An extensible, maintainable and elegant approach to hardware source code generation in Reconfig-P

Van Nguyen*, David Kearney, Gianpaolo Gioiosa

School of Computer and Information Science, University of South Australia, Mawson Lakes Boulevard, Mawson Lakes, Adelaide 5095, Australia

ARTICLE INFO

Article history:
Available online 7 May 2010

Keywords:
Membrane computing
Reconfigurable hardware
Design patterns

ABSTRACT

Reconfig-P is a prototype hardware implementation for membrane computing applications. Currently there are two alternative designs for the implementation of P systems in Reconfig-P: the rule-oriented design and the region-oriented design. Driven by the goal of high performance, the rule-oriented design treats reaction rules as the primary computational entities and represents regions only implicitly. In contrast, the region-oriented design represents regions, rather than reaction rules, as the primary computational entities, and thereby more directly reflects the intuitive conceptual understanding of a P system and promotes the extensibility of Reconfig-P. To improve its practical usefulness and versatility, not only should Reconfig-P include both of the hardware designs, it should be maintainable and extensible so that it can easily and effectively incorporate new implementation strategies and implementations of additional types of P systems. 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 developed a new version of P Builder, our intelligent hardware source code generator, in accordance with a novel design pattern called Content-Form-Strategy. In this paper, we describe the Content-Form-Strategy pattern and the implementation of the new version of P Builder.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

We have recently devised two alternative designs for the implementation of P systems in hardware: the rule-oriented and region-oriented designs. Driven by the goal of high performance, the rule-oriented design treats reaction rules as the primary computational entities, and represents regions only implicitly. In contrast, the region-oriented design represents regions, rather than reaction rules, as the primary computational entities, and thereby more directly reflects the intuitive conceptual understanding of a P system and promotes the extensibility of Reconfig-P, our prototype hardware implementation for membrane computing applications. The analyses and results presented in [18] show that the rule-based design and region-based design have different strengths and weaknesses. Therefore, depending on the users’ requirements, either design might be the preferred option in a given circumstance. To improve its practical usefulness and versatility, not only should Reconfig-P include both of the hardware designs, it should be maintainable. In particular, changes made to one part of the system should, as far as possible, not necessitate changes in other parts of the system. It is also desirable that Reconfig-P be extensible so that it can easily and effectively incorporate new implementation strategies and implementations of additional types of P systems. These requirements have motivated us to re-engineer P Builder, the component of Reconfig-P responsible for generating customised Handel-C source code for the P system to be executed. In re-engineering P Builder, we aimed at promoting its
maintainability and extensibility through the use of a novel design pattern called Content-Form-Strategy. With this design pattern, P Builder is able to generate source code according to the rule-oriented and region-oriented designs, according to the space-oriented and time-oriented techniques, and according to other implementation approaches and techniques that might be incorporated into Reconfig-P in the future. P Builder is designed in such a way that it allows one to incorporate additional implementation approaches and techniques in a systematic and reliable way. In this paper, we explain the design and implementation of the re-engineered version of P Builder.

The content of this paper is as follows. In Section 2, we present the current status of Reconfig-P. In Section 3, we state the requirements for the re-engineered version of P Builder. In Section 4, we present the methodology that guided the design of the re-engineered version of P Builder. In Section 5, we describe the Content-Form-Strategy design pattern. In Section 6, we describe certain aspects of the implementation of P Builder. Finally, in Section 7, we discuss the significance of the major research contributions described in the paper, and identify opportunities for future research.

2. Current status of Reconfig-P

Reconfig-P [15–18] 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 automatically constructed by a hardware source code generator called P Builder using the hardware specification language Handel-C [6].

In [15,16], we proposed a strategy for the implementation of P systems in hardware to be used in Reconfig-P, which we call the rule-oriented design. The rule-oriented design takes a minimalistic approach to the implementation of the features of a P system in hardware. In this design, only those features of the intuitive conceptual understanding of a P system absolutely necessary to the computational operation of a P system (i.e., reaction rules and multisets of objects) are implemented explicitly as processing units or data structures. Membranes and regions are not directly represented, but must be inferred on the basis of the connections that exist between the reaction rules and the multisets of objects.

Although it promotes performance and efficiency, the rule-oriented design used in the existing version of Reconfig-P has some disadvantages which diminish the extensibility, understandability (and therefore maintainability) and elegance of Reconfig-P (please refer to [18] for more details). Furthermore, driven by the goal of high performance, this design puts an emphasis on P systems as models of parallel computation at the expense of not fully representing P systems as models of distributed computation. The alternative hardware design proposed in [18] is called the region-oriented design. In this design, regions, instead of reaction rules, are implemented as core processing units. The region-oriented design is intended to more faithfully reflect the distributed and compositional aspects of conceptual P systems at the implementation level and to promote the elegance and understandability of Reconfig-P by more closely reflecting the intuitive conceptual understanding of a P system. It is also intended to promote the extensibility of Reconfig-P by providing a framework within which the future implementation of additional types of P systems that require the explicit representation of regions and membranes can more easily be achieved.

Fig. 1 illustrates the region-oriented and rule-oriented strategies for implementing an input P system in hardware. Fig. 2 depicts the hardware circuit that Reconfig-P generates for an example P system when the region-oriented design is used.

To resolve resource conflicts that may occur in situations in which the multiplicity of an object type is to be updated by more than one parallel process, both the region-oriented and rule-oriented designs include two resource conflict resolution strategies: the space-oriented strategy and the time-oriented strategy. 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 (statically in the case of the rule-oriented design, and both statically and dynamically in the case of the region-oriented design) 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. For a more detailed description of the space-oriented and time-oriented strategies, please refer to [15–18].

3. Requirements for the new version of P Builder

As already mentioned, the region-oriented design has several attractive features, such as faithfulness to the intuitive conceptual understanding of a P system and 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. If Reconfig-P is to be used in practice, it is important for Reconfig-P to support the two alternative designs and to be maintainable and extensible. In [15,18], hardware circuits are generated by hardware source code generators which are

---

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 the FPGA to execute a variety of applications.
Fig. 1. A high-level illustration of how an input P system is represented according to the region-oriented and rule-oriented strategies.

Fig. 2. Illustration of the hardware circuit that Reconfig-P generates for an example P system when the region-oriented design and space-oriented conflict resolution strategy are used.

independent and hard-coded to accommodate the rule-oriented and region-oriented designs, respectively. Consequently, many sections of the source code for the hardware source code generators are very specific to the designs, and there are sections of code repeated in the two versions of the source code. Simply combining these hardware source code generators in Reconfig-P in order to achieve the functionality of both the rule-oriented and region-oriented designs is likely to compromise the desired maintainability and extensibility of Reconfig-P. For instance, in regard to maintainability, one would have to manually modify every piece of source code associated with the changes as well as every duplicate of the pieces. Such a process is very tedious and error-prone – it is entirely plausible that one would miss one or more pieces of source code, or inadvertently cause ripple effects in other pieces of source code. In regard to extensibility, the incorporation of an alternative implementation strategy for the existing P system models in Reconfig-P or an implementation of an additional type of P system would require the development of the implementation mostly from scratch, which is likely to require the duplication of a substantial amount of source code in the existing system. Not only is this expensive in terms of labour and detrimental to the maintainability of Reconfig-P, it also compromises the extensibility and robustness of the system – as the system grows, there is a risk that it will collapse under its own weight. Our purpose in re-engineering P Builder was to broaden the range of
implementation approaches with which P systems can be realised as hardware circuits, as well as to develop a sophisticated object-oriented design that promotes source code reusability and thus the maintainability and especially the extensibility of Reconfig-P.

4. Design methodology

FPGAs are increasingly used to provide high-performance solutions for applications in a wide range of domains. However, implementing an application on an FPGA is not a trivial task. Even with the introduction of C-based hardware description languages (e.g., Handel-C and SystemC) which ease the process of synthesising a hardware circuit for an application, writing hardware description source code requires significant effort. This has motivated researchers to investigate the partial or complete automation of the process of developing hardware description source code for applications to be executed on a hardware platform such as an FPGA [1–3,5,10,11,14,21–24]. Most of these research efforts have focused on the idea of generating hardware description source code from a set of UML diagrams that model the application to be executed. However, to generate hardware description source code for an application using UML, the execution of the application first needs to be completely specified in a set of domain-specific UML diagrams. Producing such a specification is difficult. Although many research results related to the application of UML in hardware design and the translation of UML to hardware description source code have been obtained, these results are mostly theoretical in nature. For example, results related to the performance and efficiency of the hardware circuits synthesised from the generated hardware description source code have not yet been reported.

As P Builder directly targets membrane computing applications, we did not adopt the aforementioned approach since the modelling of the complete behaviour of a membrane computing application using UML introduces an additional and unnecessary layer of complication to the system. Instead, we took an approach according to which P Builder generates hardware description source code directly from the specification of the input application. In addition, we designed P Builder in such a way that it is capable of generating hardware description source code according to a variety of implementation strategies chosen so as to enhance the efficiency of the synthesised hardware circuit that realises the membrane computing application and to satisfy different user requirements.

In the design of the new version of P Builder, our guiding design principle was that of separation. More specifically, we viewed the hardware implementation of a P system as a complex of form and content, and attempted to treat the form of this complex in isolation from its content. We now introduce this separation strategy in the form of a design pattern. A design pattern prescribes thoroughly tested and effective solutions to design problems, and therefore enables the creation of flexible, reusable and elegant object-oriented designs.2

5. 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 which instantiates one of a number of P system models (e.g., the tissue-like P system model) according to one of a variety of alternative implementation techniques (e.g., with the rule-based design and space-oriented conflict resolution, or with the region-based design and time-oriented conflict resolution). This problem can be viewed as an instance of a more general problem: that of producing a parallel implementation for a range of applications defined in terms of a computational model, where this parallel implementation must be constructed according to one of a variety of possible strategies. To make this problem feasible to solve, it is essential that a unifying abstract representation for the parallel implementations of the applications based on various strategies be devised. Also, when constructing such a parallel implementation, 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 strategies in the future. However, it is often quite difficult to achieve such a separation. Our novel design pattern, which we call Content-Form-Strategy, prescribes a general solution to the general problem just outlined. For the sake of simplicity, we now describe the Content-Form-Strategy pattern in the context of Reconfig-P, even though the pattern can be applied to other systems that share the same objective. We first start with a general analysis of the execution of P systems.

5.1. General analysis of the execution of a P system

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 = \((p \text{DIV}, q \text{MIN}, r \text{MUL}, s \text{SUB}, t \text{COM}, u \text{ADD})\), where
- \(p = 0 \text{ or } 1\),
- \(q, r, s, t, u \geq 0\).

2 For more information about object-oriented design patterns, we refer the reader to [8], the classic reference in the field.
Fig. 3. An analysis of the behaviour of a P system: the execution of a reaction rule involves zero or more logically atomic operations, each of which is realised as an atomic hardware component.

- **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).

From this perspective, the execution of a P system fundamentally involves the execution of a combination of these logically atomic operations in a certain temporal order. At the hardware implementation level, each of the atomic operations can be realised as an atomic hardware component, as illustrated in Fig. 3. These atomic components are the building blocks for the construction of any particular hardware circuit. The names of the components reflect the main computational operations involved in their implementation. To realise higher-level operations occurring at the level of reaction rules, at the level of regions, or at the level of entire P systems, it is necessary to link and synchronise the execution of the basic hardware components. The different ways to connect and synchronise these basic hardware components, which result in hardware circuits implementing different P systems or implementing a P system using different strategies, are illustrated in Fig. 4.

5.2. Definition of the pattern

The observations presented in the previous section form the foundation of the Content-Form-Strategy design pattern. Thus a key idea of the Content-Form-Strategy design pattern is that a parallel implementation of a P system may be viewed as a complex of **form** and **content**, where the units of content are the logically atomic computational operations and the form is the way in which these units of content relate to each other logically and temporally. If a parallel implementation is represented as a flowchart such as that shown in Fig. 4, then the shaded boxes in the flowchart comprise the content of the implementation, and the diamonds, arrows, bars, unshaded boxes and overall structure of the flowchart comprise the form of the implementation. Note that computational operations that are included in the implementation solely for the purpose of linking and synchronising other computational operations are regarded as part of the form of the implementation (i.e., they would be represented as unshaded boxes in a flowchart). A **strategy** is a specification of a particular way of constructing a parallel implementation as a complex of form and content that has been specially designed to achieve certain implementation goals (e.g., high performance, low memory usage or minimal routing). Different strategies may require the inclusion of different logically atomic computational operations, and will necessitate the realisation of different logical and temporal relationships between these operations.
The steps of the solution prescribed by the Content-Form-Strategy pattern are as follows:

1. Define an abstract model of a parallel implementation as expressed in the implementation language.
2. For each implementation strategy, identify the logically atomic computational operations in terms of which the parallel implementation 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 a parallel implementation defined in step 1.
4. For each implementation strategy, 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 a parallel implementation.
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 preconditions, data writing and temporal relationships (i.e., those preprocessing operations, postprocessing operations, data writing and temporal relationships that obtain regardless of the implementation strategy).
Table 1
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.

<table>
<thead>
<tr>
<th>Aspect of Content-Form-Strategy pattern</th>
<th>Example of how the aspect applies to the generation of a hardware circuit for a P system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation to be generated</td>
<td>Handel-C program that specifies the hardware circuit for the input P system</td>
</tr>
<tr>
<td>Current implementation approaches</td>
<td>Rule-oriented/region-oriented design; space-oriented/time-oriented resource conflict resolution; deterministic/nondeterministic execution</td>
</tr>
<tr>
<td>Possible future implementation approaches</td>
<td>Alternative algorithms for nondeterministic object distribution</td>
</tr>
<tr>
<td>Logically atomic computational operations</td>
<td>DIV, MIN, MUL, SUB, COM, ADD (see Section 5)</td>
</tr>
<tr>
<td>Logically atomic operations as expressed in terms of the abstract model of an implementation</td>
<td>An ADD operation is implemented as a function or a macro in the space-oriented mode</td>
</tr>
<tr>
<td>Preprocessing for the execution of a logically atomic operation</td>
<td>An ADD operation proceeds only if the corresponding reaction rule is applicable</td>
</tr>
<tr>
<td>Temporal relationship between logically atomic operations</td>
<td>A SUB operation should execute only after its associated MIN operation has executed</td>
</tr>
<tr>
<td>Temporal relationship expressed in terms of the abstract model of an implementation</td>
<td>Basic parallel and sequential operations are defined in the par and seq constructs provided by Handel-C</td>
</tr>
<tr>
<td>Operations included solely for the purpose of linking and synchronisation</td>
<td>Operations performed by the system execution coordinator</td>
</tr>
<tr>
<td>Abstract model of an implementation as expressed in the implementation language</td>
<td>Abstract model of a hardware circuit as expressed in a Handel-C program</td>
</tr>
<tr>
<td>Template algorithm</td>
<td>An algorithm that specifies the high-level stages of a transition (e.g., the object assignment phase)</td>
</tr>
<tr>
<td>Filling out of the template algorithm</td>
<td>Inclusion of an algorithm for the space-oriented or time-oriented conflict resolution strategy to fill out the object production phase</td>
</tr>
</tbody>
</table>

7. Based on the result of step 6, define a template algorithm which specifies the features common to all possible parallel implementations for all implementation strategies in terms of the elements of the abstract model of a parallel implementation.

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 a parallel implementation.

9. Express each of the algorithms defined in step 8 in the desired implementation language.

Table 1 illustrates how the Content-Form-Strategy 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.

6. Implementation of the new version of P Builder

Our implementation of the new version of P Builder is based on a well-defined architecture derived from the Content-Form-Strategy pattern. This architecture is shown in Fig. 5. The white boxes in the figure represent the major modules in the implementation, while the shaded boxes represent object-oriented design patterns that were adopted in the implementation of these modules.

The UML class diagram in Fig. 6 shows the main classes used in the implementation. These classes implement the modules defined in the architecture. The main modules of P Builder include Operation Builder, State Machine Generator, Implementation Strategies and Hardware Circuit Abstraction. The Operation Builder, State Machine Generator and Implementation Strategies modules correspond to the content, form and strategy components of the Content-Form-Strategy pattern.
pattern, respectively. The fact that a hardware circuit is viewed as a complex of form and content necessitates the creation of a corresponding representation of the hardware circuit at the implementation level that facilitates the construction of the hardware circuit via the separate construction of its content and form. The Hardware Circuit Abstraction module is implemented for this purpose. It provides an abstract representation of a Handel-C specification of a hardware circuit. The Operation Builder module is responsible for generating Handel-C hardware components for the logically atomic operations, whereas the State Machine Generator module is responsible for the linking and synchronisation of these components. The Operation Builder and State Machine Generator modules customise the hardware circuit abstraction to produce a specific hardware circuit implementing a P system. The customisation process is performed according to an implementation strategy.
provided by the Implementation Strategies module. The Implementation Strategies module consists of a set of algorithms for the generation of hardware circuits for input P systems. Depending on the type of the input P system and the implementation strategy chosen, the Implementation Strategies module provides a specific algorithm which guides the Operation Builder module and State Machine Generator module in the generation of a hardware circuit for the input P system. We now briefly describe the modules.

6.1. Hardware Circuit Abstraction

The high-level function performed by P Builder is the conversion of an input P system into a Handel-C specification of a hardware circuit which can be used to execute the P system on the reconfigurable hardware platform of Reconfig-P. To facilitate P Builder in the performance of this function according to the Content-Form-Strategy pattern, the Hardware Circuit Abstraction module provides a novel abstract representation of a Handel-C specification of a hardware circuit.

A hardware circuit (or, more precisely, a Handel-C specification of a hardware circuit) may be regarded as a complex processing unit composed of simpler processing units. According to this abstraction, the ‘contents’ of the circuit (the hardware components that implement the logically atomic operations) are implemented as processing units. The ‘form’ of the hardware circuit in this abstraction may be differentiated into two classes: static form and dynamic form. The static form comprises the logical and temporal relationships among the processing units in the circuit that can be determined at compile time, while the dynamic form comprises the logical and temporal relationships among the processing units that can be determined at execution time. For instance, temporal relationships among a group of processing units of the static type can be represented by means of a composite processing unit which contains related processing units ordered according to the temporal relationships. The embedded processing units in turn can contain other temporally related processing units. So the static form of the circuit is represented as a nested structure of different types of processing units. The dynamic logical and temporal relationships among a group of processing units include the control flow, linking and synchronisation among the processing units of the static type.

The key idea of the proposed hardware circuit abstraction is that a Handel-C specification of a hardware circuit is represented as a tree of processing units, as illustrated in Fig. 7. The structure of the tree defines the static form of the corresponding hardware circuit. There are two types of processing units: parallel processing units and sequential processing units. These two types of processing units are used for the explicit purpose of capturing the static logical and temporal relationships among the processing units in the hardware circuit. A processing unit can be used for the implementation
of part of the ‘content’ of the circuit, for the implementation of synchronisation operations, or as a container for 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 a 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 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 the control flow, linking and synchronisation of the dynamic form of the circuit, 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 (e.g., the condition for a while loop), the execution of a single statement (e.g., the storage of data in a register), or the execution of a collection (block) of statements.

The most elegant and efficient way to represent a compositional structure of processing units in an object-oriented system is to use the Composite design pattern [8]. 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.

The UML class diagram in Fig. 8 shows how these ideas are represented in the implementation of P Builder. Fig. 7 shows an example of Handel-C source code generated from a tree of processing units. The correspondence between the processing units and the code sections is marked in the figure.

6.2. Implementation Strategies

The Implementation Strategies module provides a set of algorithms that construct specific hardware circuits implementing P systems based on the hardware circuit abstraction presented above. The hardware circuits are constructed according to a set of implementation strategies. Each of the algorithms achieves its goal by defining the static form of the hardware circuit, which contains high-level composite processing units and placeholders for the units of content of the circuit and their interconnection, and guiding the Operation Builder and State Machine Generator modules to create and place specific ‘content’ processing units into the static form of the circuit to generate the dynamic form among the processing units required for the realisation of the desired behaviour. The Implementation Strategies module constructs the static form for a hardware circuit by making use of basic parallel and sequential Handel-C constructs such as par and seq constructs. Currently there are classes for the rule-oriented, region-oriented, space-oriented and time-oriented strategies. This module could be extended in future to include classes for other strategies.

The top-level class of the Implementation Strategies module is called GeneralStrategy. This class implements the high-level algorithm defined in step 7 of the Content-Form-Strategy pattern (see Section 5). This high-level algorithm implements the features common to all strategies, and specifies the processing steps common to all strategies (implemented as subclasses
of GeneralStrategy). Depending on the requirements of the client using the Implementation Strategies module, a specific strategy for the implementation of the input P system on a hardware circuit needs to be applied. 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 (defined in step 8 of the pattern) in a separate class. However, taking such an approach would likely result in duplication of code, and consequently diminish maintainability. Therefore, a better implementation approach is required.

As discussed in [15], the high-level execution algorithm that underlies each of the different implementation strategies consists of an object assignment phase and an object 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 combined to realise high-level operations. Therefore, we implement the high-level execution algorithm in the GeneralStrategy class, and implement the specialised versions of this algorithm for the different strategies in different subclasses of the GeneralStrategy class. This is achieved elegantly through the use of the Template design pattern [8]. Following this 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, and possibly adds other code (e.g., wrapper code) in order to implement a more specialised algorithm. A template method should be placed in the top-level class of the hierarchy of classes that share the procedure specified in the template method. As shown in Fig. 6, 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.

6.3. Operation Builder

The Operation Builder module is responsible for implementing hardware components for the logically atomic operations in P systems (see steps 2 and 3 of the Content-Form-Strategy pattern).

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, multiple operations are combined and 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. Addition and subtraction are relatively inexpensive operations and, according to the performance results for the current version of Reconfig-P (reported in [15]), 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, PBuilder has the ability to generate the hardware circuit in such a way that the logically atomic operations are performed over several clock cycles.

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 [8].

The Operation Builder module has a similar structure to the Implementation Strategies module, and the classes in each of its layers fulfill similar roles. One of the major operations a SpecificBuilder 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 [8] is used. The Visitor pattern applies to contexts in which an operation needs to be performed on elements

---

3 The Xilinx Virtex-II FPGA used in the implementation contains hardware multipliers that allow efficient and high-performance implementation of multiplication operations [25]. However, where one of the operands is a constant, multiplications can be more efficiently implemented on slices using either bitshifts or constant coefficient multipliers.
of an object structure. It allows a new operation to be defined without changing the classes of the elements on which it operates.

6.4. State Machine Generator

Recall from Section 6.2 that the Implementation Strategies module, which defines the skeleton of the hardware circuit implementing a P system and its static form, guides the Operation Builder and State Machine Generator modules in generating the hardware components for the skeleton of the hardware circuit and the linking and synchronisation among them. Also recall from Section 6.3 that the Operation Builder module generates these hardware components. Now, in order for the hardware circuit to achieve a desired behaviour, it is necessary to implement the dynamic temporal relationships (processing units). The State Machine Generator module is responsible for generating such linking and synchronisation. For instance, Fig. 4 shows high-level UML activity diagrams for the execution of a region processing unit with different implementation strategies. The hardware components implementing logically atomic operations (described in Section 5), which are represented as shaded boxes in the diagram, are constructed by the Operation Builder module. The region processing units, vertical down arrows and solid bars are generated by the Implementation Strategies module as part of the static form of the hardware circuit. All other arrows, diamond shapes and unshaded boxes are implemented in hardware by the State Machine Generator module.

The process of generating linking and synchronisation involves using the control constructs of Handel-C (e.g., if and for) to implement basic control flow, as well as defining special-purpose components for the implementation of high-level linking and synchronisation of application-level processing units. We will now briefly describe some of the linking and synchronisation tasks performed by the State Machine Generator module.

6.4.1. Control flow in a sequential processing unit

To implement the simple control flow within a sequential (composite) processing unit (e.g., a rule or region processing unit), the basic control constructs provided by Handel-C such as if, while, for and goto are used. For example, the diamonds in the activity diagram shown in Fig. 4 are implemented using conditional constructs such as if.

6.4.2. Linking and synchronisation in a parallel processing unit

In the implementation of a parallel processing unit (e.g., the system processing unit), it is necessary to link and synchronise the embedded processing units. For instance, in Fig. 4, the linking and synchronisation needs to be established between the region processing unit and the processing units implementing the logically atomic operations such as DIV, MIN and MUL.

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 (this is part of the static form and is implemented by the Implementation Strategies module) 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:

```c
// 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;
        }
    }
}
```

The State Machine Generator module creates the dynamic form for the hardware circuit by generating the linking and synchronisation among the processing units at the level of reaction rules, at the level of regions, and at the level of the whole

---

4 Fine-grained instances of linking and synchronisation, which are intrinsic parts of the execution of a hardware component, are generated by the Operation Builder module.
P system. These three levels of linking and synchronisation are performed by three different classes (together with their subclasses): RuleStateMachine, RegionStateMachine and SystemStateMachine. For example, if the region-oriented design is adopted, SystemStateMachine produces state machines and processing units for the synchronisation of the execution of region processing units in order to accomplish the transition-by-transition evolution of the P system.

The representation of the hardware circuit as a tree of processing units (see Fig. 7) facilitates powerful and flexible approaches to the implementation of state machines. A ProcessingUnitManager in the hardware circuit abstraction is able to traverse the tree of processing units. The ProcessingUnitManager can assist the State Machine Generator module to link and synchronise the 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.

7. Conclusion and future work

In the course of re-engineering P Builder, we have made two significant contributions. The first contribution is a novel design pattern which prescribes a general solution to the problem of designing a parallel implementation (source code) generation system in such a way that the logical and implementation aspects as well as the content and form of the implementation are kept separate. The second contribution is 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 has 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.

With P Builder implemented according to the Content-Form-Strategy design pattern, Reconfig-P provides a general framework for the incorporation of various implementation strategies for a range of P system models. Consequently, there are several extensions that could be immediately implemented to augment the functionality of Reconfig-P and thus broaden the range of applications it can execute. We now discuss some possible extensions.

First, it is possible to augment Reconfig-P so that it implements nondeterministic object distribution. Our algorithm for nondeterministic object distribution is the DND algorithm proposed in [17]. To incorporate an implementation of the DND algorithm into Reconfig-P, it is essential to express the algorithm in terms of the logically atomic operations and associated linking and synchronisation operations (as prescribed by the Content-Form-Strategy pattern). The latest version of P Builder, which was designed according to the Content-Form-Strategy pattern, provides a template that defines the fundamental components underlying most of the types of P system models and a flexible way to establish the required logical and temporal relationships between these components. Indeed, we have already succeeded in expressing the DND algorithm in terms of the components and connections identified in this template.

Second, implementing a P system model with true membrane creation and dissolution would require the hardware circuit that implements the P system to perform runtime reconfiguration. Although run-time reconfiguration has been achieved in some computing systems (e.g., see [4,7,9]), given the current state of FPGA technology, implementing membrane creation and dissolution using run-time reconfiguration would very likely result in a significant performance reduction. However, if one desires to execute applications with membrane creation and dissolution, Reconfig-P facilitates a straightforward implementation of a technique that simulates a true implementation of membrane creation and dissolution. Similar to what has been proposed in [20], this technique involves, instead of dynamically creating and destroying relevant components in the hardware circuit, constructing all the regions at compile-time, and then activating or deactivating them as necessary at run-time.

Third, one of the most interesting potential application areas for Reconfig-P 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. Obviously, the extensibility of Reconfig-P would facilitate such an augmentation.

Fourth, probabilistic P systems play an important role in applications of membrane computing, especially biological applications. For example, probabilities associated with reaction rules can be used to model chemical reaction rates. Reconfig-P does not currently support the specification of arbitrary probability measures. The feasibility of adding such support is a topic for future research.

Finally, it would be interesting to see how far the techniques used in the development of our FPGA-based system can be transferred to other computing platforms, such as GPU-based systems. GPUs (graphical processing units) are being increasingly used to accelerate general-purpose tasks as opposed to just graphics applications. Recently, Martínez-del-Amor et al. [12,13] have investigated the idea of implementing a GPU-based membrane computing simulator that aims to exploit the massive parallelism of a GPU for the purpose of efficient execution of membrane computing applications. The
simulator consists of a host CPU and a GPU (a Tesla C1060) that is coupled with the CUDA C programming environment. The Tesla C1060 consists of 240 streaming-processor cores that can execute thousands of concurrent threads over large datasets. Further investigation is required to determine whether the GPU programming model is suitable for the development of a system such as Reconfig-P.

References


