A Region-Oriented Hardware Implementation for Membrane Computing Applications and Its Integration into Reconfig-P

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. We have recently developed a prototype hardware implementation of membrane computing based on reconfigurable computing technology called Reconfig-P. The existing hardware design treats reaction rules as the primary computational entities and represents regions only implicitly. In this paper, we present an alternative hardware design that more directly reflects the intuitive conceptual understanding of a P system and therefore promotes the extensibility of Reconfig-P. A key feature of the design is the fact that regions, rather than reaction rules, are the primary computational entities. More specifically, in the design, regions are represented as loosely coupled processing units which communicate objects by message passing. Experimental results show that for many P systems the region-oriented and rule-oriented designs exhibit similar performance and hardware resource consumption. To accomplish a seamless integration of the rule-oriented and region-oriented designs and other alternative implementation strategies in Reconfig-P, and to make Reconfig-P amenable to future integration of additional implementation strategies, we have produced a new version of P Builder, our intelligent hardware source code generator. The sophisticated new design for P Builder was produced in accordance with a novel design pattern called Content-Form-Strategy. We describe the design and implementation of the new version of P Builder in the paper.

1 Introduction

We have recently developed a prototype hardware implementation of membrane computing based on reconfigurable computing technology called Reconfig-P. The existing hardware design treats reaction rules as the primary computational entities and represents regions only implicitly. Consequently there is not always a direct mapping between the components of the intuitive conceptual understanding of a P system and the hardware components. Such indirectness is a byproduct of our attempt to simplify the hardware circuit and therefore promote the performance and efficiency of Reconfig-P. Nevertheless, a more faithful rendering of the intuitive conceptual understanding of a P system in hardware would have benefits

for the extensibility of Reconfig-P. In particular, it would facilitate the process of augmenting Reconfig-P to support additional types of P systems. In this paper, we present an alternative hardware design that more directly reflects the intuitive conceptual understanding of a P system and therefore promotes the extensibility of Reconfig-P. A key feature of the design is the fact that regions, rather than reaction rules, are the primary computational entities. More specifically, in the design, regions are represented as loosely coupled processing units which communicate objects by message passing. Experimental results show that for many P systems the region-oriented and rule-oriented designs exhibit similar performance and hardware resource consumption. To accomplish a seamless integration of the rule-oriented and region-oriented designs and other alternative implementation strategies in Reconfig-P, and to make Reconfig-P amenable to the future integration of additional implementation strategies, we have produced a new version of P Builder, our intelligent hardware source code generator. The sophisticated new design for P Builder was produced in accordance with a novel design pattern called Content-Form-Strategy. We describe the design and implementation of the new version of P Builder in the paper.

The contents of the paper are as follows. In Section 2, we discuss the background to the research described in the paper. In Section 3, we describe the region-oriented hardware design. In Section 4, we explain some aspects of our implementation of regions in hardware. In Section 5, we describe the motivation for a new version of P Builder, and describe its design and implementation. In Section 6, we present the results of an empirical analysis of the hardware resource consumption and performance of hardware circuits using the region-oriented design. Finally, in Section 7, we draw some conclusions regarding the significance of our contributions.

2 Background

2.1 The intuitive conceptual understanding of a P system

Although in one sense a P system is a pure mathematical construct, in another sense a P system is seen as having non-mathematical properties. For example, in an informal discussion of P systems one might speak of membranes 'dissolving', of regions being 'inside' other regions, or of objects being 'consumed' by reaction rules. The very frequent use of such physicalistic metaphors in describing the operation of a P system is, of course, a result of the fact that P systems have since their introduction been modelled after biological cells. The biological interpretation of a P system, far from being dispensable, provides one with a means of intuitively grasping the computational characteristics of P systems.

According to what we call the *intuitive conceptual understanding of a P system*, a P system comprises a hierarchy of membranes, each of which defines a region that contains a collection of objects and is associated with a set of reaction rules. The P system evolves in a series of stages. At each stage, the reaction rules in every region are applied. The application of the reaction rules in a region

results in the occurrence of an object transformation process within the region. The object transformation processes in the different regions occur independently. Sometimes an object transformation process results in the movement of objects between regions. Therefore, although the processes in the different regions occur independently, they may influence each other indirectly by influencing their respective inputs for the next stage of the evolution of the P system.

Given the intuitive conceptual understanding of a P system, in the context of implementing P systems on a computing platform, it is natural to regard a P system as a collection of distributed processing units (the object transformation processes occurring in the different regions) that interact only by means of message passing (the transfer of objects).

2.2 Current status of Reconfig-P

Reconfig-P [8] [9] is an implementation of membrane computing based on reconfigurable hardware (specifically, a field-programmable gate array[1]) that is able to execute P systems at high performance. It exploits the reconfigurability of the hardware by constructing and synthesising a customised hardware circuit for the specific P system to be executed. The hardware circuit is constructed using the hardware specification language Handel-C [2].

To maximise performance and minimise hardware resource consumption, the current version of Reconfig-P takes a minimalistic approach to the implementation of the features of a P system in hardware. According to this approach, only those features of the intuitive conceptual understanding of a P system absolutely necessary to the computational operation of a P system are implemented explicitly as processing units or data structures. As a consequence, some features that are of primary importance in the conceptual understanding of a P system are not explicitly represented as components of the hardware circuits generated by the current version of Reconfig-P. Most significantly, membranes and the regions defined by membranes are not explicitly represented. Instead, the existing implementation represents these features implicitly as logical constructions arising from the connections that exist between processing units corresponding to the reaction rules and arrays corresponding to the multisets of objects available in the regions of the P system. In other words, the conceptual model of a P system underlying the design of the current version of Reconfig-P includes only reaction rules and multisets of objects as primary features; membranes and regions are not directly represented in the model, but must be inferred on the basis of the connections that exist between the reaction rules and multisets of objects.

[1] A standard field-programmable gate array (FPGA) consists of a matrix of configurable logic blocks (CLBs). The CLBs, which are connected by means of a network of wires, can be used to implement logic or memory. 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.

2.3 Motivation for the alternative hardware design

Although it promotes performance and efficiency, the hardware design used in the existing version of Reconfig-P has some disadvantages. These disadvantages diminish the elegance, understandability (and therefore maintainability), flexibility and extensibility of Reconfig-P. First, by deviating from the intuitive conceptual understanding of a P system, the design is not as elegant and understandable as it could be. Second, the design does not facilitate the implementation of P systems that represent membranes as active entities or include membrane-mediated rules (such as symport and antiport rules). Third, the design removes the possibility of adopting an elegant region-oriented strategy for the distribution of computation across parallel processing units. These three disadvantages have motivated us to develop an alternative hardware design.

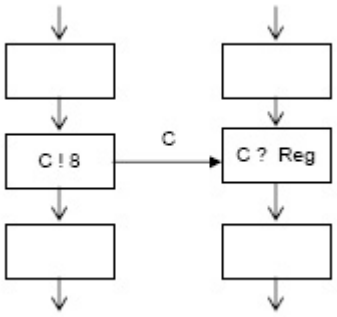
The alternative hardware design proposed in this paper, which we call the *region-oriented design*, is intended to

- promote the elegance and understandability of Reconfig-P by more closely reflecting the intuitive conceptual understanding of a P system,
- promote the extensibility of Reconfig-P by providing a framework within which the future implementation of additional types of P systems — especially P systems that include cell-to-cell connections (e.g., tissue-like P systems [12] and spiking neural P systems [6]), represent membranes as active entities, or include membrane-mediated rules (e.g., [1], [10], [3], [11] and [12]) — can more easily be achieved, and
- facilitate an elegant region-oriented approach to the distribution and parallelisation of the computational activities occurring in a P system.

A region-oriented approach to the distribution of the computational activities occurring in a P system is desirable because, not only does it match the intuitive conceptual understanding of how these activities are distributed in a P system, it also allows a very natural means of scaling the amount of available hardware resources to suit the size of the P system to be executed. For example, one can envision implementing a P system using multiple hardware circuits, where each hardware circuit implements the processing associated with a particular region (or subhierarchy of regions) of the P system. Indeed, the techniques developed in implementing a region-oriented approach could be adapted to allow the composition of whole P systems into larger systems. That is, these techniques could be adapted to allow hardware circuits implementing distinct P systems to communicate and therefore form a larger system.

3 The region-oriented hardware design

In this section, we provide an overview of the region-oriented hardware design. For the sake of simplicity, in this overview we do not treat aspects of the design related to nondeterministic object distribution.




```
main (void)
{
 unsigned 8 Reg;
 chan C;
 ...
 par
 {
   C ! 8; // 'send'
   C ? Reg; // 'receive'
 }
 ...
}
```

Fig. 1. Example of a Handel-C `chan` (channel) construct being used to implement communication between two parallel branches.

3.1 Basic characteristics of the design

In the region-oriented hardware design, instead of being represented only implicitly, regions are implemented explicitly as hardware components. More specifically, the design has the following three key attributes:

1. Regions are implemented as core processing units.
2. Region processing units operate independently. That is, each region processing unit coordinates all the activities occurring in one particular region of the P system and is not aware of activities occurring in other regions.
3. The movement of objects between regions is implemented as message passing between region processing units.

A key aspect of the region-oriented implementation is the use of the `chan` (channel) construct of Handel-C to accomplish inter-region communication. The `chan` construct supports the implementation of synchronous communication between parallel processing units. The example Handel-C code in Figure 1 shows a channel `C` being used to transfer the value 8 to the register `Reg`.

3.2 Region processing units

Similar to the rule processing units in the rule-oriented design, the region processing units in the region-oriented design complete the execution of a transition in two phases: an *object assignment phase* and an *object production phase*[2]. In the object assignment phase, a region processing unit determines the maximum number of

[2] The object assignment phase and object production phase roughly correspond to the preparation phase and updating phase in the rule-oriented design, respectively (see [8] for details)

instances, and hence the applicability status, for each reaction rule in the region in the current transition. In the object production phase, a region processing unit carries out the consumption, production and communication of objects for the reaction rules in the region based on their maximum number of instances. In the case of P systems that contain reaction rules with relative priorities, the region processing unit must calculate the maximum numbers of instances for those reaction rules with higher priorities before doing so for those reaction rules with lower priorities. To save clock cycles, the region processing unit carries out the object consumption for the reaction rules with higher priorities in the object assignment phase rather than in the object production phase.

Object assignment phase

An important aspect of the hardware design for the object assignment phase is the way in which the region processing unit respects the relative priorities of the reaction rules (if indeed such priorities are defined), while minimising the number of clock cycles required to complete the phase by avoiding the processing of inapplicable reaction rules.

It is an assumption of the design that reaction rules in a region that consume common object types are assigned relative priorities (using the relation $>$, which is to be interpreted as 'has higher priority than'). Given this assumption, the set of reaction rules in a region may be partitioned into (a) a collection of singleton sets, where for each reaction rule not related by priority to any other reaction rule, there is exactly one singleton set containing that reaction rule in the collection, and there are no other singleton sets in the collection, and (b) a collection of totally $>$-ordered sets, where each reaction rule related by priority to another reaction rule is in exactly one totally $>$-ordered set, and if two reaction rules have relative priorities then they belong to the same totally $>$-ordered set. In the example illustrated in Table 1, the columns correspond to totally $>$-ordered sets of reaction rules. The totally $>$-ordered sets are: $T_1 = \{R_{11}, R_{12}, R_{13}, R_{14}, R_{15}\}$, $T_2 = \{R_{21}, R_{22}, R_{23}, R_{24}, R_{25}\}$, and $T_3 = \{R_{31}, R_{32}, R_{33}, R_{34}\}$. From the sets in the partition formed in this way, one or more partially time-ordered sets of reaction rules can be constructed that may be interpreted as indicating the possible temporal orders in which the region processing unit can process the reaction rules in the object assignment phase. The constraints on the possible temporal orders are that (a) reaction rules with the same priority should be processed at the same time, and (b) reaction rules with relative priorities should be processed one after the other according to their priorities.

The temporal order in which the region processing unit should process reaction rules in the object assignment phase can be determined at compile-time as follows:

```
In parallel
   In sequence
      Process reaction rules in T1
   In sequence
      Process reaction rules in T2
   In sequence
      Process reaction rules in T3
```

Which of the possible temporal orders is actually followed depends on how many clock cycles are required to process each specific reaction rule.

It might appear that this static approach allows the degree of parallelisation to be maximised. However, the approach neglects the fact that reaction rules may be inapplicable at the outset of the object assignment phase or become inapplicable as other reaction rules are assigned objects, and therefore may not need to be fully processed in the phase. To maximise the performance of the implementation, we use a technique that avoids the processing of inapplicable reaction rules. Naturally, such a technique must be applied at run-time. The technique involves checking the applicability status of reaction rules both at the beginning of the phase and whenever any objects have been assigned to a reaction rule, and using this applicability information to determine the temporal order in which the currently applicable reaction rules should be processed in order to minimise the total number of clock cycles used in the remainder of the phase. For the example shown in Table 1, after checking the applicability of the reaction rules at the beginning of the object assignment phase, the region processing unit determines that only reaction rules R_{11}, R_{13}, R_{15}, R_{25}, R_{32} and R_{34} are applicable. Based on this information and the totally $>$-ordered sets T_1, T_2 and T_3, it determines that the currently most time-efficient way of processing the reaction rules is to first process R_{11}, R_{25} and R_{32} in parallel, then process R_{13} and R_{34} in parallel, and finally process R_{15}. It then proceeds to process R_{11}, R_{25} and R_{32} in parallel. After doing this, it again checks the applicability of the reaction rules, and based on the applicability information obtained re-evaluates the temporal order in which the currently applicable reaction rules should be processed. The region processing unit continues in this way until no reaction rules are applicable.

It is desirable to implement the dynamic determination of the partially time-ordered set of executable reaction rules in as few clock cycles as possible. In our current implementation, the number of clock cycles required to perform this task is 0. See Section 4 for details about our implementation.

Object production phase

In the object production phase, a region processing unit (a) updates the multiplicities of the object types in its region and attempts to send objects to and receive objects from the other regions, and then (b) updates the multiplicities of the object types in its region based on the objects it has received from other regions. All

Execution order	Reaction rules		
1	R_{11}	R_{21}	R_{31}
2	R_{12}	R_{22}	R_{32}
3	R_{13}	R_{23}	R_{33}
4	R_{14}	R_{24}	R_{34}
5	R_{15}	R_{25}	

Execution order	Reaction rules		
1	**R_{11}:a**	R_{21}:na	R_{31}:na
2	R_{12}:na	R_{22}:na	**R_{32}:a**
3	R_{13}:a	R_{23}:na	R_{33}:na
4	R_{14}:na	R_{24}:na	R_{34}:a
5	R_{15}:a	**R_{25}:a**	

Table 1. An illustration of how a region processing unit determines the order in which to process reaction rules in the object assignment phase. The region processing unit begins with a preliminary order determined at compile-time, as shown in the table on the left. At the start of the object assignment phase, the region processing unit checks the applicability of the reaction rules. The results of the applicability check are shown in the table on the right (applicable reaction rules are labelled 'a', and inapplicable reaction rules 'na'). The region processing unit then updates the processing order by removing the inapplicable reaction rules from consideration. The reaction rules that the region processing unit processes immediately after the first applicability check are shown in boldface.

of the updating and communication tasks are accomplished in a massively parallel manner.

To resolve resource conflicts that may occur in the object production phase (i.e., situations in which the multiplicity of an object type is to be updated by more than one parallel process), the region-oriented design includes two resource conflict resolution strategies: the *space-oriented strategy* and the *time-oriented strategy*. These strategies are similar to those adopted in the rule-oriented hardware design (see [8] and [9]). In the space-oriented strategy, copy registers are created for those object types whose multiplicities are to be updated by more than one parallel process, and the relevant parallel processing units store the updated multiplicity values in their assigned copy registers. The time-oriented strategy involves interleaving the operations of distinct parallel processes so that update operations which would conflict if executed in the same clock cycle are executed in different clock cycles.

The space-oriented strategy is implemented in basically the same way in both the rule-oriented and region-oriented hardware designs, with a couple of differences. The first difference is that, whereas in the rule-oriented design a special *multiset replication coordinator* processing unit needs to be introduced to coordinate the values stored in the copy registers, in the region-oriented design this coordination task can be performed by the already introduced region processing unit. The second difference is that in the region-oriented design copy registers do not need to be introduced for processing units sending objects to the region from other regions, because the already introduced register dedicated to the storage of the data received over the relevant communication channel can be used as a copy register. As in the rule-oriented design, in the region-oriented design, when the space-oriented strategy is used, the object production phase takes two clock cycles to complete. In the first clock cycle, register updates implementing the production of objects by reaction rules local to the region are performed. In the second clock

cycle, the values stored in the various registers representing multiplicity values of object types (including registers associated with channels) are coordinated, and the new multiplicity values for the object types in the region are stored in the original registers storing such values.

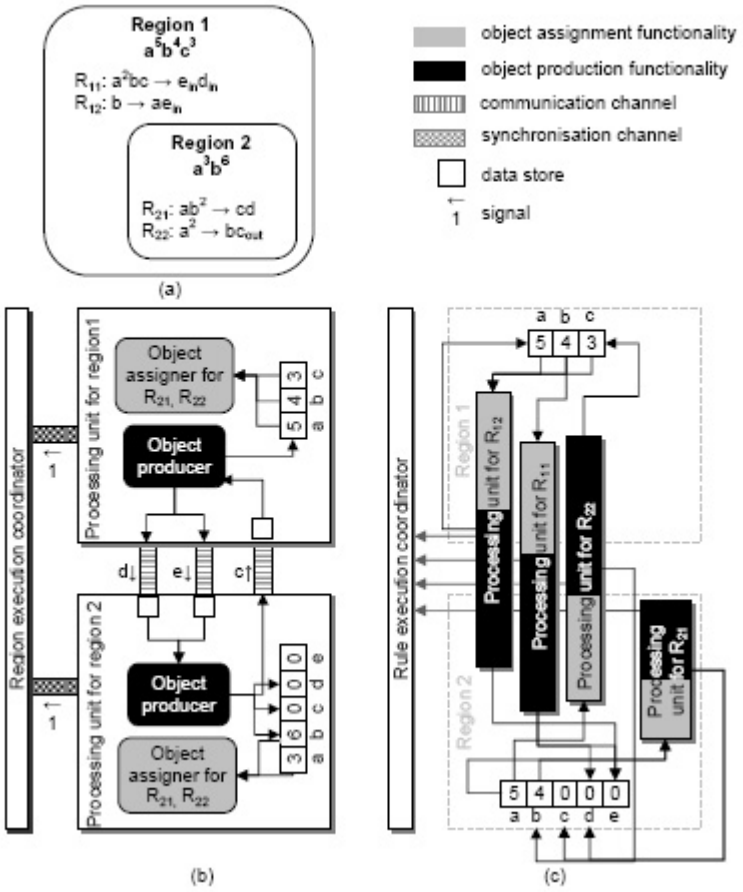


Fig. 2. An illustration of the region-oriented hardware design in comparison to the rule-oriented hardware design for a sample P system. (a) The sample P system. (b) The region-oriented design for the sample P system in which regions are implemented as processing units that communicate via channels. (c) The rule-oriented design for the sample P system in which reaction rules are implemented as processing units.

The time-oriented strategy is implemented differently in the two designs. In the rule-oriented design, the way in which updates are interleaved over time can be completely determined at compile-time, and so can be hard-coded into the source code defining the hardware circuit (see [8] and [9]). In the region-oriented design,

a distinction is drawn between *internal objects* and *external objects* for a region in a particular transition. The internal objects of a region in a transition are those objects produced in the transition by one of the reaction rules associated with the region. Objects sent to the region during the transition from other regions are external to the region. While it is possible to determine at compile-time the appropriate interleaving for updating operations occasioned solely by the production of internal objects, the interleaving for updating operations occasioned wholly or partly by the receipt of external objects must be determined at run-time. This is because, to preserve the independence of region processing units, information about when it might receive external objects is unavailable to the relevant region processing unit. To accomplish the run-time determination of the interleaving, an approach based on the use of semaphores is used.

3.3 Synchronisation

It is necessary to synchronise the execution of the object assignment phases of distinct region processing units. Without such synchronisation, it would be possible for objects produced in the object production phase for one region to be sent to another region still in its object assignment phase, thereby improperly interfering with the results of the object assignment phase in that region.

Unlike in the rule-oriented design, where synchronisation of the object assignment phases of reaction rules across regions is implemented explicitly using signals and flags, the synchronisation of the execution of the object assignment phases of distinct region processing units in the region-oriented design is implemented in a more implicit manner by having region processing units communicate over channels.

Channels are also used to perform explicit synchronisation at the end of each transition. The region-oriented design includes a *system coordination processing unit* which is responsible for coordinating the execution of the region processing units so that the transition-by-transition evolution of the P system can be realised. The system coordination processing unit is connected to each of the region processing units via dedicated synchronisation channels. Once a region processing unit has completed all of its tasks for a particular transition, it sends a signal down its synchronisation channel. Once the system coordination processing unit has received a signal from every region processing unit, it triggers a new transition.

One potential problem associated with the use of channels to implement the movement of objects between regions is the occurrence of deadlock. Handel-C channels operate in a synchronous manner. That is, once a pair of processing units have started engaging in a communication, neither the sending nor the receiving processing unit can move on to perform other tasks until the communication has been accomplished. Consequently, unless the operations of sending and receiving objects among region processing units are conducted in an appropriate order, deadlock can occur. To prevent deadlock from occurring we ensure that the channel communications for different regions are carried out in distinct parallel branches of execution.

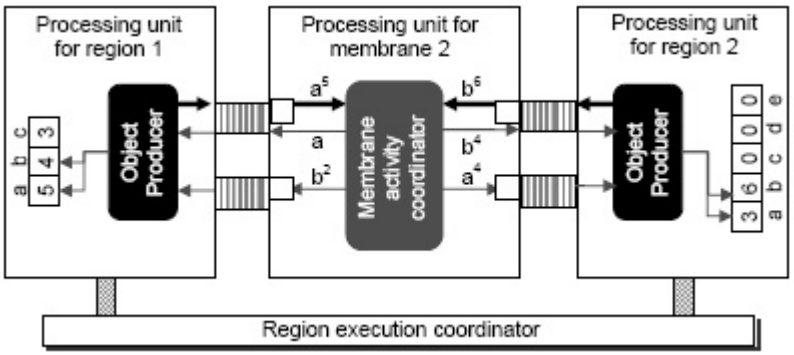


Fig. 3. An illustration of the implementation of a P system with an antiport rule (aa, in; b, out) using the region-oriented design. In this example, two instances of the antiport rule are executed.

3.4 Extensibility of the design

The region-oriented hardware design makes it possible to implement P systems with features that require the explicit presence of membranes in an intuitive way. In particular, each membrane can be implemented as a processing unit associated with two region processing units (corresponding to the inner and outer regions of the membrane) (see Figure 3). Such a membrane processing unit could, for example, mediate the exchange of objects between regions effected by antiport rules. For an antiport rule to be applicable, enough objects of the right types need to be available in both regions. As each of the two region processing units for the two regions do not know the multiset of objects available in the other region, it is not possible for the region processing units to implement the antiport rule on their own — a membrane processing unit is also required. Nevertheless, it is still possible for the membrane processing unit to remain quite independent from the two region processing units. For example, the region-oriented hardware design could be augmented to implement antiport rules as follows. The region processing units for the inner and outer regions send objects to a membrane processing unit. The membrane processing unit attempts to couple objects in the way specified by the antiport rule, and sends coupled objects to their destination regions and returns uncoupled objects to their regions of origin. In this way, not only do the region processing units not know about each other's multiset of objects, but the membrane processing unit does not need to know this information either. It is sufficient that both region processing units know about the existence of the membrane processing unit.

4 Implementing regions in hardware

In this section, we describe how the regions of a P system are implemented using Handel-C when the region-oriented hardware design is adopted.

4.1 Atomic operations associated with the application of reaction rules

From one perspective, the overall behaviour of a P system emerges from the application of reaction rules. At the implementation level, the execution of a single application of a reaction rule involves the execution of a certain number of instances of each of a set of logically atomic operations:

Rule execution = (pDIV, qMIN, rMUL, sSUB, tCOM, uADD), where

$p = 0$ or 1, q, r, s, t, $u \geq 0$, DIV denotes the operation of dividing the multiplicity of the objects of a given type available in the region by the number of objects of that type required for the application of one instance of the reaction rule, MIN denotes the operation of computing the maximum number of instances of the reaction rule that can be applied in the current transition, MUL denotes the operation of computing the number of objects of a particular object type to be consumed/produced by the reaction rule in the current transition, SUB denotes the operation of reducing the multiplicity of a particular object type available in the region (by a certain amount), COM denotes the operation of sending (or attempting to send) a certain number of objects of a particular type to a particular region, and ADD denotes the operation of increasing the multiplicity of a particular object type available in the region (by a certain amount).

In Reconfig-P, each of the above operations is realised as an atomic operation. These atomic operations are the building blocks for the construction of any particular hardware circuit. The names of the operations reflect the main computational operations involved in their implementation. Mapping the atomic operations onto a hardware circuit requires making decisions about their temporal granularity. At fine granularity, an operation is performed over multiple clock cycles and therefore needs to be decomposed into suboperations. At coarse granularity, an operation is performed in one clock cycle. Although assigning a logically atomic operation a fine granularity at implementation results in a greater number of clock cycles, it often reduces logic depth, and therefore can lead to an increased system clock rate.

To determine the appropriate degree of granularity for a given logically atomic operation, it is necessary to examine the implementation characteristics of the operation in terms of hardware resource consumption and logic depth. Multiplication and division can generate complicated combinatorial circuits and therefore in general are expensive to implement in one clock cycle. However, in the specific case of the execution of a P system, in both multiplication and division operations one of the operands is a constant. This significantly reduces the logic depth of

the combinatorial circuits that implement the operations[3]. Addition and subtraction are relatively inexpensive operations and, based on the performance results for the current version of Reconfig-P (reported in [8]), do not compromise the performance of the hardware circuit. Given these considerations, in the hardware implementation the default scenario is that each of the logically atomic operations is performed in one clock cycle. However, to accommodate situations in which a large number of processing units is required and therefore the system clock rate would otherwise be compromised significantly, P Builder has the ability to generate the hardware circuit in such a way that the logically atomic operations are performed over several clock cycles.

4.2 Implementations of the logically atomic operations

We now describe how we have implemented the logically atomic operations identified in the previous section in hardware.

DIV and MUL The DIV and MUL operations are implemented in a similar way. The obvious implementation approach is to devote a separate piece of hardware to the execution of each of the operations for each reaction rule. However, this approach would lead to unnecessary duplication of hardware resources because it is often the case that different reaction rules consume/produce the same number of objects for an object type (i.e., the multiplicity of the consumed/produced object type is the same in the definitions for the reaction rules). Duplication of hardware resources can be particularly problematic when Handel-C is used as the specification language, since the Handel-C compiler generates distinct pieces of hardware for the same division or multiplication operation if this operation occurs in different places in the source code. Our solution to the problem of unnecessary hardware duplication is to have distinct DIV/MUL operations which share the same operands implemented as a single processing unit, and for the collection of all such processing units to be implemented as a pool of servers. The DIV and MUL servers continuously perform their respective division/multiplication operations. They execute their operation in one clock cycle, and then store the result in an output register. Each of the servers has direct access to the data for both operands for its operation, and so operates totally independently from its clients. A client processing unit that needs to evaluate one of the relevant divisions or multiplications is required to invoke the appropriate server (but is not required to supply any input to the server). It first waits for one clock cycle (to allow the server to execute its operation with the current values for the operands), and then reads the appropriate output register to obtain the result. As there is one division pool and one multiplication pool per region (rather than for the P system as a

[3] The Xilinx Virtex-II FPGA used in the implementation contains hardware multipliers that allow efficient and high-performance implementation of multiplication operations [13]. However, where one of the operands is a contant, multiplications can be more efficiently implemented on slices using either bitshifts or constant coefficient multipliers.

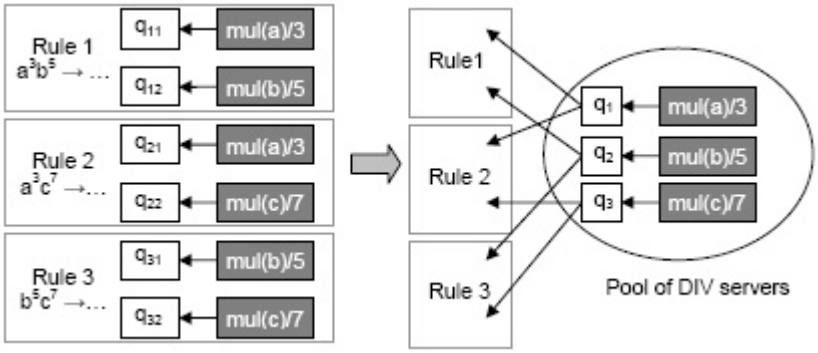


Fig. 4. An illustration of the implementation of a pool of DIV servers.

whole), our implementation approach does not cause routing problems. Figure 4 shows an example of a pool of DIV servers.

MIN For reasons similar to those described above, a pool of processing units is used to perform MIN operations. By default, each MIN operation is implemented as a hard-coded Handel-C macro expression which executes in one clock cycle. However, as the logic depth of a MIN operation is linearly proportional to the numbers of object types consumed by the reaction rules in the region, a MIN operation can easily be subjected to logic depth reduction.

SUB As reaction rules with relative priorities are not processed simultaneously, there are two implementations of the SUB operation: SUB_M and SUB_F. SUB_M is used for those reaction rules that are unrelated by priority to any other reaction rule. It is implemented in Handel-C as a macro expression, which corresponds to a single unshareable piece of hardware with both operands hard-coded. SUB_F is used for reaction rules with relative priorities. It is implemented as a Handel-C function, which corresponds to a single shareable piece of hardware. Since in general the subtractions performed by different reaction rules have different operands, to make SUB_F processing units shareable among the processes implementing the application of reaction rules, the implementation of a SUB_F processing unit operates at the level of object types rather than at the level of reaction rules. More specifically, there is a SUB_F processing unit for each object type.

ADD All ADD operations (which are used in the implementation of the production of objects by reaction rules, a process which is not subject to any temporal constraints) can in principle be executed simultaneously. However, unless appropriate precautions are taken, the parallel execution of ADD operations can result in parallel processes attempting to update the same register at the same time. There are two main strategies for the avoidance of such update conflicts: the time-oriented strategy and the space-oriented strategy (see Section 3.2).

When the space-oriented strategy is used, for each copy register there is one ADD_M processing unit responsible for updating that register. Each ADD_M process-

ing unit is implemented as a Handel-C macro. This allows all updating operations in the object production phase to be completed in one clock cycle.

When the time-oriented strategy is used, there are three types of processing units implemented. The first type, called $\mathrm{ADD_M}$, is used to update the multiplicity value for an object type with no conflicts. The second type, called $\mathrm{ADD_F}$, is used to update the multiplicity value for a local object type with conflicts. The third type, called $\mathrm{ADD_S}$, is used to update the multiplicity value for an external object type. $\mathrm{ADD_S}$ implements semaphore-based interleaving using the `trysema` and `releasesema` constructs provided by Handel-C.

COM COM operations, which apply only to the region-oriented design, are implemented using channels (see Section 3). Whenever it is possible for objects to move from one region to another region, the implementation includes a channel connecting the region processing unit for the source region to the region processing unit for the destination region. There are various ways in which one could implement inter-region communication using channels. The approach one takes influences the number of channels required, as well as the amount of processing needed to complete sending and receiving operations. We will now briefly discuss three possible implementation methods.

In the first method, for every reaction rule r in a region x that sends objects (of any type) to a region y, there is exactly one channel connecting the region processing units for x and y. This channel is used only for the distribution of objects produced by r. Therefore, the data sent over the channel must allow the region processing unit for y to determine which object types are being sent and how many of each type are being sent. To avoid making the definition of r available to the region processing unit for y (and thereby compromising the independence of this region processing unit), this could be achieved by having the region processing unit for x send an n-tuple over the channel, where n is the number of object types found in the whole P system (not only those produced by r destined for y), which contains for each object type found in the whole P system the multiplicity of that object type being sent to y. Upon receiving the n-tuple, the region processing unit for y would proceed to update the multiset array for y. Obviously, if there are multiple reaction rules in x that produce objects destined for y, there will be multiple channels between the region processing units for x and y. The region processing unit for y would need to coordinate the data received over these channels, as it would receive data relating to the same object type on different channels.

As it is possible to determine at compile-time which types of objects might be produced by which reaction rules and sent to which regions, it is possible to hard-code the relevant pieces of this information in the implementation of the receiving region processing unit. The second method of implementing inter-region communication illustrates this possibility. To implement this method, we would need to relax (albeit to a minimal extent) our requirement that region processing units be independent of each other. In the method, for each reaction rule r in a region x and for each object type o produced by r to be sent to a region y, there is exactly one channel connecting the region processing units for x and y. This

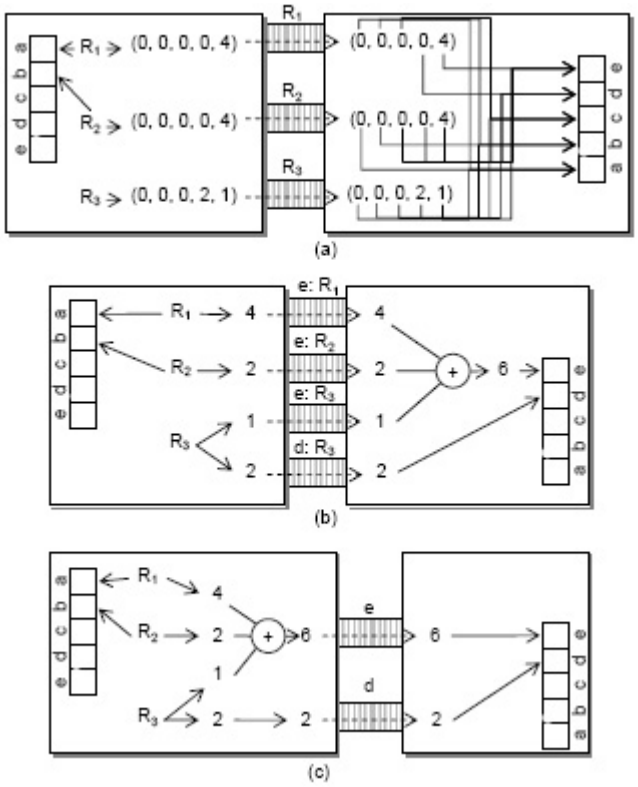


Fig. 5. An illustration of different strategies of implementing inter-region communication using channels. Diagram (a) illustrates the first method mentioned in the text, diagram (b) illustrates the second method, and diagram (c) illustrates the third method.

channel is used only for the distribution of objects of type o produced by r destined for y. Assume that the region processing unit for y has access to information about which channel is associated with which object type. Then the region processing unit for x needs to send only the multiplicity value for o (i.e., the number of objects of type o that are to be sent in the current transition) down the channel. As in the first method, because the region processing unit for y might receive objects of the same type on different channels, it needs to coordinate the data received over the different channels before proceeding to update the multiset array for y.

In the third method, for every object type that might be produced in a region x and sent to a region y, there is exactly one channel between the region processing units for x and y. Again assume that the region processing unit for y knows which channel is associated with which object type. In this scenario, the region processing unit for x needs to evaluate for each object type the total number of objects of that type to send before engaging in the relevant channel communication. Once it has done this, it sends a single value down the channel. The region processing

unit for y simply stores this value in the appropriate register of the multiset array for y.

Figure 5 illustrates the three methods of implementing inter-region communication described above.

We now discuss the relative merits of the three methods of implementing inter-region communication. As regards faithfulness to the biological inspiration of P systems, we rank the third method highest. The first method is perhaps the least in keeping with the biological inspiration of P systems. If we regard the channels in the implementation as representations of cellular transport mechanisms (such as ion channels and osmosis), and reaction rules as representations of chemical reactions, then according to the first method each cellular transport mechanism facilitates the transportation of only the products of a single chemical reaction. In the general case, this is biologically unrealistic. The second method is also quite removed from the biological inspiration of P systems in that cellular transport mechanisms would again be regarded as facilitating the transportation of only the products of a specific chemical reaction. The third method is the most biologically realistic because, in this method, cellular transport mechanisms would be regarded as facilitating the transportation of single types of chemicals (such as potassium ions), as is commonly found in biological cells. As regards the extent to which the independence of region processing units is preserved, the first method ranks highest, with the second and third methods being roughly equivalent. Even so, neither the adoption of the second method nor the adoption of the third method would result in a significant reduction in the independence of region processing units. This is because in these methods the information a region processing unit possesses about other region processing units is available only in an implicit sense. The information is embedded into the very structure of the region processing unit, and so the region processing unit does not explicitly refer to this information when carrying out its operations. As regards efficiency, the third method ranks highest, both in terms of the number of channels used and the amount of processing required. Based on the considerations just outlined, we decided to adopt the third method when implementing the region-oriented design.

4.3 Linking and synchronisation

In the previous section, we described the hardware components that implement the logically atomic operations. To realise operations occurring at the level of reaction rules, at the level of regions, or at the level of the entire P system, it is necessary to link and synchronise the execution of these basic components. In this section, we describe how the components are linked and synchronised to accomplish some of the processing performed by a region processing unit. We have chosen to focus on this particular case because it is fundamentally important to the region-oriented design.

Figure 6 shows a high-level UML activity diagram for the object assignment phase of the execution of a region processing unit (see Section 3.2 for a description

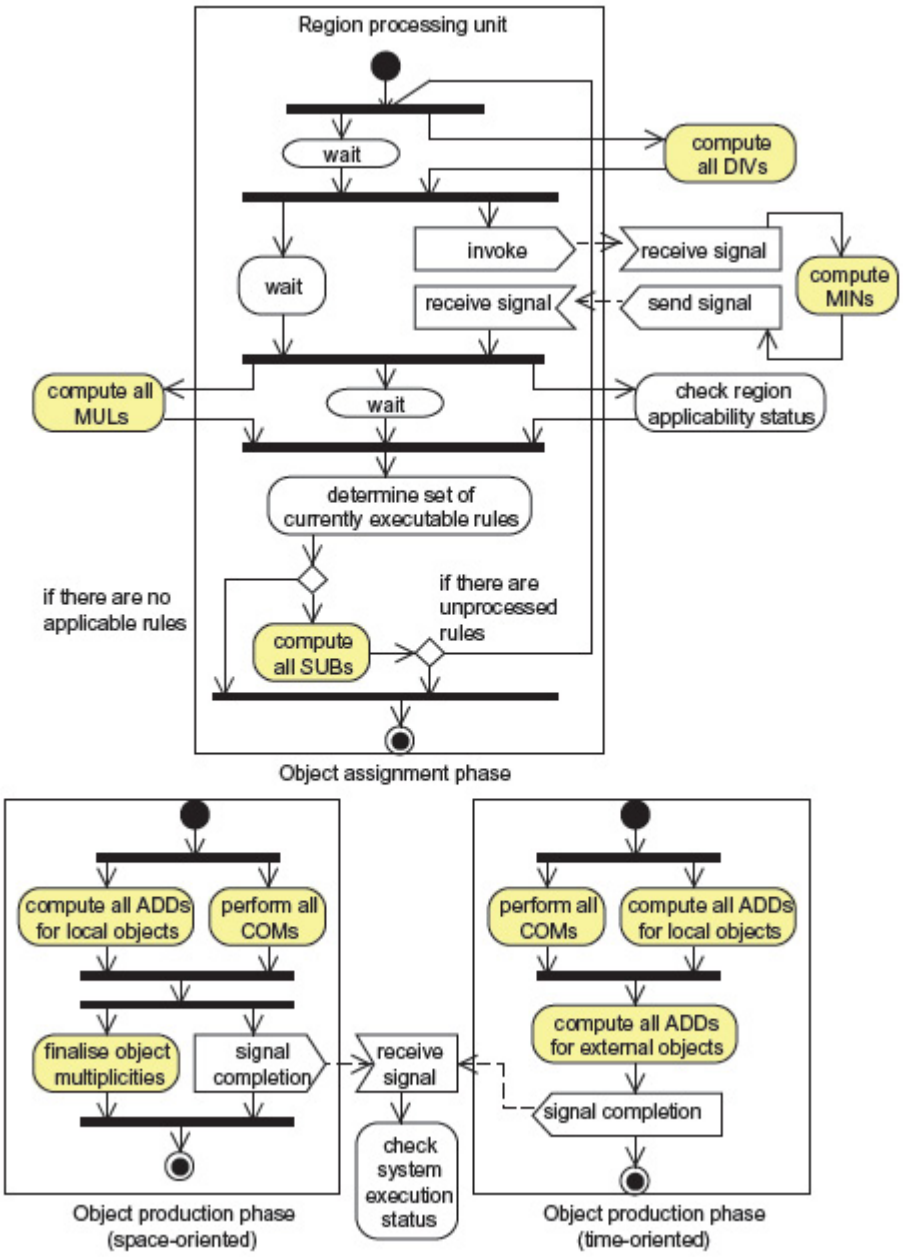


Fig. 6. A UML activity diagram presenting high-level views of the implementations of the object assignment and object production phases of a region processing unit.

of this phase). Hardware components for logically atomic operations (described in Section 4.2) are represented as shaded boxes in the diagram. This section contains a description of how the other aspects of the diagram — the control flow, linking and synchronisation represented by arrows, solid bars and unshaded boxes — are implemented in hardware.

Linking and synchronisation within a region processing unit

To implement the simple internal control flow within a region processing unit, we use the basic control constructs provided by Handel-C. For example, the arrows in the activity diagram shown in Figure 6 are implemented using the `seq` construct, the solid bars are implemented as `par` constructs, and diamonds are implemented using conditional constructs such as `if`.

Linking and synchronisation between a region processing unit and external processing units

In the implementation of the object assignment phase of a region processing unit, it is necessary to link the region processing unit with processing units implementing the logically atomic operations DIV, MIN, MUL, SUB and ADD, and to synchronise the execution of the region processing unit with these other processing units.

In our implementation, processing units may be categorised according to whether they execute constantly without invocation or execute only when invoked. Among the processing units that execute constantly are the processing units implementing the DIV and MUL operations as well as a processing unit responsible for checking whether at least one reaction rule in the region is applicable (see below). Due to the continuous execution of these processing units, when a region processing unit uses one of these processing units, it needs to read the register in which the processing unit stores the result of its computation. However, to ensure that it reads the result applicable to the current transition, the region processing unit must wait for the currently applicable data to be stored in the register. This can be done by inserting the appropriate number of `delay` statements in the relevant section of the Handel-C code implementing the region processing unit or, preferably, by having the region processing unit perform other processing during the clock cycles over which the external processing unit is performing the currently applicable computation. As for the processing units that must be invoked, a region processing unit can invoke these processing units efficiently by using a set of signals and flags as follows:

```
//Processing unit 1:
 while(1) {
   signal = 1; // clock cycle x
   ...
}

//Processing unit 2:
 while(1) {
   par{
      flag = signal; //clock cycle x
      if(flag == 1) {
         ... //clock cycle x+1
      } else
         delay;
    }
}
```

Checking the region applicability status

Our implementation of the region-oriented design includes for each region processing unit a processing unit which is responsible for checking whether at least one reaction rule in the relevant region is applicable. This processing unit is used by the region processing unit for the purpose of preemptive termination. After the region processing unit calculates the maximum number of instances of a reaction rule, it immediately records the applicability status of the reaction rule in a 1-bit register. The external processing unit reads the applicability registers for all the reaction rules in the region, computes whether there is at least one applicable reaction rule, and then writes the result to a 1-bit output register. Therefore there is a single `delay` statement in the Handel-C code implementing the region processing unit just before the code implementing the reading of the output register.

Reporting completion

As mentioned in Section 3.3, our implementation of the region-oriented design includes a system execution coordinator processing unit, which is responsible for checking whether all region processing units have completed their executions for the current transition, and triggering a new system transition when this condition is satisfied. In the implementation, once it has completed its operations for the current transition, a region processing unit signals this fact to the system execution coordinator via a synchronisation channel (see Figure 2). The following Handel-C code shows how this is achieved in the case where there are two region processing units.

```
//Region processing unit 1
 while(1) {
   ...
   synChan1 !1; //report completion
}
//Region processing unit 2
 while(1) {
   ...
   synChan2 !1; //report completion
 }
//System execution coordinator
 while(1) {
   par {
      synChan1 ? temp1; //receive completion signal on first channel
      synChan2 ? temp2; //receive completion signal on second channel
    }
   //trigger new transition
 }
```

Determining which reaction rules are applicable

As discussed in Section 3.2, when processing reaction rules in the object assignment phase, it is advantageous for a region processing unit to check the applicability of the reaction rules. If a reaction rule is inapplicable, it need not be processed further. The implementation approach for the applicability checking operation that most readily comes to mind is the use of `if` and `else` constructs. However, because the Handel-C compiler enforces an `else` implementation with every `if` implementation, unless one is willing to spread the operation over multiple clock cycles, this approach will in general result in a deeply nested if-else construction. This problem does not arise if `goto` statements are used instead of `else` constructs. Consequently, in our implementation we use `goto` statements. Such statements are inserted just before the code implementing the processing of a reaction rule, and allow this code to be skipped. The combination of `goto` statements results in a hardware state machine which allows the region processing unit to process the reaction rules in the most time-efficient manner. Specifically, if it is found that a reaction rule is inapplicable, the control will jump to the part of the code for the reaction rule with the highest priority out of all the remaining reaction rules. Because `if` and `goto` statements take zero clock cycles to execute, no clock cycles are wasted in determining which reaction rule should be processed next. Taking this implementation approach allows the various reaction rules to be processed in a consistent manner, and therefore greatly simplifies the control flow required for the processing of the reaction rules.

5 Design and implementation of a new version of P Builder

The existing version of Reconfig-P implements only the rule-oriented hardware design. In the previous sections, we have described a new design, the region-oriented design. The region-oriented design has several attractive features, such as its faithfulness to the intuitive conceptual understanding of a P system and its modularity. Nevertheless, the rule-oriented design has features which make it preferable to the region-oriented design in many scenarios. For example, since the adoption of the rule-oriented design can result in a higher system clock rate, a user of Reconfig-P might prefer to use the rule-oriented design when high performance is a priority. Given that the rule-oriented design and region-oriented design have different strengths and weaknesses, it is desirable to include both in Reconfig-P. That way, the decision about which design is most suited to the problem at hand can be left to the user or made based on an analysis of the characteristics of the input P system.

P Builder, implemented in Java, is the component of Reconfig-P responsible for generating customised Handel-C source code for the input P system. When developing the new version of Reconfig-P, we re-engineered P Builder so that it can accommodate *both* the rule-oriented and region-oriented designs. When re-engineering P Builder, we aimed at promoting its maintainability and extensibility through the use of appropriate software engineering design patterns. In this section, we explain the design and implementation of the new version of P Builder.

5.1 Requirements for the new version of P Builder

When determining the hardware specification for the input P system, P Builder aims to maximise performance and minimise hardware resource consumption. The existing version of P Builder achieves this aim, as evidenced by the fact that the existing version of Reconfig-P delivers a good balance between performance, flexibility and scalability as a computing platform for membrane computing applications. Performance refers to the extent to which the system can execute P systems in a time-efficient manner. Flexibility refers to the extent to which the system can support the execution of a variety of classes of P systems. And scalability refers to the extent to which the system can support the execution of large P systems. Our primary purpose in re-engineering P Builder was to improve the flexibility of Reconfig-P, while not too significantly compromising its existing levels of performance and scalability. More specifically, our primary purpose was to broaden the range of implementation approaches according to which P system models can be realised as hardware circuits, and to develop a sophisticated object-oriented design that promotes the maintainability and especially the extensibility of Reconfig-P, where extensibility refers to the extent to which the system can readily be augmented to support additional P system models and additional implementation approaches.

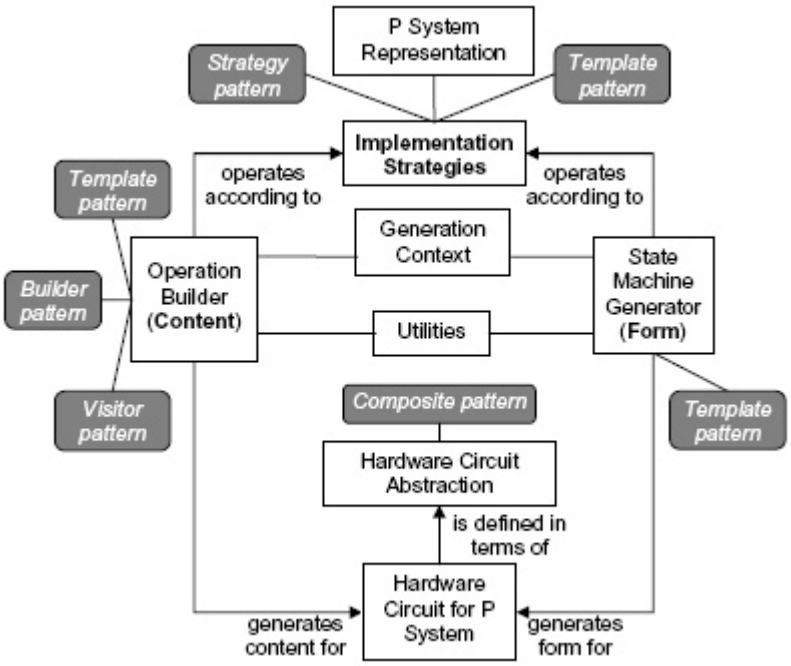


Fig. 7. The high-level architecture of the new version of P Builder, which was developed according to the Content-Form-Strategy design pattern.

5.2 Design methodology

In the design of the new version of P Builder, our guiding design principle was that of *separation*. First, we viewed the hardware implementation for a P system as a complex of form and content, and attempted to treat the formal aspects of this complex in isolation from its content. Second, we attempted to cleanly separate the different functions performed by P Builder. We achieved the first type of separation through the use of a novel design pattern, and achieved the second type by allocating different functions to different modules. We now briefly discuss these separation strategies.

The Content-Form-Strategy design pattern

The basic problem that P Builder is intended to solve is the generation of Handel-C source code for a hardware circuit which implements an input P system according to one of a variety of alternative implementation strategies (for example, with a rule-oriented design and space-oriented resource conflict resolution). This problem can be viewed as an instance of a more general problem: that of producing an algorithm for the solution of a problem, where this algorithm must be constructed

according to one of a variety of possible implementation strategies. An implementation strategy does not affect the logical characteristics of an algorithm, but only its implementation characteristics (such as performance or memory usage). When constructing such an algorithm, it is beneficial to separate as much as possible the logical characteristics of the algorithm from its implementation characteristics. Not only does this make the algorithm easier to understand, it also facilitates the use of new implementation strategies in the future. However, it is often quite difficult to achieve a clean separation of the logical and implementation aspects of an algorithm. Our novel design pattern, which we call Content-Form-Strategy, prescribes a general solution to the general problem just outlined.

A key idea of the Content-Form-Strategy pattern is that an algorithm may be viewed as a complex of form and content, where the units of content are logically atomic computational operations and the form is the way in which these units of content relate to each other logically and temporally. If an algorithm is viewed as a flowchart such as that shown in Figure 6, then the shaded boxes in the flowchart comprise the content of the algorithm, and the diamonds, arrows, bars and unshaded boxes comprise the form of the algorithm. Note that computational operations that are included in the algorithm solely for the purpose of linking and synchronising other computational operations are regarded as part of the form of the algorithm (they would be represented as unshaded boxes in a flowchart). A strategy for the construction of an algorithm influences both the content and form of the algorithm. That is, different strategies may require the inclusion of different logically atomic computational operations, and will necessitate different logical and temporal relationships between these operations.

The solution prescribed by the Content-Form-Strategy design pattern consists of nine steps:

1. Define an abstract model of an algorithm as expressed in the desired implementation language.
2. For each implementation strategy, identify the logically atomic computational operations in terms of which the algorithm to be constructed can be defined.
3. Express the logically atomic operations identified in step 2 in terms of the elements of the abstract model of an algorithm defined in step 1.
4. For each implementation strategy, and for each of the logically atomic operations identified in step 2, determine (a) the preprocessing operations and postprocessing operations (if any) for the execution of the operation, (b) the data writing (if any) performed by the operation, and (c) the temporal relationship of the operation with all the other logically atomic operations.
5. Express the preprocessing operations, postprocessing operations, data writing and temporal relationships determined in step 4 in terms of the elements of the abstract model of an algorithm.
6. Based on the results of steps 2 and 4, identify (a) the logically atomic computational operations that apply to all implementation strategies, and (b) invariant preprocessing operations, postprocessing operations, data writing and temporal relationships (i.e., those preprocessing operations, postprocessing op-

erations, data writing and temporal relationships that obtain regardless of the implementation strategy).
7. Based on the result of step 6, define a template algorithm which specifies the features common to all possible algorithms for all implementation strategies in terms of the elements of the abstract model of an algorithm.
8. For each implementation strategy, define an algorithm for the filling out of the template algorithm defined in step 7 in terms of the elements of the abstract model of an algorithm.
9. Express each of the algorithms defined in step 8 in the desired implementation language.

Table 2 illustrates how the Content-Form-Strategy design pattern applies to the specific problem of generating Handel-C source code for a circuit that implements an input P system according to one of a variety of alternative implementation approaches.

Modularisation

The new version of P Builder has been designed as a set of modules. Components within a module are relatively tightly coupled, whereas components in different modules are relatively loosely coupled. Each module has a well-defined interface which indicates the high-level functions performed by the components in the module. Via this interface, components in other modules can make use of these functions. Because of the relatively loose coupling between modules, future modifications to a module will usually not necessitate changes in other modules. Clearly, this improves the maintainability of P Builder.

5.3 Overview of the design and implementation

Figure 7 shows the high-level architecture of the new version of P Builder. This architecture was designed according to the Content-Form-Strategy design pattern. The UML class diagram shown in Figure 10 indicates how some elements of the architecture are implemented as classes.

The main modules in P Builder are P System Representation, Strategies, Operation Builder, System Utilities, State Machine and Hardware Circuit Abstraction. The modules have been designed and implemented with the aid of object-oriented design patterns [4], which prescribe thoroughly tested and effective solutions to design problems, and therefore enable the creation of flexible, elegant and reusable object-oriented designs[4]. We now briefly describe the modules.

[4] For more information about object-oriented design patterns, we refer the reader to [4], the classic reference in the field.

P System Representation

The P System Representation module provides an object-oriented representation of a P system. It specifies classes of entities (such as `Region`, `Rule` and `ObjectType`), attributes of these classes (such as the `multiplicity` attribute of the `ObjectType` class), and relationships that hold between classes (such as the reflexive one-to-many relationship of containment that holds for the `Region` class). The main purpose of this module is to allow P Builder to represent the input P system as an object.

Hardware Circuit Abstraction

The Hardware Circuit Abstraction module provides an abstract representation of a Handel-C specification of a hardware circuit.

A hardware circuit may be regarded as a complex processing unit composed of simpler processing units. The most elegant and efficient way to represent such a compositional structure of processing units in an object-oriented system is to use of the Composite design pattern [4]. This pattern prescribes a way of representing part-whole hierarchies using tree structures of objects. The advantage of using the Composite pattern is that it allows atomic objects (individual objects) and composite objects (trees of objects) to be treated uniformly. Therefore in P Builder we represent a Handel-C specification of a hardware circuit as a tree of processing units. There are two types of processing units: *parallel processing units* and *sequential processing units*. A processing unit may contain other processing units. If a sequential processing unit contains other processing units, then these other processing units are to be executed sequentially. If a parallel processing unit contains other processing units, then these other processing units are to be executed in parallel. A processing unit which is not composed of other processing units is called an *atomic processing unit*. For the sake of neatness, we regard an atomic processing unit as a parallel processing unit. Each atomic processing unit is associated with the specification of an operation, which we call a *statement*, which executes in the smallest possible time interval. Atomic processing units correspond to Handel-C statements, which execute in one clock cycle. The root node of the tree of processing units, which is called the *root processing unit*, represents the full execution of the hardware circuit. It corresponds to the `main` function in the Handel-C program for the circuit. In the region-oriented design, the region processing units are immediate children of the root processing unit, whereas in the rule-oriented design the immediate children of the root processing unit include the rule processing units. The leaf nodes of the tree are all atomic processing units. To allow for the representation of control flow, every processing unit begins with a preprocessing phase and ends with a postprocessing phase. Each type of phase consists of a sequence of zero or more operations. Such an operation might be the checking of a condition (for example, the condition for a `while` loop), the execution of a single statement (for example, the storage of data in a register), or the execution of a collection (block) of statements.

Aspect of Content-Form-Strategy pattern	**Example of how the aspect applies to the generation of a hardware circuit for a P system**
Algorithm to be generated	Handel-C program that specifies the hardware circuit for the input P system
Current implementation approaches	Rule-oriented/region-oriented design; space-oriented/time-oriented resource conflict resolution; deterministic/ nondeterministic execution
Possible future implementation approaches	Alternative algorithms for nondeterministic object distribution
Logically atomic computational operations	DIV, MIN, MUL, SUB, COM, ADD (see Section 4.2)
Preprocessing for the execution of a logically atomic operation	An ADD operation proceeds only if the corresponding reaction rule is applicable
Temporal relationship between logically atomic operations	A SUB operation should execute only after its associated MIN operation has executed
Operations included solely for the purpose of linking and synchronisation	Operations performed by system execution coordinator
Abstract model of an algorithm as expressed in the implementation language	Abstract model of a hardware circuit as expressed in a Handel-C program
Template algorithm	An algorithm that specifies the high-level stages of a transition (e.g., object assignment)
Filling out of the template algorithm	Inclusion of object assignment for reaction rules with priorities and object production when space-oriented strategy is used

Table 2. Illustration of how the Content-Form-Strategy design pattern applies to the problem of generating Handel-C source code for a circuit that implements an input P system according to one of a variety of implementation approaches.

The UML class diagram in Figure 9 explains how these ideas are represented in the implementation of P Builder. Figure 8 shows an example of Handel-C source code generated from a tree of processing units. The correspondence between the processing units and code sections is marked in the figure.

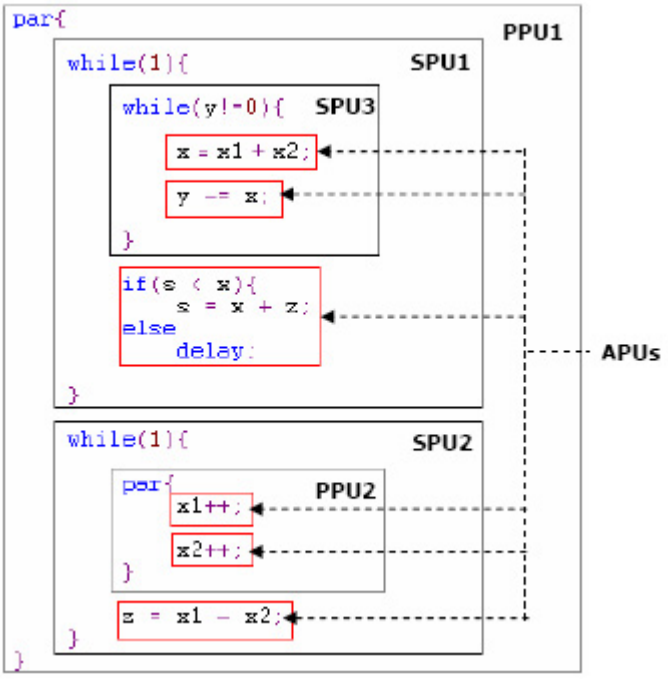


Fig. 8. Illustration of the correspondence between Handel-C source code and a tree of sequential processing units (SPUs), parallel processing units (PPUs) and atomic processing units (APUs).

General design for all modules

All the modules share the same basic structure. More specifically, each module contains four layers. The top layer is an interface which is exposed to the other modules. The rest of the layers constitute a hierarchy of classes. The second layer is a class that implements general operations for the module. When a client module requests that one of these operations be performed in a certain way, the class returns the specific subclass that implements the operation as requested. The third layer consists of classes responsible for implementing specific overall implementation strategies (the currently available overall implementation strategies are the rule-oriented and region-oriented strategies). The bottom layer consists of classes responsible for implementing resource conflict resolution strategies (the currently available resource conflict resolution strategies are the space-oriented and time-oriented strategies).

Strategies

The Strategies module contains classes that realise different overall strategies for the implementation of a P system on a hardware circuit. Currently there are

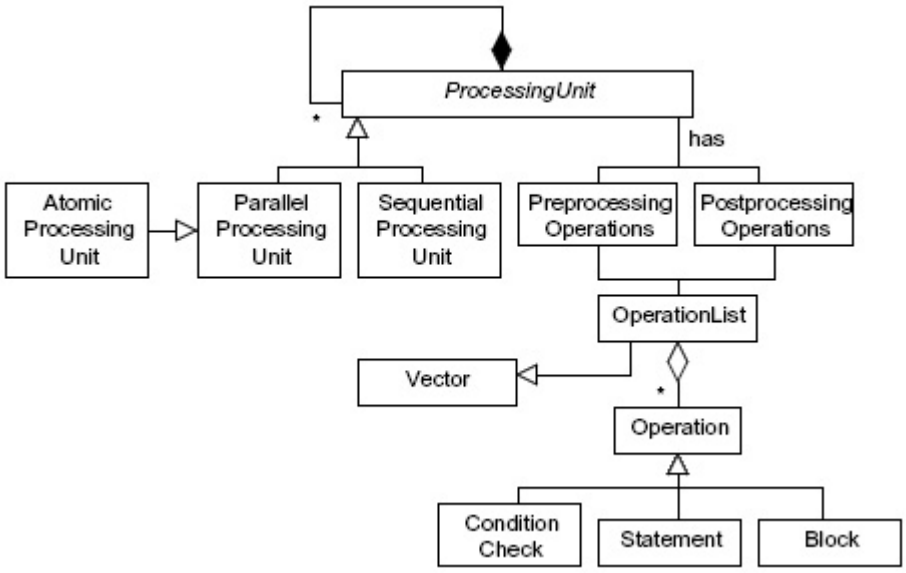


Fig. 9. The main classes of entities used in an abstract representation of a hardware circuit.

classes for the rule-oriented and region-oriented strategies. This module could be extended to include classes for other strategies, such as nondeterministic reaction rule application (see [7]).

GeneralStrategy

The top-level class of this module is `GeneralStrategy`. This class implements the features common to all strategies. Depending on the requirements of the client using the Strategies module, a specific strategy for the implementation of the input P system on a hardware circuit needs to be applied. To decouple the client from the various complex algorithms for the specific strategies, the Strategy design pattern [4] is used. This design pattern states that a family of algorithms solving the same problem in different ways should be defined in such a way that (a) the algorithm that is used to solve the problem for a client is selected dynamically based on the specific requirements of the client, and (b) the client does not need to know which particular algorithm is used. In our design, the algorithms for the different strategies are implemented in different subclasses of the `GeneralStrategy` class, which implements the `StrategyInterface` interface. When a client wishes for the strategy most appropriate to its requirements to be applied, it needs only to pass its requirements as parameters to an instance of a class called `StrategySelector`, which returns an instance of a subclass of `GeneralStrategy` which satisfies the requirements, and then invoke the `execute()` method of the returned instance. So

all the client needs to know is its own requirements and the interface specified in `StrategyInterface`.

One way of implementing different strategies for the hardware representation of the execution of a P system and for resource conflict resolution is to implement each specific algorithm in a separate class. However, taking such an approach would likely result in duplication of code, which is bad for maintainability. Hence a better implementation approach is required.

As discussed in Section 3.2, the high-level execution algorithm that underlies each of the different implementation strategies consists of an object assignment phase and an object overall production phase, each of which is defined in terms of a set of logically atomic operations. Distinct strategies for the hardware representation of the execution of a P system differ with respect to the way in which the logically atomic operations are put together to realise high-level operations. Therefore, we implement the high-level execution algorithm in the `GeneralStrategy` class, and implement the specialised versions of the execution algorithm for the different strategies in different subclasses of `GeneralStrategy`. This is achieved in an elegant way through the use of the Template design pattern [4]. By following this design pattern, we can enforce that specialised algorithms in the subclasses conform to the high-level algorithm, and make the implementation transparently reflect the logical characteristics of the execution of a P system. A template method in a superclass defines the skeleton of an algorithm, which can be filled out in different ways in different subclasses. That is, a subclass fills some or all of the placeholders in the template method in order to implement a more specialised algorithm. As shown in Figure 10, the algorithms defined by the `assignObjects()` and `produceObjects()` methods, which implement the object assignment and object production phases in terms of the logically atomic operations, are implemented as template methods. They define the sequence of logically atomic operations that needs to be executed to accomplish the processing for the relevant phase. Each method is declared as `final` in order to prevent subclasses from overriding the method (and therefore from being able to change the order in which the logically atomic operations are executed). To define specialised algorithms, one need only provide implementations of the logically atomic operations in the subclasses of the class containing the template method.

SpecificImplementationStrategy

A `SpecificImplementationStrategy` class implements the execution algorithm for a specific overall implementation strategy (i.e., currently either the rule-oriented strategy or region-oriented strategy). This can be achieved by (a) implementing the relevant atomic operations defined in the template methods for `assignObjects()` and `produceObjects()` so that the object assignment and object production phases take on the specific behaviours characteristic of the implementation strategy, and/or (b) implementing suitable wrapper code for the object assignment and object production phases.

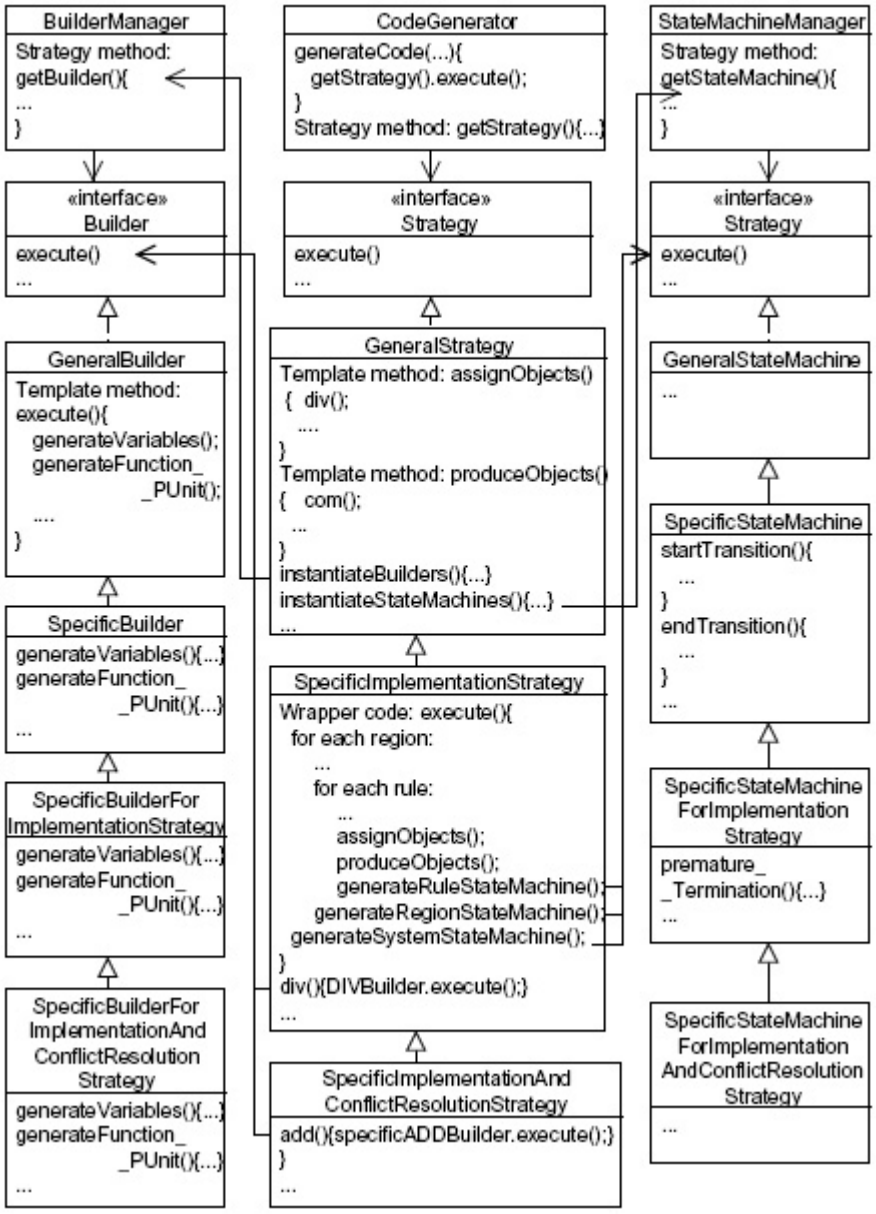


Fig. 10. A high-level view of the major modules in P Builder and their relationships.

SpecificImplementationAndConflictResolutionStrategy

A `SpecificImplementationAndConflictResolutionStrategy` class represents a strategy combining both an overall implementation strategy and a resource conflict resolution strategy (e.g., a combination of the region-oriented implementation strategy and the time-oriented resource conflict resolution strategy). The class implements the specific resource conflict resolution strategy by implementing the methods relevant to resource conflict resolution. For instance, because the updating of the multiset of objects in a region is accomplished in different ways in the different resource conflict resolution strategies, the definition of the method which is devoted to this operation is deferred to the `SpecificImplementationAndConflictResolutionStrategy` classes.

Builder

Instead of having each implementation class implement the complicated steps of generating the source code for a logically atomic operation, which would result in complicated code, we separate the actual construction of complex objects from the high-level procedure or algorithm according to which the construction is to proceed by delegating the actual construction to other classes. This is done in accordance with the Builder design pattern [4].

The Builder module has a similar structure to the Strategy module, and the classes in each of its layers fulfil similar roles. Briefly, a `BuilderManager` determines the specific Builder to instantiate and execute based on the options passed by the client to the Strategies module. The `BuilderInterface` interface specifies the basic functionality of Builder instances. The `Builder` class defines a template method which specifies all the steps a specific `Builder` implementation should execute (see Figure 10), and is responsible for selecting the specific `Builder` to execute given the client's requirements. The `SpecificBuilder` classes implement concrete `Builder` implementations for specific atomic operations (e.g., `ADDBuilder`, `DIVBuilder` and `COMBuilder`). The classes in the bottom layer of the module refine these concrete `Builder` implementations in order to accommodate the overall implementation strategy and conflict resolution strategy.

One of the major operations a concrete `Builder` implementation carries out is `generateFunctionPUnit`, which involves generating the hardware component that implements a specific logically atomic operation. There may be more than one approach to the implementation of the operation. For instance, at present the MIN operation is implemented as a macro expression which executes in one clock cycle. However, when the input P system is large, one might wish to apply logic-depth reduction in the implementation of this operation, or implement the operation in a different way. To add flexibility to the implementation of hardware components for the logically atomic operations, the Visitor pattern [4] is used. The Visitor pattern applies to contexts in which an operation needs to be performed on elements of

an object structure. It allows a new operation to be defined without changing the classes of the elements on which it operates. The source code below illustrates how the Visitor pattern allows the addition of a new method of implementing the MIN operation — a method which reduces the logic depth of the operation — without changing any of the existing classes.

```
interface MINMethod {
   generateMINImplementation();
   getNumberOfClockCyclesConsumed(); //needed for synchronisation purposes
}
class NoLogicDepthMIN implements MINMethod {
   ...
}
class LogicDepthMIN implements MINMethod {
   ...
}
class MINBuilder{
   MINMethod method; //the method of implementing MIN
   public MINBuilder() {
      method = new NoLogicDepthMIN;//default method is NoLogicDepthMIN
   }
      public void accept (MINMethod newMethod){
         method = newMethod; //set the method of implementing MIN
      }
      public void generateFunctionPUnit(){
         method.generateMINImplementation();
         ...
      }
   }
```

StateMachineGenerator

This module implements the linking and synchronisation of processing units discussed in Section 4.3 by generating the appropriate state machines. This involves using the control constructs of Handel-C (e.g., `if` and `for`) to implement basic state machines, as well as defining special-purpose components for the implementation of high-level linking and synchronisation of application-level processing units. When generating hardware source code for an operation, the `Builder` classes implement the state machine for the internal implementation of the operation. The StateMachineGenerator module is different in that it implements state machines for the linking and synchronisation of application-level processing units: at the level of reaction rules, at the level of regions, and at the level of the whole P system.

The representation of the hardware circuit as a tree structure of processing units (see Section 5.3) facilitates powerful and flexible approaches to the implementation of state machines. A `ProcessingUnitManager` is able to traverse the

tree of processing units. It can link and synchronise processing units by adding code to the preprocessing and postprocessing phases of the processing units. The added code can take on the form of a conditional expression, a single statement or a block of statements. It can create new processing units, delete processing units, include a processing unit in another processing unit, and modify the temporal relationships between processing units.

As already mentioned, the StateMachineGenerator module generates state machines at three levels. These three levels are handled by three different classes: `RuleStateMachine`, `RegionStateMachine` and `SystemStateMachine`. For example, if the region-oriented implementation strategy is adopted, `SystemStateMachine` produces state machines and processing units (such as the system execution coordinator mentioned in Section 3.3) for the synchronisation of the execution of region processing units in order to accomplish the transition-by-transition evolution of the P system.

Utilities

To ensure that the names of variables, functions, macros, signals and other constructs in the Handel-C representation of a hardware circuit consistently follow predefined naming conventions, a class called `NameGenerator` is implemented. This class contains a method for each type of construct that returns a unique identifier following the relevant naming convention. Although the functionality of `NameGenerator` is quite basic, it plays a fundamental role in the generation of understandable code, and therefore in the promotion of the maintainability of P Builder.

In the source code generated for a hardware circuit, various types of procedures are repeated in different parts of the code. For example, the procedure of initialising an array of registers is usually instantiated multiple times. A class called `FunctionGenerator` contains various methods which, when invoked with the appropriate parameters, returns the code for common types of procedures specialised according to the parameters. By eliminating duplication of code, the inclusion of `FunctionGenerator` promotes the maintainability of P Builder.

A class called `SystemConstants` defines all of the constant values used by the various components of P Builder. Defining the constant values in one place has two advantages, both of which promote the maintainability of P Builder. First, if any of these constants need to be modified in the future, only one modification needs to be made. Second, it eliminates the possibility of different components giving different values to the same constant, and therefore helps to prevent errors.

GenerationContext

To avoid the passing of parameters related to the current status of P Builder's operations (e.g., the region of the input P system currently being processed by P Builder), which can be error-prone and inefficient, a class called `GenerationContext`

is implemented using the Singleton design pattern [4]. The use of the Singleton pattern ensures that there is only one instance of the class. The various classes of objects implementing the functionality of P Builder access the fields of this instance in order to obtain information about the current status of P Builder's operations, and also update these fields in order to reflect changes to this status brought about by their own operations.

6 Evaluation of the region-oriented design

In this section, we evaluate our new region-oriented hardware design. More specifically, we report on the hardware resource consumption and clock rates exhibited by hardware circuits implementing P systems using the region-oriented design, and compare the results obtained with those obtained for hardware circuits implementing P systems using the rule-oriented design. We conclude the section with comments about the performance and scalability of the region-oriented hardware design in particular and Reconfig-P in general.

6.1 Details of the experiments

In the experiments, Reconfig-P was used to synthesise hardware circuits for a set of input P systems, according to different implementation strategies. This hardware source code was then synthesised into hardware circuits. The target hardware platform was a Virtex-II RC2000.

Table 3 describes the characteristics of the input P systems used in the experiments, including the number of regions and reaction rules in the P system, the number of objects (i.e., the product of the number of object types and the number of regions), the number of inter-region communications of object types in the definitions of reaction rules, the number of communication channels used in the implementation of the P system, and the total number of resource conflicts. In the table, P systems P1 through to P5 are used to test the effect of increasing the size of the input P system, and P6 and P7 are used to investigate the effect of using channels for the communication of objects and the effect of using semaphores for the dynamic updating of multisets of objects, respectively. Unlike P systems P1 through to P5, which have region hierarchies, P system P7 contains regions connected in a tissue-like fashion. P7 was included in order to facilitate the testing of the effect of having large numbers of communications and channels.

6.2 Experimental results

Efficiency of hardware circuits using the region-oriented design

Figure 11 illustrates the hardware circuits generated for the input P system P5 (which contains 5 regions and 50 reaction rules) using different implementation

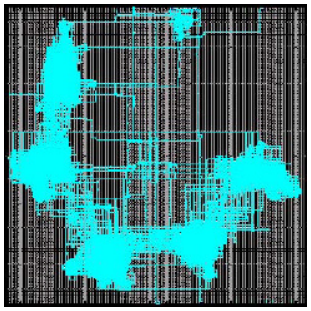
P system	Rules	Regions	Total objects	Inter-region communications	Channels	Total conflicts
P1	10	1	3	80	0	21
P2	20	2	6	16	5	32
P3	30	3	9	36	9	32
P4	40	4	12	44	12	40
P5	50	5	15	49	15	42
P6	50	25	75	74	64	42
P7	50	25	200	319	315	45

Table 3. Details of the input P systems used in the experiments.

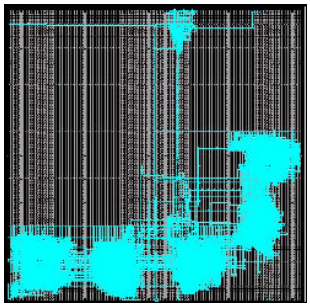
strategies. In keeping with the desired logical independence of regions in the region-oriented design, the five regions are realised as five separate, decoupled processing units on the hardware circuits using this design ('region-oriented circuits'). In contrast to the region-oriented circuits, the circuits using the rule-oriented design ('rule-oriented circuits') implement the P system as 50 rule processing units which operate both within and across the (merely conceptual) regions, resulting in intermingled, strongly coupled circuits (especially when the time-oriented conflict resolution strategy is used). Although regions are not explicitly represented when the rule-oriented design is used, it is still possible to discern the regions in the rule-oriented, space-oriented circuits. This is a consequence of the existence of processing units which operate at the region level (e.g., processing units involved in the replication of registers storing the multiplicity values of object types in a region and in the coordination of the values stored in these registers) and therefore implicitly represent regions.

Hardware resource consumption

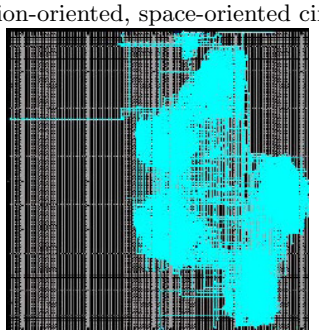
The results of the experiments demonstrate that region-oriented circuits tend to be more efficient in terms of hardware resource consumption than rule-oriented circuits. This is because (a) there are fewer core processing units to realise (since the number of regions is usually smaller than the number of reaction rules) in region-oriented circuits than in rule-oriented circuits, and (b) the number of channels used to implement inter-region communication, the main extra resource used in the region-oriented design, is minimised in our design and is therefore relatively small in general. As expected, among region-oriented circuits, those circuits using the time-oriented resource conflict strategy (region-oriented time-oriented circuits) consume fewer hardware resources than those using the space-oriented resource conflict strategy (region-oriented space-oriented circuits).



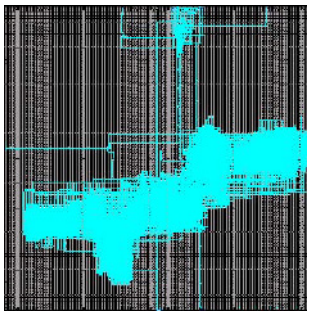
Region-oriented, space-oriented circuit



Region-oriented, time-oriented circuit



Rule-oriented, space-oriented circuit



Rule-oriented, time-oriented circuit

Fig. 11. Hardware circuits implementing input P system P5.

However, when the number of regions becomes large (e.g., in P system P6), the hardware resource consumption exhibited by region-oriented circuits is similar to that exhibited by rule-oriented circuits. If the number of communications is also large, and the number of channels used is large due to the specific characteristics of these communications (as is the case in, for example, P system P7), the hardware resource consumption exhibited by region-oriented circuits is greater than that exhibited by rule-oriented circuits. It is notable that in this case the region-oriented time-oriented circuits consume more hardware resources than the region-oriented space-oriented circuits. This is because our time-oriented conflict resolution strategy performs static interleaving for updating operations only for local objects (which account for only 2% of all updating operations in the case of P system P7) and therefore has to rely on Handel-C semaphores for the updating of external objects (a method which is less efficient in terms of hardware resource consumption).

P system	Resource consumption (%LUT)			
	Region-oriented		Rule-oriented	
	Space-oriented	Time-oriented	Space-oriented	Time-oriented
P1	2.03	1.83	2.25	2.08
P2	3.82	3.53	4.24	3.75
P3	5.72	5.47	6.49	5.79
P4	7.34	7.16	8.29	7.69
P5	9.20	8.85	10.43	9.33
P6	12.28	11.68	12.00	11.81
P7	14.20	14.72	13.00	13.32

P system	Clock rate (MHz)			
	Region-oriented		Rule-oriented	
	Space-oriented	Time-oriented	Space-oriented	Time-oriented
P1	63.84	65.62	67.20	66.96
P2	60.07	64.63	66.05	64.30
P3	58.15	60.41	66.51	66.52
P4	63.69	59.67	66.18	66.47
P5	58.9	58.52	65.78	64.90
P6	65.74	66.56	65.95	63.61
P7	58.79	43.48	62.50	60.00

Table 4. Experimental results for the hardware resource consumption and clock rates exhibited by circuits implementing the P systems listed in Table 3 according to various implementation strategies.

Clock rates

The clock rates achieved by region-oriented circuits tend to be lower than those achieved by rule-oriented circuits. This is due to the logic depth associated with the dynamic determination of applicable reaction rules in the object assignment phase in region-oriented circuits. However, the lower clock rates are compensated by a possible reduction in the number of clock cycles consumed in region-oriented circuits: the dynamic determination of applicable reaction rules guarantees that an optimal number of clock cycles is used in each round of the object assignment phase in region-oriented circuits, which is something that cannot be guaranteed in rule-oriented circuits. Therefore, in general the performance of region-oriented circuits is satisfactory. Having said this, however, when the object production phase involves a large number of external objects (as is the case in P system P7), the clock rate achievable by region-oriented time-oriented circuits is significantly reduced due to the logic depth associated with the use of semaphores.

Other observations

The experimental results show that circuits generated by Reconfig-P are very efficient in terms of hardware resource consumption, with the biggest P system P7 consuming only 14% of the total available resources when the region-oriented strategy is used and only 12% when the rule-oriented strategy is used.

As the target computing platform used in the experiments was the Virtex-II RC2000, the maximum clock rate at which a hardware circuit could execute and communicate with the host computer was 65 MHz. Given this maximum clock rate, the clock rates achieved by all the generated circuits are satisfying, especially when one considers that Reconfig-P did not apply logic-depth reduction in the experiments.

Reconfig-P also achieves good scalability. The hardware resource consumption increases sub-linearly as the size of the P system increases. Therefore increasing the size of the input P system does not have a significant effect on the circuits (especially rule-oriented circuits).

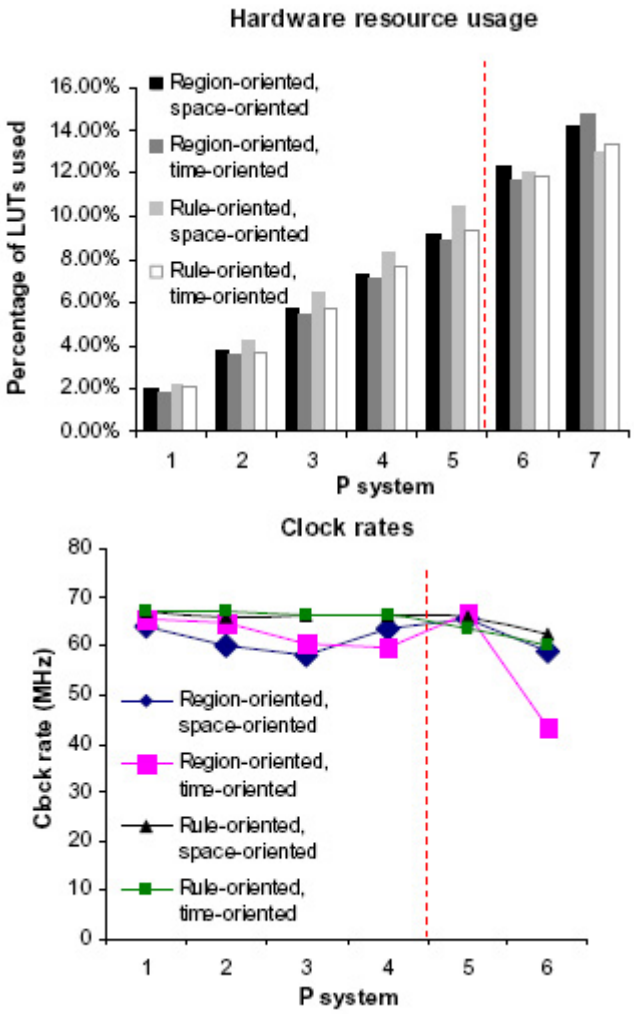


Fig. 12. Graphs of the experimental results presented in Table 4.

Since in the rule-oriented design a P system is implemented in such a way that the application of reaction rules is accomplished by dedicated rule processing units, hardware circuits generated according to this design are relatively insensi-

tive to those characteristics of a P system related to regions (e.g., the number of regions, the hierarchical structure of regions, and the way in which regions communicate). Therefore, in the experiments rule-oriented circuits exhibited consistently good performance. Region-oriented circuits, for obvious reasons, are more sensitive to region-related characteristics of the input P system. Nevertheless, for many P systems, region-oriented circuits and rule-oriented circuits exhibited a similar level of performance. Therefore, unless having the highest possible performance is important, if one wishes to execute a P system that involves computational activities directly associated with membranes or regions, the region-oriented strategy is perhaps the better choice.

7 Conclusion

In the course of developing the newest version of Reconfig-P, we have contributed

- an implementation of an elegant region-oriented hardware design that closely matches the intuitive conceptual understanding of a P system, exhibits good performance and scalability, and facilitates the future implementation of additional types of P systems,
- a novel design pattern which prescribes a general solution to the problem of designing an algorithm (source code) generation system in such a way that the logical and implementation aspects of the algorithm are kept separate, and
- a new version of P Builder designed according to the aforementioned design pattern which seamlessly integrates the rule-oriented, region-oriented, space-oriented and time-oriented implementation strategies and facilitates the adoption of additional implementation strategies.

We believe that the work described in this paper has enhanced the versatility of Reconfig-P and provided a solid foundation for the eventual development of a hardware platform for membrane computing applications responsive to the needs of a wide range of users. Indeed, we envision that Reconfig-P could be used in the not-too-distant future for the execution of significant real-life applications.

One of the most interesting potential application areas is the simulation of biological processes. In one sense, Reconfig-P is already ready for the simulation of biological processes. The only requirement is that such processes be modelled in terms of the basic P system models supported by Reconfig-P. However, the biological applications of membrane computing published to date typically involve P systems that incorporate non-standard or special features (such as reaction rates). For Reconfig-P to be able to execute specialised biological applications involving P systems with non-standard features, it would need to be augmented to incorporate these features. The extensibility of the newest version of Reconfig-P would facilitate such an augmentation.

One aspect of Reconfig-P that has been relatively neglected until now is its user interface. In particular, currently an ad hoc language for the specification of input

P systems is used. To enhance the usability of Reconfig-P, a more standardised language for the specifications of input P systems could be incorporated. One possibility is the incorporation of the P-Lingua language [5].

References

1. Bernardini, F. and Manca, V. 2002. P Systems with Boundary Rules. In G. Păun et al. (eds) *WMC-CdeA 2002*. Vol. 2597 of Lecture Notes in Computer Science, Springer, pp. 107–118.
2. Celoxica Ltd. 2005. Handel-C Language Reference Manual. `http://babbage.cs.qc.edu/courses/cs345/Manuals/HandelC.pdf`
3. Freund, R. and Oswald, M. 2002. P Systems with Activated/Prohibited Membrane Channels. In G. Păun et al. (eds) *WMC-CdeA 2002*. Vol. 2597 of Lecture Notes in Computer Science, Springer, pp. 261–269.
4. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman.
5. García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I. and Pérez-Jiménez, M. J. 2009. P-Lingua 2.0: New Features and First Applications. In R. Gutiérrez-Escudero et al. (eds), Proceedings of the Seventh Brainstorming Week on Membrane Computing, Sevilla, Spain, February 2–6, 2009, Vol. 1, pp. 141–167.
6. Ionescu, M., Păun, G. and Yokomori, T. 2007. Spiking Neural P Systems with an Exhaustive Use of Rules. *International Journal of Unconventional Computing*, Vol. 3, pp. 135–154.
7. Nguyen, V., Kearney, D. and Gioiosa, G. 2008. An Algorithm for Non-deterministic Object Distribution in P Systems and Its Implementation in Hardware. In D. W. Corne et al. (eds) *WMC9 2008*. Vol. 5391 of Lecture Notes in Computer Science, Springer, pp. 325–354.
8. Nguyen, V., Kearney, D. and Gioiosa, G. 2007. Balancing Performance, Flexibility and Scalability in a Parallel Computing Platform for Membrane Computing Applications. In G. Eleftherakis et al. (eds) *WMC8 2007*. Vol. 4860 of Lecture Notes in Computer Science, Springer, pp. 385–413.
9. Nguyen, V., Kearney, D. and Gioiosa, G. 2008. An Implementation of Membrane Computing using Reconfigurable Hardware. *Computing and Informatics*, Vol. 27, pp. 551–569.
10. Păun, A. and Păun, G. 2002. The Power of Communication: P Systems with Symport/Antiport. *New Generation Computing*, Vol. 20, pp. 295–305.
11. Păun, G. 2000. Computing with Membranes — A Variant: P Systems with Polarized Membranes. *International Journal of Foundations of Computer Science*, Vol. 11, pp. 167–182.
12. Păun, G. 2002. *Membrane Computing: An Introduction*. Springer.
13. Xilinx. 2007. Virtex-II Complete Data Sheet. `http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf`.