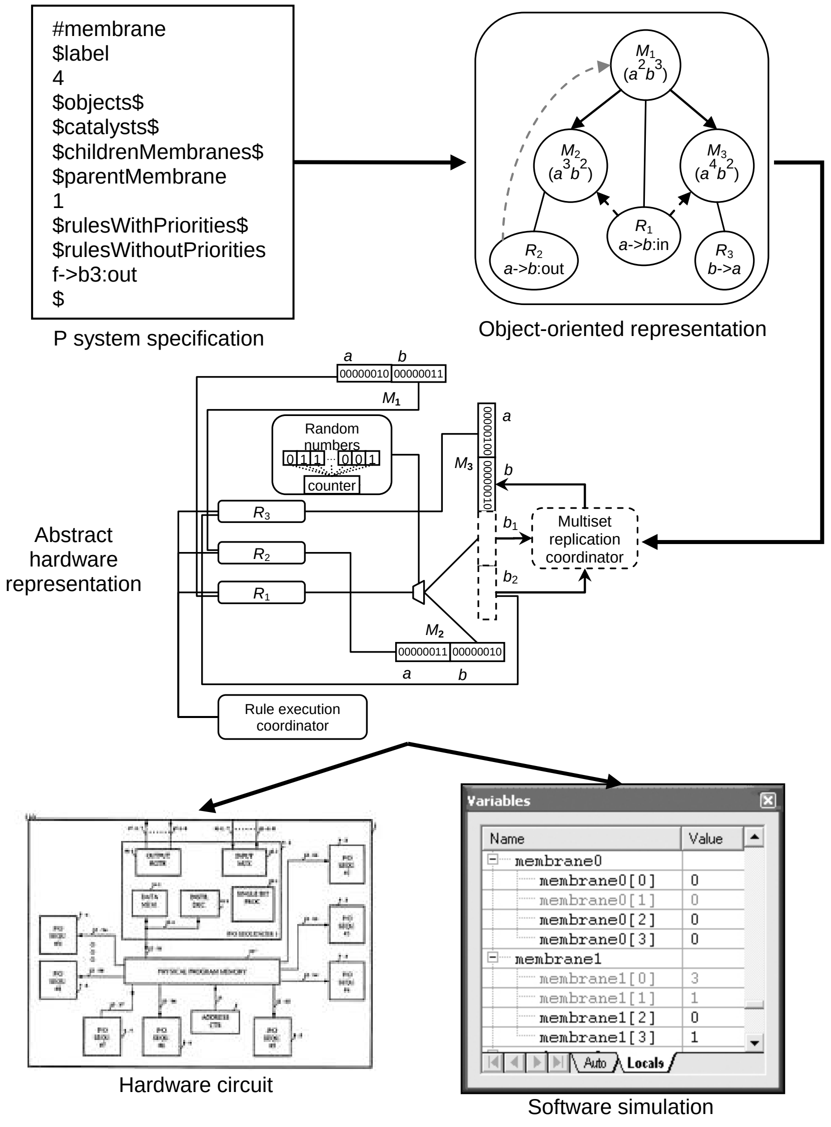
3.5 Hardware implementation of core P system features

P systems can differ significantly with respect to size, structure and information content. Reconfig-P takes advantage of this fact by configuring the hardware according to the specific requirements of the P system to be executed.

Although P systems can differ significantly, there are certain core features common to all P systems. These include (a) regions and their containment relationships, (b) the mutiset of objects in every region, (c) application of reaction rules, and (d) synchronisation of the application of reaction rules. This section describes how these core features are implemented in hardware.

Regions and their containment relationships

As the evolution of a P system is essentially a matter of the modification of the contents of regions according to certain rules, regions do not need to be explicitly



```
#membrane
$label
4
$objects$
$catalysts$
$childrenMembranes$
$parentMembrane
1
$rulesWithPriorities$
$rulesWithoutPriorities
f->b3:out
$
```

Variables

Name	Value
membrane0	
membrane0[0]	0
membrane0[1]	0
membrane0[2]	0
membrane0[3]	0
membrane1	
membrane1[0]	3
membrane1[1]	1
membrane1[2]	0
membrane1[3]	1

Auto Locals

Fig. 3. P Builder converts the text specification of a P system into an executable hardware circuit or a software simulation of such a hardware circuit. In the intermediate stages of the conversion process, the P system is represented as software objects and then as abstract hardware components.

represented in hardware. Instead, a region is represented in hardware implicitly via its contents. The only inter-region containment relationships that it is important to represent are those between regions between which it is possible for objects to traverse through the application of a reaction rule. These containment relationships are represented implicitly by ensuring that each reaction rule with an ‘in’ or ‘out’ target directive has, for each region to/from which it sends/receives objects, access to the multiset of objects in that region.

Multisets of objects in regions

Because the multiplicity values of objects in a region can be accessed by multiple reaction rules simultaneously, the hardware elements that store them should support concurrent accesses. Therefore a multiset is implemented as an array of registers (see Figure 4). Because it is infeasible to predict which types of objects may become available in which regions during the evolution of a P system, the array of registers that represents the multiset of objects in a region contains one register for every type of object in the alphabet of the P system. A common bitwidth is used for all object types (the default width is 8 bits).

Using registers can be expensive if a large amount of data needs to be stored. However, because in the hardware design each register corresponds to the multiplicity of a *type* of object in a region (rather than an individual object), for most P systems only a relatively small amount of data needs to be stored.

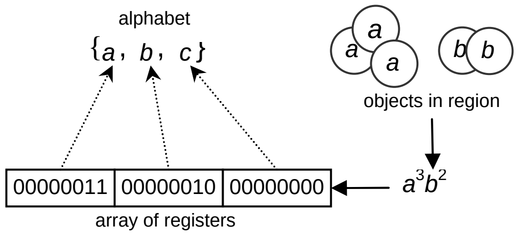


Fig. 4. A multiset of objects in a region is implemented as an array of registers.

Reaction rules

A reaction rule is implemented as a processing unit. This processing unit is represented in Handel-C as a potentially infinite while loop that contains code that specifies the processing associated with the application of the reaction rule. If a reaction rule operates on the multiplicity value for a particular object type in a particular region, then the section of the code for its corresponding processing unit that accomplishes this operation contains a reference to the array element representing that multiplicity value.

In a transition of a P system, all the reaction rules in the system complete one instance of execution, which consists of two phases. In the first phase, called the *preparation phase*, objects are assigned to the reaction rules that require them as reactants or catalysts. That is, for each reaction rule in each region, the number of instances of the reaction rule that can be applied is determined. In the second phase, called the *updating phase*, each applicable reaction rule updates one or more multisets of objects according to its definition and the number of instances of the reaction rule that can be applied.

The rest of this section describes the processing performed by the processing units for reaction rules during the preparation and updating phases.

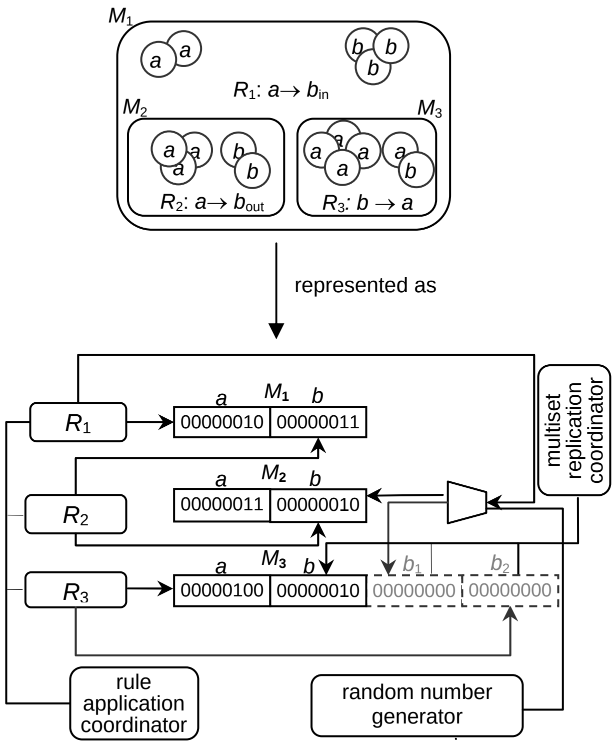


Fig. 5. An illustration of how the structural aspects of a P system are represented as high-level hardware components. (The multiset replication coordinator and extra array elements for object type b in region 3 are included only if the space-oriented conflict resolution strategy is used.)

Preparation phase

In the preparation phase, each reaction rule attempts to obtain as many of each of its required types of object as possible so as to maximise the number of instances of the reaction rule that can be applied in the updating phase. Therefore implementing the preparation phase involves calculating for each reaction rule r the value max-instances$_r$, which is the maximum number of instances of r that can be applied in the current transition of the P system given (a) the current state of the multiset of objects in its region and (b) the relative priorities and requirements of the other reaction rules in its region. The processing unit corresponding to r performs the calculation.

To calculate max-instances$_r$, the processing unit for a reaction rule r first calculates for each of its required object types (using integer division) the ratio of the number of available objects of that type in the region of r to the number of objects of that type needed to apply one instance of r. This is done in one clock cycle. It then calculates max-instances$_r$, which is equal to the minimum ratio calculated in the previous step. The operation of determining the minimum ratio can be represented as a binary tree in which each node corresponds to the execution of a binary MIN operation and executing the MIN operation at the root node gives the value of the minimum ratio. This tree has $\log_2 n$ levels, where n is the sum of the number of reactants and the number of catalysts in the definition of r. The processing unit for r evaluates max-instances$_r$ by first executing in parallel all the MIN operations at the bottom (leaf) level of the tree, then executing in parallel all the MIN operations at the next level up, and so on, until finally it executes the MIN operation at the root node to obtain the value of max-instances$_r$. Therefore calculating max-instances$_r$ takes $\log_2 n$ clock cycles.

If two reaction rules attempt to obtain objects of the same type, then their corresponding processing units execute the relevant operation one after the other according to their relative priorities. (It is assumed that reaction rules that attempt to obtain objects of the same type have been assigned relative priorities.) Otherwise, the processing units for different reaction rules execute in parallel. Therefore the number of clock cycles taken to complete the preparation phase for the entire P system is the maximum number of clock cycles taken by an individual reaction rule, out of all the reaction rules in the P system, to complete its preparation phase.

Updating phase

At the start of the updating phase, the processing unit for a reaction rule r inspects the value of max-instances$_r$ to determine whether r is applicable in the current transition. If max-instances$_r = 0$, r is inapplicable; otherwise r is applicable. As it takes zero clock cycles to evaluate a conditional expression in Handel-C, determining the applicability of r takes zero clock cycles. The applicability status of r is recorded in the isApplicableFlag of r (see Figure 6). Once the applicability status of each reaction rule has been determined and recorded, the processing unit that coordinates the execution of reaction rules is able to determine whether the P system should halt or continue the updating phase. Assume that the P system should continue the updating phase. If r is inapplicable, the processing unit for r simply waits for the next transition. If r is applicable, it moves on to the next step of the updating phase.

In the next step of the updating phase, every instance of every applicable reaction rule is applied. This is implemented by having the processing unit for each applicable reaction rule r bring about the combined effect of the execution of the instances of r. That is, the processing unit decreases/increases certain multiplicity values in certain multiset data structures according to the type, amount and source/destination of the objects consumed/produced by the instances of the reaction rule. For example, in Figure 5, in the next transition of the P system represented at the top of the figure, the processing unit R_2 would decrease by 2 the value stored in the register corresponding to object type a in the multiset data structure for region 2, and increase by 2 the value stored in the register corresponding to object type b in the multiset data structure for region 1.

If a reaction rule includes 'in' target directives, the definition of a P system calls for nondeterministic targeting of objects if there are multiple child regions. Such nondeterministic targeting can be approximated through the use of pseudorandom numbers. Therefore, the hardware design associates a *random number generator* to each processing unit for a reaction rule that might produce objects in multiple child regions of its own region. When such a processing unit needs to select a destination child region, it invokes its random number generator to obtain a number which identifies the child region to be selected. For example, the processing unit R_1 in Figure 5 invokes its random number generator to determine whether to produce b objects in region 2 or in region 3.

Processing units for reaction rules that do not manipulate any multiplicity values in common execute in parallel during the updating phase. This is not necessarily the case for processing units for reaction rules that do manipulate at least one multiplicity value in common, since without further measures being taken, the parallel execution of such processing units would lead to situations where multiple processing units write to the same register at the same time. Section 3.6 describes two alternative techniques Reconfig-P makes available for the prevention of such situations, and shows the extent to which each technique allows conflicting processing units to execute in parallel during the updating phase. The number of clock cycles taken to complete the updating phase depends on the conflict resolution strategy that is adopted.

Synchronisation of reaction rules

Figure 6 illustrates the synchronisation of reaction rules involved in the execution of a transition of a P system.

The synchronisation of reaction rules is controlled by three sentinels — preparationSentinel, applicableSentinel and updatingSentinel — and corresponding flags associated with each reaction rule — preparationCompleteFlag, updatingCompleteFlag and isApplicableFlag. The sentinels are implemented as 1-bit registers. Each type of flag is implemented as an array of 1-bit registers, each element of which being associated with one reaction rule in the P system. The flags preparationCompleteFlag, isApplicableFlag and updatingCompleteFlag for a reaction rule are used to indicate whether the reaction rule has completed its preparation phase, is applicable, and has completed its updating phase, respectively. The value of each sentinel is the result of performing the AND or OR function to the values of all its corresponding flags. The value of preparationSentinel indicates whether all reaction rules in the P system have completed their preparation phase. The value of applicableSentinel indicates whether at least one reaction rule is applicable (i.e., whether the P system should continue execution). And the value of updatingSentinel indicates whether all applicable reaction rules in the P system have completed their updating phase (and hence whether the P system is ready to proceed to the next transition).

The management of synchronisation is the responsibility of the *rule application coordinator*, a processing unit that executes in parallel with the processing units for the reaction rules (see Figure 5). The rule application coordinator monitors the conditions relevant to synchronisation at each clock cycle.

Let n be the number of reaction rules in the P system. The most natural way to implement the updating of a sentinel value in Handel-C is to write an assignment statement of the form $s = f_1 \# f_2 \# ... \# f_n$, where s is the variable that stores the sentinel value, each $f_i (1 \leq i \leq n)$ is a flag for a reaction rule, and # is either the AND operator or the OR operator. Clearly, the greater the number of reaction rules in the P system, the greater the number of AND or OR operations that need to be performed, and therefore the greater the depth of the logic that implements the assignment statement. Since in Handel-C an assignment statement always takes one clock cycle to execute, increasing the depth of the logic that implements an assignment statement can result in a reduction in the clock rate of the FPGA. Therefore, in some situations, decomposing an assignment statement into multiple assignment statements of reduced logical depth can prevent or mitigate a reduction in the clock rate of the FPGA. Whether or not performing such a decomposition is advantageous depends on whether the beneficial effect of reducing the clock cycle length outweighs the detrimental effect of introducing extra clock cycles. For P systems with a large number of reaction rules it might be advantageous to decompose each assignment statement that implements the updating of a sentinel value into multiple assignment statements of reduced logical depth. Therefore Reconfig-P incorporates a *logic depth reduction* feature. It decomposes an assignment statement with n operands into multiple assignment statements, each of with has at most $x \leq n$ operands. If as many of these assignment statements as possible contain x operands, then the original assignment statement is replaced by $\lceil \log_x n \rceil$ assignment statements. The user sets the value of x in order to obtain the best results. By default, Reconfig-P does not perform logic depth reduction (i.e., $x = n$ by default).

3.6 Conflict resolution in the updating phase

As discussed in Section 3.5, a conflict occurs in the updating phase when multiple processing units for reaction rules write to the same register at the same time.

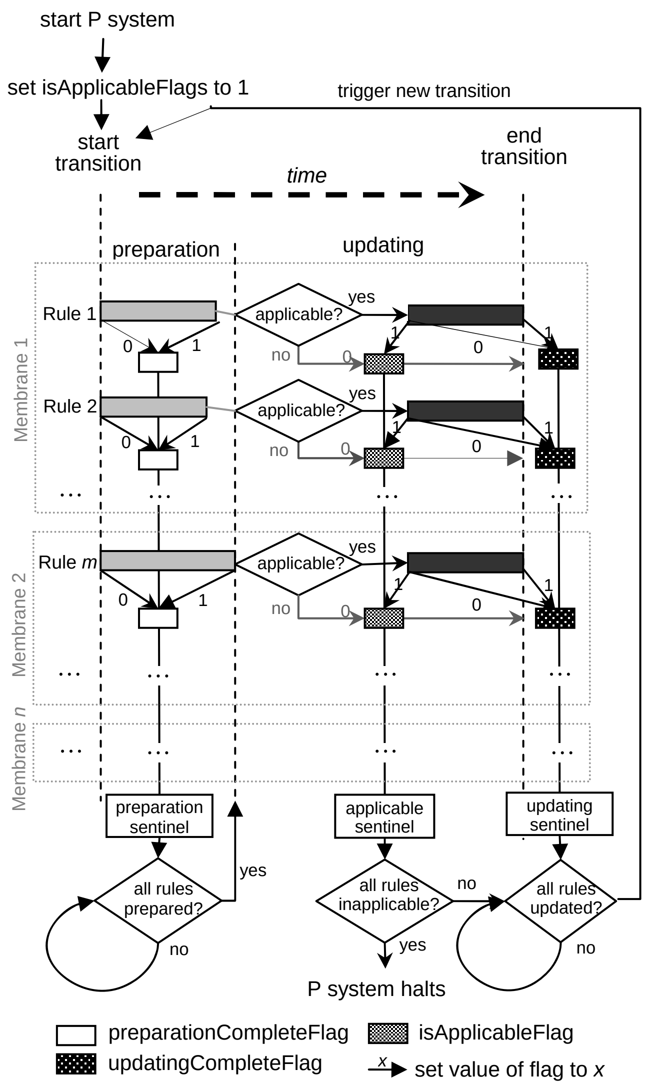


Fig. 6. An illustration of the synchronisation performed to accomplish a transition of a P system.

This occurs if the reaction rules consume or produce the same type of object in the same region in the same transition. As mentioned in Section 2.4, Petreska and Teuscher's hardware implementation avoids the conflict problem by totally sacrificing the parallelism that gives rise to the problem. This is an undesirable strategy, because it hinders performance significantly.

Reconfig-P implements two alternative conflict resolution strategies: the *time-oriented strategy* and the *space-oriented strategy.* The time-oriented strategy consumes time, whereas the space-oriented strategy consumes space. Therefore, the best strategy to use depends on whether it is more important to optimise space or time usage. The user selects the strategy to be used.

Both strategies involve determining in software before run-time all of the potential conflicts that might occur between reaction rules, and then generating the hardware circuit for the P system in such a way that all processing units can execute independently without any possibility of writing to the same register at the same time. The task of determining the resource conflicts has a time complexity of $\Theta(n_r n_o)$, where n_r is the number of reaction rules in the P system, and n_o is the number of object types in the alphabet of the P system. Therefore it has a negligible impact on performance. Note that, since the circuit for the P system need be generated only once, the task is performed only once.

In both strategies, potential conflicts are determined through the construction of a *conflict matrix.* Each row of a conflict matrix for a P system is a quadruple (p, q, r, s), where p is an object type in the alphabet of the P system, q is a region in the P system, r is the set of reaction rules whose application results in the consumption and/or production of objects of type p in q, and s — called the *conflict degree* of (p, q) — is the size of r. There is a row for every pair (p, q).

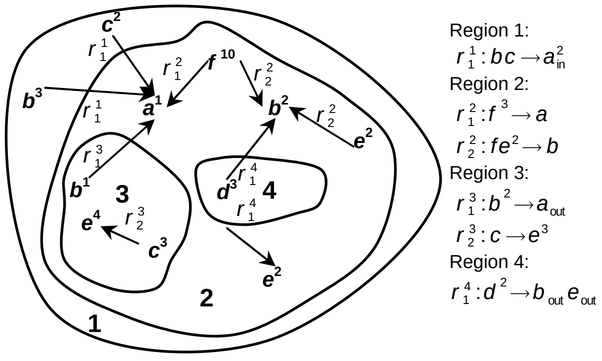
We now describe how the updating phase occurs when (a) the time-oriented strategy is used, and (b) the space-oriented strategy is used.

Time-oriented conflict resolution

In the time-oriented conflict resolution strategy, if two reaction rules need to update the multiplicity value for the same type of object in the same region, then they do so one after the other (the order in which they do so is not important and so is chosen arbitrarily).

Table 1 illustrates the time-oriented strategy. In the table, 'u(p, q)' denotes the operation of updating the multiplicity value of object type p in region q.

The correct interleaving of the various conflicting operations of the processing units is determined by means of analysis of the conflict matrix for the P system before run-time. That is, the Handel-C source code that is generated for the P system specifies the interleaving directly. This is achieved by inserting the appropriate number of single-clock-cycle delay statements in the appropriate places in the source code for the processing units. For example, the code in the processing unit for r_1^3 that updates the multiplicity value of object type a in region 2 is preceded by two delay statements, whereas the corresponding code for object type b in region 3 is not preceded by any delay statements. For the general case, take a quadruple (p, q, r, s) from the conflict matrix for a P system. Assume that the reaction rules $r_1, r_2, \ldots, r_n \in r$ are ordered (for the purpose of conflict resolution) according to the natural ordering of their subscripts. Then the number of delay statements to be inserted immediately before the code in the processing unit for the reaction rule $r_i \in r$ that updates the multiplicity value of object type p in region q is equal to $i-1$. As Table 1 illustrates, the number of clock cycles taken



c^2
r_1^1
b^3
r_1^1
a^1
r_1^2
f^{10}
r_2^2
b^2
r_2^2
e^2
r_1^3
b^1
3
e^4
r_2^3
c^3
d^3
r_1^4
4
r_1^4
e^2
2
1
Region 1:
$r_1^1 : bc \rightarrow a_{in}^2$
Region 2:
$r_1^2 : f^3 \rightarrow a$
$r_2^2 : fe^2 \rightarrow b$
Region 3:
$r_1^3 : b^2 \rightarrow a_{out}$
$r_2^3 : c \rightarrow e^3$
Region 4:
$r_1^4 : d^2 \rightarrow b_{out} e_{out}$

Fig. 7. An example P system configuration. An arrow labelled by reaction rule r from object type p_1 in region q_1 to object type p_2 in region q_2 means that p_1 is a reactant of r (taken from q_1) and p_2 is a product of r (produced in q_2).

to update the multiplicity value for object type p in region q of a P system is equivalent to the conflict degree of (p, q), which is recorded in the conflict matrix for the P system.

Let k be the highest conflict degree in the conflict matrix for a P system. Then the updating phase for the P system takes k clock cycles to complete when the time-oriented conflict resolution strategy is used.

Table 1. How the processing units for the reaction rules in the P system in Figure 7 execute during the updating phase of the current transition if the time-oriented conflict resolution strategy is used.

Clock cycle	r_1^1	r_1^2	r_2^2	r_1^3	r_2^3	r_1^4
1	u(a,2), u(b,1), u(c,1) end	u(f,2)	u(b,2), u(e,2)	u(b,3)	u(c,3), u(e,3) end	u(d,4), u(e,2)
2		u(a,2) end	u(f,2) end			u(b,2) end
3				u(a,2) end		

Space-oriented conflict resolution

In the space-oriented conflict resolution strategy, if n reaction rules need to update the multiplicity value for the same type of object in the same region, then n copies are made of the register that stores that multiplicity value. The processing units for the conflicting reaction rules are assigned one copy register each, and in the updating phase write to their respective copy registers (see Figure 5 for an example). Once all of the processing units for reaction rules have completed writing to their registers, processing units called *multiset replication coordinators* (each of which is associated with one object type in one region and runs in parallel with the other processing units in Reconfig-P) read the values that have been stored in the copy registers, and set the original registers in the relevant multiset data structures accordingly (again see Figure 5). This step takes one clock cycle to complete. However, for P systems with a large number of object copies, it may be beneficial to perform logic depth reduction (see Section 3.5).

Table 2 illustrates the space-oriented strategy.

Table 2. How the processing units for the reaction rules in the P system in Figure 7 execute during the updating phase of the current transition if the space-oriented conflict resolution strategy is used.

Clock cycle	r_1^1	r_1^2	r_2^2	r_1^3	r_2^3	r_1^4
1	u(a,2), u(b,1), u(c,1) end	u(f,2) u(a,2) end	u(b,2), u(e,2) u(f,2) end	u(b,3) u(a,2) end	u(c,3), u(e,3) end	u(d,4), u(e,2) u(b,2) end
2	Multiset replication coordinators update original registers in relevant multiset data structures					

4 Evaluation of Reconfig-P

In this section, we evaluate the performance, flexibility and scalability of Reconfig-P. First, we present a theoretical analysis of the performance of Reconfig-P. Then we present and discuss empirical results that give insight into the performance and hardware resource usage of Reconfig-P. Finally we comment on the flexibility of Reconfig-P.

4.1 Theoretical evaluation of the performance of Reconfig-P

Figure 8 presents the time complexity of the parallel algorithm executed by Reconfig-P (in both the time-oriented and space-oriented modes) as well as the time complexity of the sequential algorithm used in sequential implementations of membrane computing. Table 3 illustrates the relative theoretical performances of (a) the sequential algorithm, (b) an algorithm that implements parallelism only at the system level (as in Petreska and Teuscher's computing platform), (c) the time-oriented parallel algorithm executed by Reconfig-P, and (d) the space-oriented parallel algorithm executed by Reconfig-P. It does so by evaluating their time complexities for example P systems.

In Table 3, larger and larger P systems are derived from one initial basic P system using either horizontal cascading or vertical cascading. In horizontal cascading, more and more regions are added, but the number of reaction rules per region is held constant. In vertical cascading, the number of reaction rules per region increases, but the number of regions is held constant. The following assumptions, deemed to represent the average case, are made: (a) there are 20 object types; (b) each reaction rule has four reactant object types, four catalyst object types and four product object types; (c) there are conflicts on 20% of the object types in the preparation phase; and (d) there are conflicts on 60% of the object types in the updating phase. Assumption (d) gives rise to a k value for each P system, which is the highest conflict degree in the conflict matrix for the P system. P systems are also assigned an arbitrary a value, which is the percentage of reaction rules that are applicable in a transition on average.

Table 3 clearly demonstrates the superior speed of the algorithm executed by Reconfig-P over both the sequential algorithm and the algorithm with one level of parallelism. When horizontal cascading is applied, the time-oriented and space-oriented algorithms both show exceptional scalability. When the k value is small and reaction rules are evenly distributed across regions, the time-oriented algorithm is more effective than the space-oriented algorithm because it uses less space while achieving similar speeds. When vertical cascading is applied, the time-oriented and space-oriented algorithms are significantly faster than the algorithm with one level of parallelism. The space-oriented algorithm is faster than the time-oriented algorithm. The main reason is that, whereas increasing the k value reduces the degree of parallelism in the updating phase when the time-oriented algorithm is used, this is not the case when the space-oriented algorithm is used.

4.2 Empirical evaluation of the performance and scalability of Reconfig-P

We have conducted a series of experiments to investigate the performance and hardware resource usage of Reconfig-P. In this section, we present and discuss the results of these experiments, and evaluate the performance and scalability of Reconfig-P in light of these results.

Details of the experiments

Table 4 shows the P systems that were executed in the experiments. Each P system was constructed by first taking n copies of the basic P subsystem shown

Table 3. An illustration of the time complexity results presented in Figure 8.

Regions	Rules	*k*	*a* (%)	Number of clock cycles per transition			
				Sequential	1-level parallelism	2-level parallelism (time-oriented)	2-level parallelism (space-oriented)
Horizontal cascading							
10	50	3	30	470	25	10	10
10	50	3	70	630	28	10	10
50	250	3	30	2350	27	12	13
50	250	3	70	3150	30	12	13
100	500	3	30	4700	27	12	13
100	500	3	70	6300	30	12	13
Vertical cascading							
10	50	3	30	470	25	10	10
10	50	3	70	630	28	10	10
10	250	15	30	2350	186	36	25
10	250	15	70	3150	351	36	25
10	500	30	30	4700	606	66	40
10	500	30	70	6300	1206	66	40

at the top-right of Figure 9, then cascading these copies in a horizontal, vertical or horizontal and vertical manner (as shown in Figure 9), and finally placing the copies into the region shown at the top-left of Figure 9. The value of n is a measure of the size of the constructed P system; the larger the value of n, the larger the P system. Thus in the experiments a series of P systems of different sizes and different structures were executed.

Table 4 lists a C value for each P system. The C value for a P system is a measure of the amount of conflict that exists between reaction rules in the P system. More specifically, C is the sum of the conflict degrees of all pairs (p, q) for the P system, where p is an object type, q is a region and the conflict degree of the pair is greater than 1.

The target circuit for the experiment was the Xilinx Virtex-II XC2V6000FF11 52-4, and the Handel-C code for the P systems was synthesised, placed and routed using Xilinx tools.

Table 4. Details of the P systems used in the experiments.

P system	*n*	Regions	Horizontal cascading			Vertical cascading			Horizontal and vertical cascading		
			Rules	*C*	*k*	Rules	*C*	*k*	Rules	*C*	*k*
1	1	4	11	27	7	11	27	7	11	27	7
2	2	7	21	53	7	22	54	7	22	54	7
3	4	13	41	97	7	44	108	7	42	106	7
4	8	25	81	193	9	88	216	7	83	211	7
5	16	49	161	377	17	176	432	7	165	415	7

Evaluation of the hardware resource usage of Reconfig-P

Table 5 shows experimental data related to the hardware resource usage of Reconfig-P, both when it executes in time-oriented mode and when it executes in space-oriented mode. We use the number of LUTs (lookup tables) on the circuit generated for a P system as the measure of the hardware resource usage of Reconfig-P for that P system. We also record the percentage of the LUTs available on the FPGA that is used by the circuit, because this percentage provides an indication of the extent to which current FPGA technology meets the hardware resource requirements of Reconfig-P.

Table 5. Experimental results related to the hardware resource usage of Reconfig-P in both the time-oriented and space-oriented modes.

P system	*n*	Time-oriented mode		Space-oriented mode	
		Number of LUTs	% of LUTs	Number of LUTs	% of LUTs
Horizontal cascading					
1	1	1046	1.55%	1102	1.63%
2	2	1667	2.47%	1801	2.66%
3	4	3058	4.52%	3248	4.81%
4	8	5752	8.51%	6060	8.97%
5	16	11106	16.43%	11719	17.34%
Vertical cascading					
1	1	1046	1.55%	1102	1.63%
2	2	1959	2.9%	2075	3.07%
3	4	3570	5.32%	3790	5.65%
4	8	6771	10.09%	7344	10.87%
5	16	13207	19.79%	14486	21.43%
Horizontal and vertical cascading					
1	1	1046	1.55%	1102	1.63%
2	2	1934	2.94%	1934	2.94%
3	4	3479	5.16%	3689	5.46%
4	8	6597	9.76%	7293	10.79%
5	16	12780	19.41%	13360	20.1%

Figures 10 and 11 illustrate the experimental data in graphical form. We make the following observations:

- The hardware resource usage of Reconfig-P scales linearly with respect to the size of the P system executed (i.e., with respect to n). This is as good as can reasonably be expected, and indicates that Reconfig-P is scalable with respect to hardware resource usage.

- The type of cascading employed in the construction of the P system that is executed has little effect on hardware resource usage.
- For all P systems, Reconfig-P uses less than 22% of the LUTs available on the FPGA. Given that the largest P system has 49 regions and 176 reaction rules, this is an impressive result. Not only does it strongly suggest that current FPGA technology meets the hardware resource requirements of Reconfig-P, it also indicates that it would be feasible to extend Reconfig-P to support P system features not covered by the core P system model.
- Reconfig-P uses only slightly more hardware resources in space-oriented mode than in time-oriented mode. This suggests that, at least for the P systems executed in the experiments, multiset replication has only a relatively small effect on hardware usage. Indeed, even for P systems with $C > 400$ and therefore with more than 400 copies of multiplicity values, the hardware resources consumed by Reconfig-P to store, access and coordinate these copies is relatively small.

In summary, the experimental results indicate that Reconfig-P makes efficient use of hardware resources, and therefore is scalable with respect to hardware resource usage. The fact that less than 22% of the available hardware resources is used for even relatively large P systems augurs well for the flexibility of Reconfig-P (see Section 4.3 for more on this point).

Evaluation of the performance of Reconfig-P

Table 6 shows the number of clock cycles that Reconfig-P executes per transition for each of the P systems used in the experiments, both when it executes in time-oriented mode and when it executes in space-oriented mode. The values shown in the table were determined empirically.

We make the following observations:

- Nearly all of the P systems used in the experiments have $k = 7$. When it executes in time-oriented mode, Reconfig-P takes 14 clock cycles to execute a transition if $k = 7$. When $k = 9$, it takes 16 clock cycles, and when $k = 17$, it takes 24 clock cycles. This is exactly as expected, given that the number of clock cycles taken to execute the updating phase across all regions in a P system is equal to k when the time-oriented conflict resolution strategy is used (see Section 3.6).
- When it executes in space-oriented mode, Reconfig-P takes 7 clock cycles to execute a transition for all the P systems used in the experiments. This suggests that when the space-oriented conflict resolution strategy is used, Reconfig-P shows exceptional scalability with respect to the number of clock cycles it takes per transition.
- Overall, Reconfig-P shows excellent scalability with respect to the number of clock cycles it takes per transition. However, when it executes in time-oriented

Table 6. Experimental data related to the number of clock cycles Reconfig-P takes to execute one P system transition.

P system	*n*	*k*	Number of clock cycles per transition	
			Time-oriented mode	Space-oriented mode
Horizontal cascading				
1	1	7	14	7
2	2	7	14	7
3	4	7	14	7
4	8	9	16	7
5	16	17	24	7
Vertical cascading				
1	1	7	14	7
2	2	7	14	7
3	4	7	14	7
4	8	7	14	7
5	16	7	14	7
Horizontal and vertical cascading				
1	1	7	14	7
2	2	7	14	7
3	4	7	14	7
4	8	7	14	7
5	16	7	14	7

mode and horizontal cascading is applied, increases in the size of the P system to be executed can lead to increases in the value of k, and hence increases in the number of clock cycles per transition.

- Reconfig-P takes considerably less clock cycles per transition when it executes in space-oriented mode than when it executes in time-oriented mode. This is as expected, given that the updating phase executes in a maximally parallel manner when the space-oriented conflict resolution strategy is used, but only in a partially parallel manner when the time-oriented conflict resolution strategy is used.

Table 7 shows the performance of Reconfig-P for each of the P systems used in the experiments, both when it executes in time-oriented mode and when it executes in space-oriented mode. It also shows, for the sake of comparison, the corresponding results for a software-based sequential computing platform (i.e., a Java simulator for the core P system model).

Figures 12, 13 and 14 illustrate the experimental data in graphical form. We make the following observations:

Table 7. Experimental data related to the performance of Reconfig-P (in both the space-oriented and time-oriented modes) and a software-based sequential computing platform.

P system	*n*	Reaction rule applications per second		
		Software-based sequential computing platform	Reconfig-P (space-oriented mode)	Reconfig-P (time-oriented mode)
Horizontal cascading				
1	1	3.7×10^5	57×10^5	53×10^5
2	2	4.5×10^5	120×10^5	110×10^5
3	4	5.3×10^5	230×10^5	210×10^5
4	8	4.7×10^5	460×10^5	410×10^5
5	16	3.5×10^5	910×10^5	710×10^5
Vertical cascading				
1	1	3.7×10^5	57×10^5	53×10^5
2	2	5.2×10^5	126×10^5	116×10^5
3	4	4.3×10^5	250×10^5	230×10^5
4	8	3.2×10^5	500×10^5	460×10^5
5	16	2×10^5	1000×10^5	890×10^5
Horizontal and vertical cascading				
1	1	3.7×10^5	57×10^5	53×10^5
2	2	4.1×10^5	126×10^5	115×10^5
3	4	4.5×10^5	240×10^5	220×10^5
4	8	3.7×10^5	470×10^5	430×10^5
5	16	2.7×10^5	930×10^5	820×10^5

- Reconfig-P executes P systems significantly faster than the software-based sequential computing platform (from 14 to 500 times faster). The larger the P system that is executed, the greater the extent to which Reconfig-P outperforms the sequential computing platform. This is as expected, because larger P systems have more regions and more reaction rules and therefore more opportunity for parallelism at both the system and region levels.
- In general, the performance of Reconfig-P in both the space-oriented and time-oriented modes increases linearly with respect to the size of the P system that is executed. This is a good result, because it indicates that as the size of the P system to be executed increases, Reconfig-P is able to take advantage of the increased opportunities for parallelism. That is, there does not appear to be any significant problems of scale in the hardware design (e.g., nonlinearly growing logic depths in certain parts of the hardware circuit that would reduce the clock rate of the FPGA).
- Reconfig-P performs better in space-oriented mode than in time-oriented mode, although only by approximately 10%. This relatively small difference is a con-

sequence of the fact that the various P systems used in the experiments have small k values. If the k values were larger, we would expect to observe a more pronounced difference in performance between the space-oriented and time-oriented modes.

In summary, the experimental results indicate that Reconfig-P achieves very good performance.

4.3 Evaluation of the flexibility of Reconfig-P

In its current prototype form, Reconfig-P supports the basic P system features covered by the core P system model. Therefore it is not able to execute P systems that include additional features such as structured objects and membrane permeability. This counts against its flexibility. However, there is good reason to believe that Reconfig-P can be extended to support additional P system features. As we have observed, Reconfig-P exhibits exceptionally economic hardware resource usage: for the P systems used in the experiments, approximately 75% of the available hardware resources are left unused. Thus there is ample space on the FPGA for the inclusion of additional data structures and logic required for the implementation of additional features. Furthermore, the fact that Reconfig-P is implemented in a high-level hardware description language should ease the process of incorporating additional features into the existing implementation.

5 Conclusion

By developing Reconfig-P, we have demonstrated that it is possible to efficiently implement both the system-level and region-level parallelism of P systems on reconfigurable hardware and thereby achieve significant performance gains.

Theoretical results demonstrate that the algorithm executed by Reconfig-P is significantly faster than the sequential algorithm used in sequential implementations of membrane computing. Empirical results show that for a variety of P systems Reconfig-P achieves very good performance while making economical use of hardware resources. And there is good reason to believe that Reconfig-P can be extended in the future to support additional P system features. Therefore, there is strong evidence that the implementation approach on which Reconfig-P is based is a viable means of attaining a good balance between performance, flexibility and scalability in a parallel computing platform for membrane computing applications.

References

1. Alhazov, A. and Sburlan, D. 2006. Static Sorting P Systems. In [14], 215-252.

2. Ardelean, I. I., Besozzi, D., Garzon, M. H., Mauri, G. and Roy, S. 2006. P System Models for Mechanosensitive Channels. In [14], 43-82.
3. Ardelean, I. I. and Cavaliere, M. 2003. Modelling Biological Processes by Using a Probabilistic P System Software. *Natural Computing*, 2(2), 173-197.
4. Balbontín Noval, D., Pérez-Jiménez, M. J. and Sancho-Caparrini, F. 2003. A MzScheme Implementation of Transition P Systems. In Păun, G., Rozenberg, G., Salomaa, A. and Zandron, C. (eds) *Membrane Computing: International Workshop WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002. Revised Papers*. LNCS 2597, Springer, 2003, 58-73.
5. Baranda, A. V., Castellanos, J., Arroyo, F. and Gonzalo, R. 2001. Towards an Electronic Implementation of Membrane Computing: A Formal Description of Non-deterministic Evolution in Transition P Systems. In Jonoska, N. and Seeman, N. C. (eds) *DNA Computing: 7th International Workshop on DNA-Based Computers, DNA7, Tampa, FL, USA, June 2001. Revised Papers*. LNCS 2340, Springer, 2002, 350-359.
6. Bel Enguix, G. and Jiménez-Lopez, M. D. 2006. Linguistic Membrane Systems and Applications. In [14], 347-388.
7. Bianco, L. Introduction to Psim. `http://psystems.disco.unimib.it/software/Bianco/psim.pdf`
8. Bianco, L., Fontana, F., Franco, G. and Manca, V. 2006. P Systems for Biological Dynamics. In [14], 83-128.
9. Cavaliere, M. 2003. Evolution-Communication P Systems. In Păun, G., Rozenberg, G., Salomaa, A. and Zandron, C. (eds) *Membrane Computing: International Workshop WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002. Revised Papers*. LNCS 2597, Springer, 2003, 134-145.
10. Cavaliere, M. and Ardelean, I. I. 2006. Modeling Respiration in Bacteria and Respiration/Photosynthesis Interaction in Cyanobacteria Using a P System Simulator. In [14], 129-158.
11. Cavaliere, M. and Sedwards, S. 2006. Membrane Systems with Peripheral Proteins: Transport and Evolution. Technical report TR-04-2006. Microsoft Research-University of Trento Centre for Computational and Systems Biology. `http://www.msr-unitn.unitn.it/Rpty_Tech.php`.
12. Ciobanu, G. 2006. Modeling Cell-Mediated Immunity by Means of P Systems. In [14], 159-180.
13. Ciobanu, G. and Paraschiv, D. 2002. P Systems Software Simulator. *Fundamenta Informaticae*, 49(1-3), 61-66.
14. Ciobanu, G., Păun, G. and Pérez-Jiménez, M. J. (eds). 2006. *Applications of Membrane Computing*. Springer-Verlag.
15. Ciobanu, G. and Guo, W. 2004. P Systems Running on a Cluster of Computers. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G. and Salomaa, A. (eds) *Membrane Computing. International Workshop, WMC2003, Tarragona, Spain, July 2003. Revised Papers*. LNCS 2933, Springer, 123-139.

16. Cordón-Franco, A., Gutiérrez-Naranjo, M. A., Pérez-Jiménez, M. J. and Sancho-Caparrini, F. 2004. A Prolog Simulator for Deterministic P Systems with Active Membranes. *New Generation Computing*, 22(4), 349-363.
17. Fernandez, L., Arroyo, F., Tejedor, J. A. and Castellanos, J. 2006. Massively Parallel Algorithm for Evolution Rules Application in Transition P Systems. Pre-proceedings of the Seventh Workshop on Membrane Computing, Leiden, The Netherlands, July 17-21, 2006, 337-343.
18. Fernandez, L., Martinez, V. J., Arroyo, F. and Mingo, L. F. 2005. A Hardware Circuit for Selecting Active Rules in Transition P Systems. Pre-pro- ceedings of the First International Workshop on Theory and Application of P Systems, Timisoara, Romania, September 26-27, 2005, 45-48.
19. Gramatovici, R. and Bel Enguix, G. 2006. Parsing with P Automata. In [14], 389-410.
20. Michel, O. and Jacquemard, F. 2006. An Analysis of a Public Key Protocol with Membranes. In [14], 283-302.
21. Nepomuceno-Chamarro, I. A. 2004. A Java Simulator for Basic Transition P Systems. *Journal of Universal Computer Science*, 10(5), 620-629.
22. Nishida, T. Y. 2006. A Membrane Computing Model of Photosynthesis. In [14], 181-202.
23. Nishida, T. Y. 2006. Membrane Algorithms: Approximate Algorithms for NP-Complete Optimization Problems. In [14], 303-314.
24. Păun, G. 2002. *Membrane Computing: An Introduction*. Springer.
25. Păun, G. and Rozenberg, G. 2002. A guide to membrane computing. *Theoretical Computer Science*, 287, 73-100.
26. Pérez-Jiménez, M. J. and Romero-Campero, F. 2004. A CLIPS Simulator for Recognizer P Systems with Active Membranes. In Păun, G., Riscos-Núñez, A., Romero-Jiménez, A. and Sancho-Caparrini, F. (eds) *Proceedings of the Second Brainstorming Week on Membrane Computing*, Sevilla, Spain, February 2-7, 2004, 387-413.
27. Petreska, B. and Teuscher, C. 2004. A Reconfigurable Hardware Membrane System. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G. and Salomaa, A. (eds) *Membrane Computing. International Workshop, WMC2003, Tarragona, Spain, July 2003. Revised Papers.* LNCS 2933, Springer, 269-285.
28. Suzuki, Y. and Tanaka, H. 2000. On a LISP Implementation of a Class of P Systems. *Romanian J. Information Science and Technology*, 3(2), 173-186.
29. Suzuki, Y. and Tanaka, H. 2006. Modeling p53 Signaling Pathways by Using Multiset Processing. In [14], 203-214.
30. Syropoulos, A., Mamatas, E. G., Allilomes, P. C. and Sotiriades, K. T. 2004. A Distributed Simulation of Transition P Systems. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G. and Salomaa, A. (eds) *Membrane Computing. International Workshop, WMC2003, Tarragona, Spain, July 2003. Revised Papers.* LNCS 2933, Springer, 357-368.

Definitions
$M = \{m_1, m_2, ..., m_n\}$ is the set of membranes in the P system. $V = \{o_1, o_2, ..., o_v\}$ is the alphabet of the P system. $R_{m_x} = \left\{r_{1,m_x}, r_{2,m_x}, ..., r_{k_{m_x},m_x}\right\}$ is the set of reaction rules in the region defined by membrane m_x. r_{y,m_x} is the y^{th} reaction rule in the region defined by membrane m_x. r^a_{y,m_x} denotes that r_{y,m_x} is applicable. $n^{\text{R}}(r_{y,m_x})$, $n^{\text{C}}(r_{y,m_x})$ and $n^{\text{P}}(r_{y,m_x})$ denote the number of reactant, catalyst and product object types in reaction rule r_{y,m_x}, respectively. $M^{\text{UPDATE}}_{m_x} : V \longrightarrow R_{m_x}$ maps each object type in the region defined by membrane m_x to the set of reaction rules that might update its multiplicity. $Max^n_{i=1}x_i$ is the function that returns the maximum value in the set $\{x_1, x_2, ..., x_n\}$. In the following performance analysis, e denotes the time taken to execute one transition of a P system. This time is composed of the separate times taken to execute the preparation phase (p), the updating phase (u) and (in the parallel algorithm) synchronisation operations (s). Synchronisation operations include updates of the sentinels preparationSentinel and updatingSentinel, as well as operations related to multiset replication coordination. Times are measured in clock cycles.

Sequential algorithm

$$e^{\text{SEQ}} = \sum_{i=1}^{n}\sum_{j=1}^{k_{m_i}} = \begin{cases} p^{\text{SEQ}}(r_{j,m_i}) & \text{, if } r_{j,m_i} \text{ is not applicable} \\ p^{\text{SEQ}}(r_{j,m_i}) + u^{\text{SEQ}}(r_{j,m_i}) & \text{, if } r_{j,m_i} \text{ is applicable} \end{cases}$$

where

$$p^{\text{SEQ}}(r_{y,m_x}) = n^{\text{R}}(r_{y,m_x}) + n^{\text{C}}(r_{y,m_x}) - 1$$

and

$$u^{\text{SEQ}}(r^a_{y,m_x}) = n^{\text{R}}(r^a_{y,m_x}) + n^{\text{P}}(r^a_{y,m_x}).$$

Parallel algorithm

$$e^{\text{PAR}} = Max^n_{i=1}Max^{k_{m_i}}_{j=1}p^{\text{PAR}}(r_{j,m_i}) + Max^n_{i=1}Max^{k_{m_i}}_{j=1}u^{\text{PAR}}(r^a_{j,m_i}) + s^{\text{PAR}},$$

where

$$p^{\text{PAR}}(r_{y,m_x}) = \begin{cases} \sum_{s=1}^{y} \log_2(n^{\text{R}}(r_{s,m_x}) + n^{\text{C}}(r_{s,m_x})), \\ \qquad \text{if } r_{s,m_x} \text{ has an assigned priority} \\ \log_2(n^{\text{R}}(r_{y,m_x}) + n^{\text{C}}(r_{y,m_x})), \\ \qquad \text{if } r_{y,m_x} \text{ does not have an assigned priority} \end{cases}$$

and

$$Max^n_{i=1}Max^{k_{m_i}}_{j=1}u^{\text{PAR}}(r^a_{j,m_i}) = \begin{cases} Max^n_{i=1}Max^v_{j=1}\left|M^{\text{UPDATE}}_{m_i}(o_j)\right|, \\ \qquad \text{if the time-oriented strategy is used} \\ 1, \\ \qquad \text{if the space-oriented strategy is used} \end{cases}$$

and

$$s^{\text{PAR}} = \begin{cases} 2(\log_x \left\lceil \sum_{i=1}^{n} |R_{m_i}| \right\rceil - 1) & \text{, if the time-oriented strategy is used.} \\ 2(\log_x \left\lceil \sum_{i=1}^{n} |R_{m_i}| \right\rceil - 1) + 1 & \text{, if the space-oriented strategy is used.} \end{cases}$$

Fig. 8. The time complexities of the parallel algorithm executed by Reconfig-P and the sequential algorithm executed by sequential implementations of membrane computing.

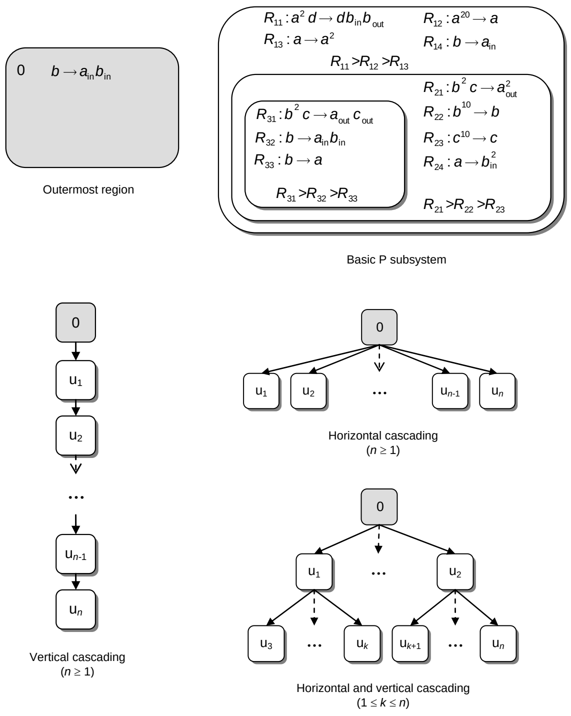

0 $b \rightarrow a_{in} b_{in}$
Outermost region
$R_{11} : a^2 d \rightarrow d b_{in} b_{out}$
$R_{12} : a^{20} \rightarrow a$
$R_{13} : a \rightarrow a^2$
$R_{14} : b \rightarrow a_{in}$
$R_{11} > R_{12} > R_{13}$
$R_{21} : b^2 c \rightarrow a^2_{out}$
$R_{22} : b^{10} \rightarrow b$
$R_{23} : c^{10} \rightarrow c$
$R_{24} : a \rightarrow b^2_{in}$
$R_{21} > R_{22} > R_{23}$
$R_{31} : b^2 c \rightarrow a_{out} c_{out}$
$R_{32} : b \rightarrow a_{in} b_{in}$
$R_{33} : b \rightarrow a$
$R_{31} > R_{32} > R_{33}$
Basic P subsystem
0
u_1
u_2
u_{n-1}
u_n
Vertical cascading ($n \geq 1$)
Horizontal cascading ($n \geq 1$)
u_3
u_k
u_{k+1}
Horizontal and vertical cascading ($1 \leq k \leq n$)

Fig. 9. Each P system used in the experiments was constructed by first cascading n copies of a basic P subsystem in a horizontal, vertical or horizontal and vertical manner, and then placing these copies into an outermost region.

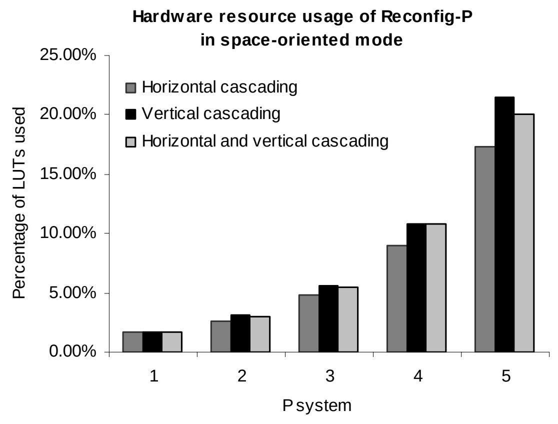


Fig. 10. An illustration of the experimental results related to the hardware resource usage of Reconfig-P in space-oriented mode.

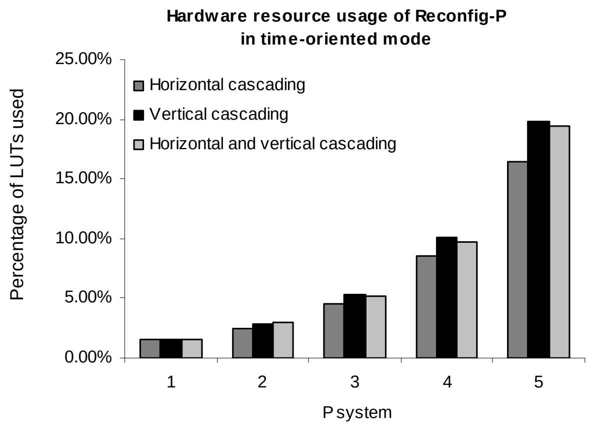


Fig. 11. An illustration of the experimental results related to the hardware resource usage of Reconfig-P in time-oriented mode.

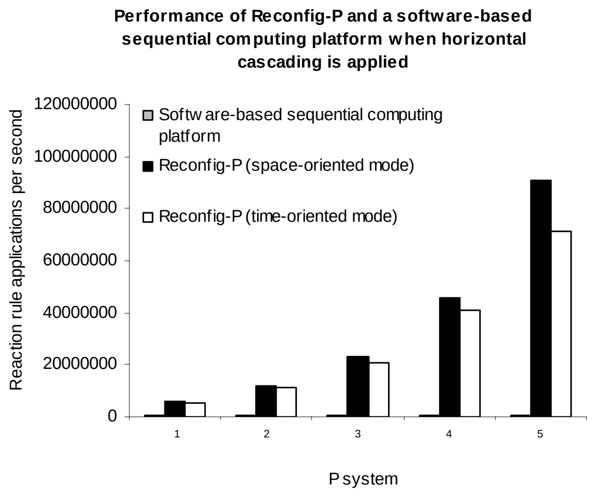


Fig. 12. An illustration of the experimental data related to the performance of Reconfig-P (in both the space-oriented and time-oriented modes) and a software-based sequential computing platform when horizontal cascading is applied.

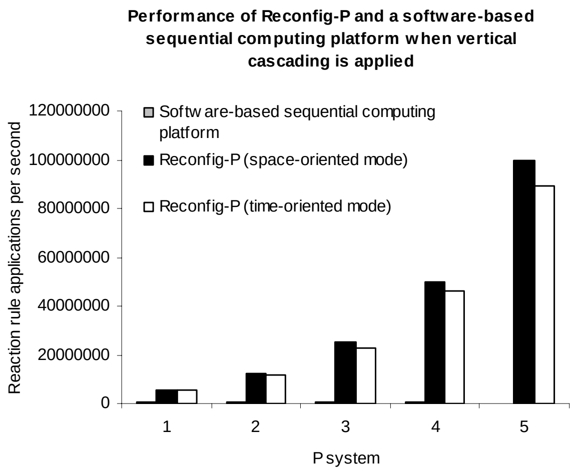


Fig. 13. An illustration of the experimental data related to the performance of Reconfig-P (in both the space-oriented and time-oriented modes) and a software-based sequential computing platform when vertical cascading is applied.

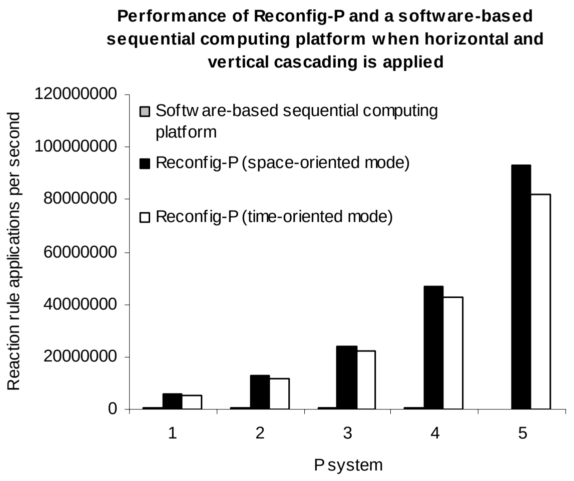


Fig. 14. An illustration of the experimental data related to the performance of Reconfig-P (in both the space-oriented and time-oriented modes) and a software-based sequential computing platform when horizontal and vertical cascading is applied.