A Hardware Compiler Realizing Concurrent Processes in Reconfigurable Logic

Oliver Diessel[1] and George Milne[2]

[1] School of Computer Science and Engineering
University of New South Wales
UNSW Sydney NSW 2052, Australia
odiessel@cse.unsw.edu.au

[2] Department of Computer Science and Software Engineering
University of Western Australia
Crawley WA 6009, Australia
george@cs.uwa.edu.au

Abstract. Reconfigurable computers based on field programmable gate array technology allow applications to be realized directly in digital logic. The inherent concurrency of hardware distinguishes such computers from microprocessor-based machines in which the concurrency of the underlying hardware is fixed and abstracted from the programmer by the software model. However, reconfigurable logic provides us with the potential to exploit "real" concurrency. We are therefore interested in knowing how to exploit this concurrency, how to model concurrent computations, and which languages allow us to program this dynamic hardware most effectively.

The purpose of this paper is to describe an FPGA compiler for the Circal process algebra. In so doing, we demonstrate that behavioural descriptions expressed in a process algebraic language can be readily and intuitively compiled to reconfigurable logic and that this contributes to the goal of discovering appropriate high-level languages for run-time reconfiguration.

The introductory sections of the paper motivate our work and describe the modelling methodology adopted by Circal before describing a technology-independent circuit representation for system behaviours specified in Circal. The latter sections describe the automatic mapping of these circuits to the XC6200 series FPGA chips of a SPACE.2 reconfigurable computer.

1 Introduction

A technique for compiling from a behavioural language oriented towards communication and concurrency into field-programmable gate array devices (FPGAs) is presented. This approach permits the rapid design and prototyping of complex, control-oriented systems, where a design is produced automatically from a much more abstract behavioural specification. Detailed implementation issues are dealt with by the compiler, with the designer concentrating on higher level design decisions such as overall system architecture. In addition, the compilation technique developed gives access to techniques for formally establishing design correctness and, furthermore, will lead to languages and compilation techniques for dynamically reconfigurable computing.

The term *reconfigurable computer* is currently used to denote a machine based on FPGA technology. This chip technology is programmable at the gate level, thereby allowing any discrete digital logic system to be instantiated. It differs from the classical von Neumann computing paradigm in that a program does not reside in memory but rather an application is realized directly in digital logic. Since this logic is repeatedly programmable, the underlying FPGA platform may repeatedly be instantiated to create completely different custom computer realizations for each distinct application. Certain FPGA technologies are also *dynamically reconfigurable* in that part of the FPGA logic may be reconfigured while another part is running and, even more radically, this technology permits part of the programmable logic to be used to reconfigure another part in real time.

For some computing and electronic control applications we are able to exploit the inherent concurrency of digital logic to directly realize algorithms as custom hardware to gain a performance advantage over software executing on conventional microprocessors. By providing a relatively cheap and rapid implementation technology, the advent of reconfigurable logic in the form of FPGAs has empowered engineers to experiment with hardware-based algorithms. It has been shown that FPGAs can host particular classes of high performance applications very successfully, and at relatively low cost [1–3].

It is the inherent presence of concurrent activity that distinguishes hardware from software. Software operates at a conceptual level of abstraction such that the concurrency in the underlying microprocessor hardware is unknown to the software programmer and is unseen when the code runs. However, reconfigurable computing allows the potential for exploiting the concurrency found in digital logic.

Unfortunately, current FPGA specification and design methodologies are not well-suited to the task of harnessing this concurrency. On the one hand, the low-level orientation of hardware description languages, such as VHDL and Verilog [4, 5], inhibit the effective expression of parallelism in abstract, behavioural terms. Synthesis and technology mapping tools for these languages are not yet designed for rapid compilation or dynamic circuit replacement. On the other hand, high-level languages that have been augmented for FPGA design [6, 7, for example] rely upon the designer's explicit expression of parallelism and upon compilation techniques to identify implicit parallelism. Languages such as these are typically translated to VHDL as an intermediate form, and then synthesized and mapped using conventional tools.

We contend that sequential high-level languages and static hardware description languages cannot satisfactorily be extended to describe inherently parallel and dynamic hardware structures. The design of suitable languages for reconfigurable computing should begin with a definition and innate understanding of such features of the hardware. This paper takes a significant step in that direction by demonstrating that we can intuitively and rapidly compile an abstract, high-level language that is oriented to describing concurrency and communication into reconfigurable logic. We show how the core features of process algebra [8–10], and the Circal process algebra in particular [8, 11], can be mapped into reconfigurable logic.

The rationale for focusing on using a process algebra as the basis of a language for specifying reconfigurable logic is that it allows us to express the behaviour of a design in an abstract, technology-independent fashion and it emphasizes computation in terms of a hierarchical, modular, and interconnected structure of concurrently active finite state machines. The modular focus of process algebra, which arises due to the constructive, algebraic nature of modelling via an appropriate composition operator, is seen by the authors as a key approach to both formalizing and thence realizing the concepts of partial and dynamic reconfiguration, where we may swap in and replace individual components rather than performing complete reconfigurations.

Our ongoing research programme has the goal of determining appropriate high-level language features and attendant compilation techniques for the rapid realization of dynamically reconfigurable applications. This paper presents the results of the first phase of this programme, during which we developed techniques for compiling the Circal process algebra language into a digital logic representation, and from this technology-independent mode into a particular FPGA technology. While the results utilize the experimental Xilinx XC6200 technology [12], they provide us with a "proof of concept" that permits us to apply similar techniques to current FPGA product technologies. Future research aims to build on the strategies presented here in order to develop techniques for compiling dynamic structures specified in the dsCircal process algebra [13], which facilitates the modelling of systems whose structure changes as the system runs, such as found in the dynamic, run-time reconfiguration of FPGA-based systems.

A high-level language based on a process algebra is quite different from classical hardware description languages, such as VHDL and Verilog, that are oriented towards register-transfer and gate-level descriptions. Instead, our approach provides designers with a design paradigm focused on behavioural process modules and their interconnection. Because of its modular focus, our approach aids the rapid compilation and partial reconfiguration of designs at run-time. Our approach also presents us with the potential for formally verifying the compilation algorithm, thereby allowing us to state that all designs expressed in the source language are correctly implemented in the underlying reconfigurable computing engine. Related research on verifiable compilation from Occam to FPGAs was performed by Shaw and Milne [14], while Page, Luk, Jifeng, and Bowen [15, 16] also developed Occam to FPGA compilation techniques.

The rapid compilation of Circal models allows assemblies of interacting finite state machines, upon which logic control paths are based, to be implemented quickly. Apart from logic controllers, such as used in [17], we may use the approach to build and quickly modify test pattern generators that function at near hardware speed, an application described in detail in [18]. The future realization of such applications will involve utilizing a source language resulting from embedding the Circal parallel programming model within a language such as C++ or Java; the intent of the research reported here has been to focus on the compilation of hierarchies of interacting, concurrently active finite state machines resulting in techniques for exploiting FPGA parallelism. As such, we have gone some way to attacking the semantic gap between application programming languages and configurable hardware, the case for which is elegantly presented by Snider, Shackleford, and Carter [7].

In the following section we provide an overview of the Circal process algebra which takes on the role of source language for our compiler. Section 3 introduces our contribution with an overview of the compiler. We describe a technology-independent circuit model of Circal processes in Section 4. The mapping of these circuits to FPGAs, and Xilinx XC6200 chips in particular, is discussed in Section 5. The automation of the mapping from behavioural Circal descriptions is described in Section 6. An outline of the research described in this paper first appeared in [19].

2 The Circal description language

In this section we present Circal as a descriptive medium for reconfigurable computing. We describe the key language concepts and how they are used to describe concurrent systems.

Circal is a formal language used to describe interacting systems by modelling the behaviour of their component processes, each as a finite state machine. These component processes then interact, based on the synchronous occurrence of transition events. Systems, and processes in turn, are described hierarchically and in a modular fashion. The description of a system thus typically proceeds in a top-down manner with the elaboration of component processes leading to further decomposition until the desired level of description is reached. In principle it is possible to elaborate designs to the level of logic gates and signals (see, for example, [11]).

We informally present the Circal language using state transition diagrams following the approach adopted in [20]. A detailed presentation of these language constructs, their semantics, and how they are used to formally describe and verify digital logic can be found in [8, 11, 21].

2.1 Describing hierarchies of interacting finite state machines

Process algebras such as Circal are mathematical formalisms for describing systems of interacting finite state machines, where interaction is achieved by synchronizing the transitions that occur in the individual finite state machines. This can be done in several ways, which leads to differentiation between the various process algebras that appear in the literature [22, 10, 9, 11].

Circal utilizes three structural language constructs, namely a *parallel composition operator*, a *hiding* operator and a *relabelling* operator. The combination of the three operators allows for the structure of any system to be modelled as a hierarchy of abstractions, as shown in Figure 1(a). Each box represents a process that is composed of the components

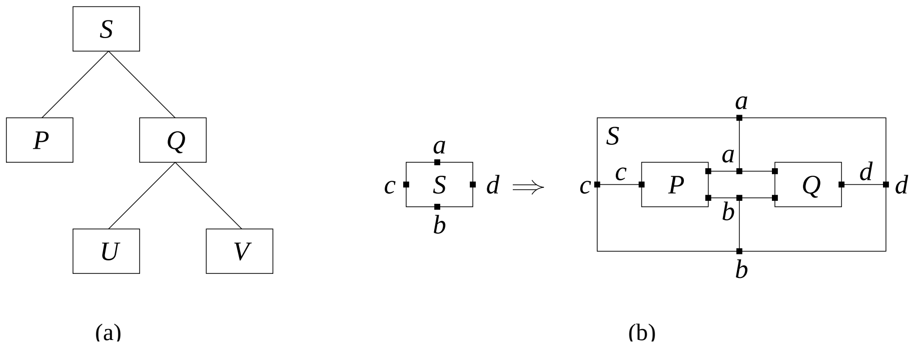

Fig. 1. (a) Hierarchy of system components; (b) Graphical representation of parallel composition.

at the next lower level in the hierarchy. For example, process Q consists of two components: process U and process V. Only the boxes that are leaves of the hierarchy explicitly encapsulate behaviour. In the following, such leaves are called *behavioural processes* and these correspond to individual finite state machines. Each process interacts with other processes through communication ports. Interaction between processes occurs via transition actions or events that are associated with the ports and in Circal a communication channel connects all the ports that are labelled with the same action.

If we look at the hierarchy given in Figure 1(a) each process, apart from the root, is embedded within its parent by composing it in parallel with its siblings, by possibly hiding some of the actions that are used for interaction, and by possibly relabelling other actions. For example, P and Q are embedded within S as shown in Figure 1(b). Communication ports are represented by bullets on the periphery of a box and communication channels are represented by lines connecting two or more ports. These channels are labelled with their associated actions. The embedding of a set of processes within the next level of the hierarchy is represented by a surrounding box with bullets on its periphery representing the ports that are not hidden after the composition. These ports are externally labelled and connected by channels to the correspondingly named internal ports. For example, in Figure 1(b), P and Q communicate through the channels labelled by actions a and b and may interact independently with their environment via ports labelled c and d.

The box that embeds a set of processes represents the *interface* of the composite process. Every process has a *sort*, which is the set of action names that label the ports on the box that embeds its components. For example, in Figure 1(b), S has sort $\{a, b, c, d\}$, P has sort $\{a, b, c\}$, and Q has sort $\{a, b, d\}$. For a behavioural process, its sort (or interface) must contain at least all the actions that occur in its embedded behaviour.

2.2 Process Behaviour

The parallel composition of behavioural processes may be expanded into a global behaviour

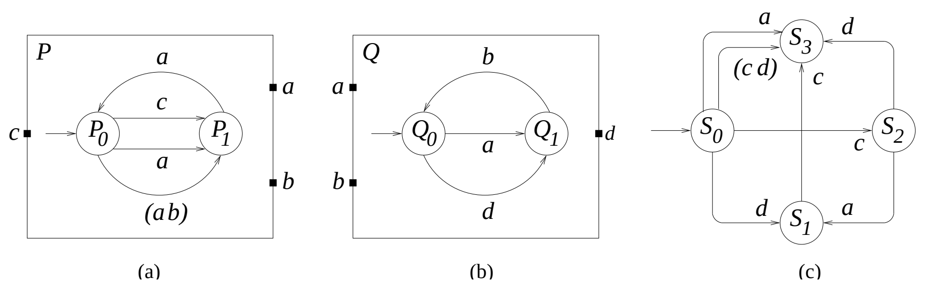

Fig. 2. (a,b) Interfaces and behaviours of P and Q; (c) Behaviour of S, the parallel composition of P and Q.

where every state of the global behaviour is given by the product of its component states.

Given a set of processes, and for each process in a particular state, a set of possible transitions, a combination of transitions *may synchronize* if, and only if, for each action that belongs to the label of at least one transition, if the action does not occur in the label of any transition for some other process in the set, then it does not belong to the sort of that process; here causally independent actions are synchronized. If a combination of transitions may synchronize, then some of these transitions *must synchronize* if and only if there is at least one action in the intersection of their labels; thus identical actions from distinct component processes synchronize. When transitions of different processes synchronize, the label of the transition of the composite process is the union of the labels of all components.

For instance, if we compose processes P and Q of Figures 2(a) and (b), then the transitions labelled a from P_0 to P_1 in P and from Q_0 to Q_1 in Q may synchronize, as may the transitions labelled c from P_0 and d from Q_0. The corresponding transitions of the composite process, which is represented in Figure 2(c), are from $S_0 = P_0 \times Q_0$ to $S_3 = P_1 \times Q_1$, and are labelled a and $(c\,d) = c \cup d$ respectively. The transitions labelled c and d may also occur independently, and thus states $S_1 = P_0 \times Q_1$ and $S_2 = P_1 \times Q_0$ can also be reached from state S_0.

Structural operators permit us to interconnect or compose processes together, as pictured in Figure 2, to “black-box” subsystems of processes by abstracting away internalized communication links, and to instantiate new process objects from generic objects. It is these structural operators that allow us to readily describe hierarchical systems that are constructed from component modules. This is significant for the source language of a compiler for reconfigurable computing since we believe most appropriate applications will be inherently modular and capable of hierarchical description.

2.3 The Circal language

Behavioural features

Termination Δ is a deadlock state from which a process cannot evolve.

Guarding $a\,P$ is a process that synchronizes to perform event a and then behaves as, or evolves to, P, while $(a\,b)\,P$ synchronizes with events a and b simultaneously and then behaves as P.

Choice $P + Q$ is a term that chooses between the actions in process P and those in Q, the choice depending upon the environment in which the process is executed. Usually the choice is mediated through the offering by the environment of a guarding event[1].

State Definition $P \leftarrow Q$ defines process P to have the behaviour of term Q. Process Q is bound to identifier P. This construct allows for the recursive definition of process behaviour.

Structural features

Composition $P * Q$ runs P and Q in parallel, with synchronization occurring over similarly named events. When P and Q share a common event, both must be in a state in which they can accept that event before the event and synchronous state evolution can occur. P and Q may independently respond to events that are unique to their specification. Should such events occur simultaneously, the processes respond independently and simultaneously.

Abstraction $P - a$ hides event set a from P, the actions in a becoming encapsulated and unobservable.

Relabelling $P[a/b]$ replaces references to event b in P with the event named a. This feature is similar to calling procedures with parameter substitution.

Processes, and the events they respond to, are central to process algebras such as Circal, CCS [9], and CSP [10]. Circal differs from most process algebras in that it has a strict interpretation of the response of processes to the simultaneous occurrence of events and is therefore well-suited to modelling synchronous devices such as FPGAs.

2.4 Example

Using the language features of Circal, we can express the definitions of processes P and Q from Figure 2 as follows:

$$P \leftarrow P_0 \tag{1}$$
$$P_0 \leftarrow a\,P_1 + (a\,b)\,P_1 + c\,P_1 \tag{2}$$
$$P_1 \leftarrow a\,P_0 \tag{3}$$

and

$$Q \leftarrow Q_0 \tag{4}$$
$$Q_0 \leftarrow a\,Q_1 + d\,Q_1 \tag{5}$$
$$Q_1 \leftarrow b\,Q_0 \tag{6}$$

Applying the Circal composition law, or alternatively, the transition relation for the Circal operational semantic [8, 11], we obtain the following equations for the composition of processes $S \leftarrow P * Q$:

$$S \leftarrow P_0 * Q_0 \tag{7}$$
$$P_0 * Q_0 \leftarrow a\,P_1 * Q_1 + c\,P_1 * Q_0 + (c\,d)\,P_1 * Q_1 + d\,P_0 * Q_1 \tag{8}$$
$$P_1 * Q_0 \leftarrow a\,P_0 * Q_1 + d\,P_1 * Q_1 \tag{9}$$
$$P_0 * Q_1 \leftarrow c\,P_1 * Q_1 \tag{10}$$
$$P_1 * Q_1 \leftarrow \Delta \tag{11}$$

[1] There is also a *non-determinism* operator used for describing the behaviour of existing systems, but that does not have a role when Circal is used as a design language

3 Overview of the compiler

This paper describes our strategy for implementing systems described in the language HCircal, which is, in effect, Circal without non-determinism, a concept that we do not wish to realize in our implementation.

An HCircal source file consists of a declaration part, a process definition part, and an implementation part. The definition part consists of a sequence of process definitions adhering to the Circal BNF. The implementation part is introduced with the `Implement` declarative and is followed by a comma-delimited list of concurrently active processes that are to be implemented in hardware.

The compiler operates via the following three stages:

1. The user inputs an HCircal specification of the system to be implemented.
2. The compiler analyzes the specification to produce a hardware implementation and a driver program for interacting with the hardware realization that is in the form of a Xilinx XC6200 FPGA configuration bit-stream suitable for loading onto XC6200-based reconfigurable coprocessors such as the SPACE.2 board [23]. The driver program is a C program that executes on the SPACE.2 host (a DEC Alpha UNIX workstation). It loads the configuration onto the coprocessor and allows the user to interact with the implemented system.
3. The user runs the driver program and interacts with the hardware model by entering event traces and observing the system response.

The following sections describe in detail the methodology we have developed to map from behavioural descriptions to technology-independent circuits, the decomposition of the circuits into modules for which FPGA configurations are readily generated, and the derivation of the module parameters from the Circal specification. The generation of the host program is a straightforward specialization of a general program that obtains appropriate event inputs, loads the input registers, and reads the process state registers.

4 A circuit representation of Circal

The aim of the model is to faithfully represent the underlying Circal semantics [11] in hardware. The design concentrates on the representation of the Circal composition operator, which is of central importance because it is through the composition of processes that potentially complex interacting behaviour occurs due to the concurrent execution of the individual processes.

The hardware implementation of the Circal system follows design principles that aim to generate fast circuits quickly. The first of these is that, for the sake of speed and scalability, the hardware representation of Circal aims to minimize its dependence upon global computation at the compilation and execution phases. The second principle is that we choose to design for ease of run-time instantiation and computational speed over area minimality. The motivation for these choices is the desire to leverage the speedup afforded by concurrently executing the Circal system in hardware; they are supported by the ability to reconfigure the gate array at run-time in order to provide a virtual chip of “limitless” circuit area. Finally, we desire a reusable design because we believe that will facilitate design synthesis, circuit reconfiguration, and future investigations into dynamically structured Circal.

4.1 Design outline

A block diagram of a digital circuit that implements a composition of Circal processes in hardware. is shown in Figure 3(a). This circuit consists of a system of interconnected, interacting processes that respond to inputs from the environment by undergoing state transitions.

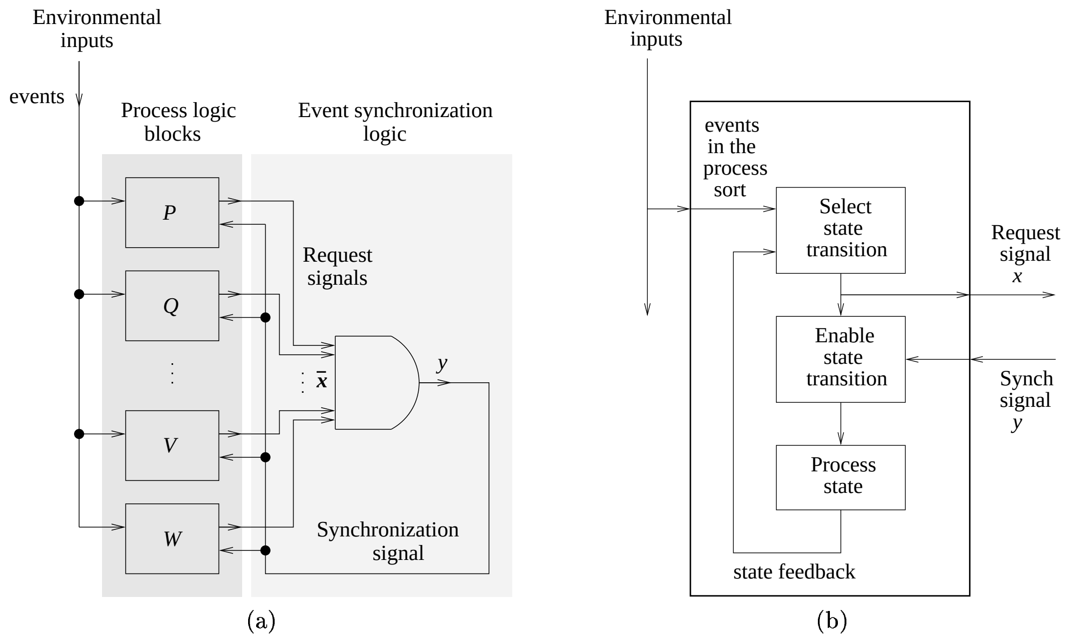

Fig. 3. (a) Circuit block diagram; (b) Circal process logic block.

An individual process is implemented as a block of logic with state. In a given state, each process responds to events according to the Circal process definition. Individual processes examine the event offered by the environment and produce a "request to synchronize" signal, as pictured in Figure 3(b), if the event is found to be acceptable. The conjunction of the request signals generated by all processes in the system produces a synchronization signal that each process responds to independently.

Implementing Circal in synchronous FPGA circuits leads us to assume that: each process is in a known state during each clock period; state transitions occur on clock edges; an event occurs at most once during a clock period; the next state is determined by the events that occurred during the previous clock period; events are sampled on positive clock edges; and, if no event occurs between consecutive positive clock edges, then the idling process transition $P \to P$ occurs upon the second clock edge by default. Since process evolution is dependent upon timed samples of events, the presence of clock ticks is implicit in all guards. Thus we have an implicitly timed system; the modelling of such systems in Circal is discussed in detail in [8, 11].

4.2 Process logic block design

Process logic blocks are derived from the process definition syntax and represented as compact localized blocks of logic to simplify the placement and routing of the system. A high-level view of a process logic block is given in Figure 3(b).

A process is designed to respond to events in the environment that are acceptable to all processes in the composed system. In order to perform this function, the process logic first checks whether the event is acceptable to itself. If all processes find the event acceptable, the event synchronization logic returns a synchronization signal that is used by individual process logic blocks to enable the state transition guarded by the event.

The current state of the process is stored in a process state register using a one-hot encoding. Each state is represented by a flip-flop that becomes active when the process enters that state and remains active while the process remains in that state.

In a given state, each process is prepared to synchronize on some subset of the events in its sort. However, in a composed system, for any event to occur, all processes must agree on the acceptability of the event.

Three steps are needed to agree upon the events that are permitted. Taken together these steps comprise the state renewal cycle and constitute a hardware realization of the operational semantics of the Circal composition operator. In [11], these semantics are presented in terms of a labelled transition system.

1. In the first step, each process independently and in parallel determines whether the combination of events offered by the environment is a valid combination for its current state. This check is performed by determining whether the combination of events that intersects its sort forms a valid guard for the current state. If so, the process flags its willingness to accept the event by raising a request signal. Otherwise the signal is not raised.
2. The second step checks whether the event combination is acceptable to all processes in the composed system by forming the conjunction of the request signals generated during the first step. The result of this calculation is broadcast to all processes in order for them to synchronize their transitions to a new state.
3. In the third step, each process determines its next state based on the combination of events offered by the environment and whether or not all processes accepted the offer. The default is to remain in the current state if no events occurred or the combination of events was not accepted by all processes during the second step.

The following subsections describe the process logic block design in greater detail.

Determining the validity of event combinations We construct a combinational circuit that checks whether the events in the sort of the process form a valid guard for the current state. The process also accepts a null event (an event not in its sort) in order to allow other processes to respond to events it does not care about. The current state of the process is recycled if an unacceptable or null event is offered by the environment.

Let us assume that at most k (recursive) definitions $P_0, P_1, \ldots, P_{k-1}$ are used to describe the evolution of process P with sort $S = \{e_0, e_1, ..., e_{n-1}\}$. Suppose that P_i has the form $P_i \leftarrow g_{i,0}\, P_{i,0} + ... + g_{i,j}\, P_{i,j} + ... + g_{i,m_i}\, P_{i,m_i}$, where each $P_{i,j}$ defines the state P_i evolves to under guard $g_{i,j} \subseteq S$, and $g_{i,j}$ represents the simultaneous occurrence of the events in the set $g_{i,j}$. Note that the definition for P_i consists of $m_i + 1$ guarded terms, whereby the simultaneous events, $g_{i,j}$, are all distinct. Note also that there may be at most k distinct next states but up to $m_i \leq 2^n - 1$ distinct guards. That there can only be k distinct next states is clear from the assumption that we have k definitions. Within any one definition, no pair of distinct next states can have the same guard, since it would then be unclear as to which choice is taken should the corresponding simultaneous event occur. Nevertheless, any subset of the power set of the events contained in the process sort forms a set of distinct guards that may be used in the definition.

If we think of the events and states as *boolean* variables, then in state P_i the process responds to event combinations in the set $\{\gamma_{i,j}\} \cup \{\nu_S\}$, where $\gamma_{i,j} = \varepsilon_0\, \varepsilon_1 \ldots \varepsilon_{n-1}$ is a boolean combination of events in the process sort and each literal corresponding to an event is uncomplemented or not depending upon whether the event is included in the guard or not, that is, $\varepsilon_l = e_l$ or $\varepsilon_l = \overline{e_l}$, for $0 \leq l \leq n-1$, depending upon whether or not $e_l \in g_{i,j}$. Furthermore, $\nu_S = \overline{e_0}\,\overline{e_1}\ldots\overline{e_{n-1}}$ is defined to be the null event for sort S. Process P in state P_i therefore accepts the canonical sum of products boolean expression of events $\nu_S + \sum_{0 \leq j \leq m_i} \gamma_{i,j}$.

The request for synchronization signal, x_P, is thus formed from the expression of accepted events for all states, $x_P = \sum_{0 \leq i \leq k-1} (\nu_S + \sum_{0 \leq j \leq m_i} \gamma_{i,j})\, P_i$, since the parenthesized terms represent the guards accepted in state P_i and the sum is over all possible states.

Checking the acceptability of an event The conjunction of all process request signals is formed and implemented external to the individual process logic blocks. The output of this global AND gate is fed back to each process as the synchronization signal, y. This realization is to be expected since the multiway synchronization of Circal composition is in essence a labelled state transition version of logical conjunction.

Enabling state transitions The state of the process is stored in flip-flops, one for each state. Let $D_{P_l}, 0 \leq l \leq k-1$, denote the boolean input function of the D-type flip-flop for state P_l. Then we can derive the following boolean equation from the process definitions: $D_{P_l} = y\,(\nu_S\, P_l + \sum_{0 \leq i \leq k-1} [\sum_{P_{i,j} = P_l} \gamma_{i,j}]\, P_i) + \overline{y}\, P_l$, for $0 \leq l \leq k-1$.

In the above equation, the terms in parentheses are enabled when the synchronization signal, y, is asserted. The first term within the parentheses is due to state recycling when a null event is offered to the process in state P_l. The double summation within the parentheses corresponds to the guards on state transitions that lead to a next state of P_l from each current state P_i. The last term in the equation forces the current state to be renewed if the system of processes could not accept the event combination offered by the environment. By observing the synchronization signal, the environment can determine whether or not an event was accepted and can thus be constrained by the process composition.

To initialize the process state in a known state, we designate one of the states as the initial state. For FPGA types such as the XC6200, which allows D-type flip-flops to be cleared but not set, the input function for the initial state is complemented and the complement of the output is used as the value for that state.

4.3 The complete process logic block

For a process, the disjunction of the parenthesized terms in the flip-flop input functions implements the same boolean function as that used to obtain the request signal. This is necessarily so since in the latter we need to form all guards for each state, and in the former, we need to combine each guard with the state it occurs in to obtain the correct next state. We therefore use the state selection circuits to form the request signal and use the synchronization signal to enable the selection, thus saving circuit replication.

4.4 Design example

The above Circal implementation strategy may be explained in more detail by extending the example of Section 2.4. The high-level structure of the realization is as in Figure 4.

The system is composed of the logic blocks for processes P and Q. These receive inputs from the environment on event ports corresponding to the union of their sorts, $\{a, b, c, d\}$ in this case. When the environment offers a combination of atomic events, each process

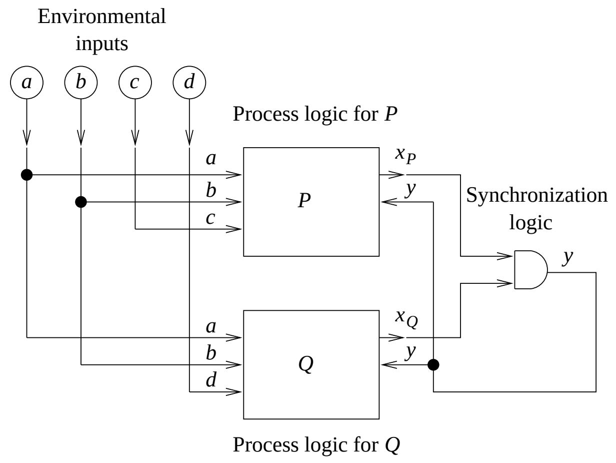

Fig. 4. Outline of composed system.

independently determines whether it can respond to the offer or not. If it can, it asserts its request to synchronize signal, x_P (respectively x_Q). Otherwise the request is not asserted. According to whether or not both processes can accept the offered event combination, each process receives the synchronization signal y, whereupon the process evolves into a new state or recycles the current state.

Determining whether an event combination is valid We construct a combinational circuit that checks whether the subset of the events offered by the environment that intersects the sort of a process forms a valid guard for the current state. Let us reconsider the process P defined by equations (1) – (3):

$$P \leftarrow P_0 \tag{1}$$
$$P_0 \leftarrow a\,P_1 + (a\,b)\,P_1 + c\,P_1 \tag{2}$$
$$P_1 \leftarrow a\,P_0 \tag{3}$$

When the events and states are thought of as boolean variables, then in state P_0 the process responds to any one of the event combinations in the set $\{a\,\overline{b}\,\overline{c}, a\,b\,\overline{c}, \overline{a}\,\overline{b}\,c, \overline{a}\,\overline{b}\,\overline{c}\}$, where the last term corresponds to the null event and we stay in state P_0. In state P_1, the process logic would accept the event expression $(a\,\overline{b}\,\overline{c} + \overline{a}\,\overline{b}\,\overline{c})$.

The request for synchronization signal, x_P, is thus formed from the expressions for both states:

$$x_P = (\overline{a}\,\overline{b}\,\overline{c} + \overline{a}\,\overline{b}\,c + a\,\overline{b}\,\overline{c} + a\,b\,\overline{c})\,P_0 + (\overline{a}\,\overline{b}\,\overline{c} + a\,\overline{b}\,\overline{c})\,P_1. \tag{12}$$

In order to simplify the implementation and reuse terms, we prefer this canonical sum of minterms representation instead of a minimized form.

Checking acceptability of the event The request signals for P and Q are ANDed together. in an AND gate The output of this gate is fed back to each process as the synchronization signal y.

Allowing state transitions Let D_{P_0} and D_{P_1} denote the boolean input functions for the P_0 and P_1 state flip-flops. Then we can derive the following boolean equations from the process definitions

$$D_{P_0} = y\,(\overline{a}\,\overline{b}\,\overline{c}\,P_0 + a\,\overline{b}\,\overline{c}\,P_1) + \overline{y}\,P_0 \tag{13}$$
$$D_{P_1} = y\,(\overline{a}\,\overline{b}\,\overline{c}\,P_1 + [\overline{a}\,\overline{b}\,c + a\,\overline{b}\,\overline{c} + a\,b\,\overline{c}]\,P_0) + \overline{y}\,P_1 \tag{14}$$

In the above equations, the terms in parentheses correspond to the guards on state transitions, and to state recycling if a null event was offered to this process. The last term in the equations forces the current state to be renewed if the processes could not accept the event combination offered by the environment.

We assume flip-flops can be cleared but not set. To initialize the process in P_0, we therefore complement its input function and use the complement of P_0's state output. That is, we form

$$D_{P_0} = \overline{y\,(\overline{a}\,\overline{b}\,\overline{c}\,P_0 + a\,\overline{b}\,\overline{c}\,P_1) + \overline{y}\,P_0} \tag{15}$$

and use the complemented output to represent the state,

$$P_0 = \overline{Q_{P_0}}. \tag{16}$$

The complete process logic block An optimized circuit to implement the logic for process P forms the request signal, x_P, by combining the logic to select the next state with an OR gate. The resulting optimized circuit is shown in Figure 5. The circuit for process Q is easily derived in a similar fashion.

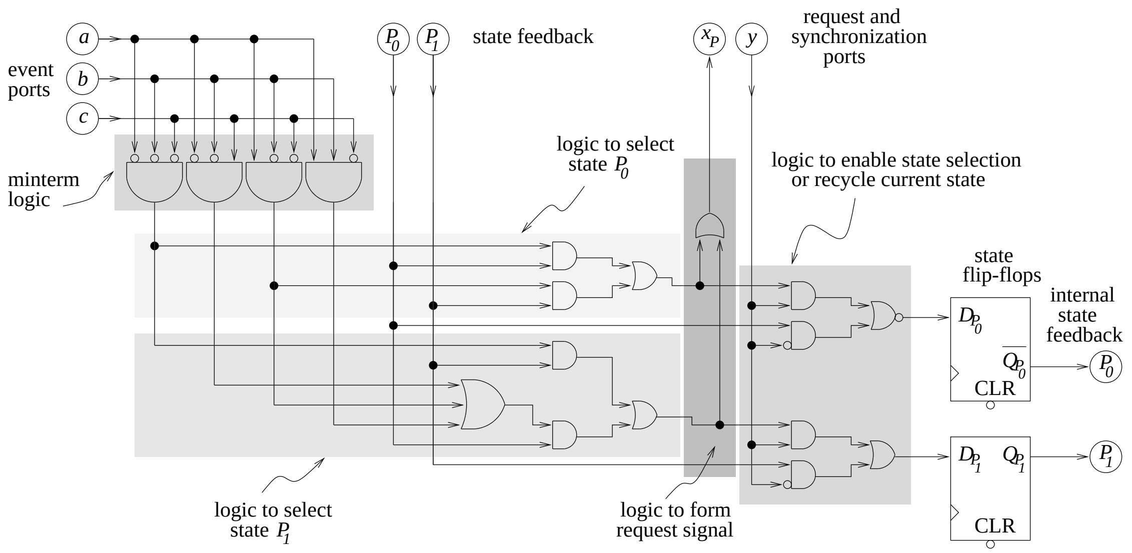
a
b
c
event ports
minterm logic
P_0
P_1
state feedback
x_P
y
request and synchronization ports
logic to select state P_0
logic to enable state selection or recycle current state
state flip-flops
internal state feedback
D_{P_0}
$\overline{Q_{P_0}}$
CLR
D_{P_1}
Q_{P_1}
CLR
logic to select state P_1
logic to form request signal

Fig. 5. Complete process logic for process P.

5 Mapping the circuits to reconfigurable logic

In this section we consider the placement and routing of the circuits derived in Section 4. The derivation of circuit requirements from the specification is discussed in the next section.

Our primary compilation goal is to generate FPGA configurations rapidly. We also want to be able to replace circuitry at run-time to explore changing process behaviours and to overcome resource limitations. A major reason for undertaking this research is that conventional specification, synthesis, and mapping methodologies do not currently support these goals.

Our approach has been to synthesize and map the abstract behavioural specifications embodied in a simple process algebra directly to hardware instead of using current specification and synthesis tools that rely upon non-deterministic, slow, and unsystematic mapping tools. This approach was adopted following difficulties with placing and routing Circal models satisfactorily using XACTstep, the Xilinx automatic place and route tool for XC6200.

When XACTstep was used to place and route the system from Section 4.4 without constraints, the tool flattened the netlist, packed the logic into a compact rectangular region of the chip, and was unsuccessful in completing the routing. The result, which took 43 seconds to produce on a 333MHz Pentium II PC, is captured in Figure 6(a). By contrast, preserving the logic hierarchy did not significantly aid the tool, but routing was completed successfully in 45 seconds. The result is depicted in Figure 6(b). While adequate as a static design, this solution does not satisfy the need to rapidly generate and modify designs. Locating logic and routes for modification is a real problem. We therefore decided to try completely constraining the placement of the logic, but to allow the router to function automatically. The result, which is depicted in Figure 6(c), still took 5 seconds to produce and did not provide the desired predictability and regularity in routing. The ultimate step was to constrain the routing as well, thereby giving the compiler complete control over placement and routing. For the example, the HCircal compiler produced and configured the design of Figure 6(d) in under a second.

We thus found that the performance of the Xilinx XACTstep place and route tool for XC6200 was extremely poor, even for simple circuits, and unless every aspect of the design was constrained, it was impossible to locate components that might later be reconfigured. These difficulties led us to consider decomposing the circuits into modules that can be placed and routed under program control. These modules serve as an attractive intermediate form since they are readily derived from the specification, they completely describe the circuits to be implemented in a hardware-independent manner, and the FPGA configuration can be generated from them without further analysis.

The process logic is represented as parameterized sub-component modules, each of which can be rapidly mapped and placed to a specific location on chip. This approach has the advantage that synthesis and floor-planning are combined at both the intermediate representation and in the physical implementation and results in circuit functions that are highly modularized and localized. As a consequence, sub-components can easily be replaced in a dynamic reconfiguration scheme. Another significant advantage of the approach is that the compilation is deterministic and extremely rapid. We therefore envisage using similar techniques in an on-line interpreter that could manage hardware virtualization and behaviourally inspired dynamic reconfiguration at run time. A drawback of the approach is that some degree of compiler portability is lost.

We distinguish between 10 module types. Inputs are captured by an Environmental Inputs (EI) block and are routed to process logic by Bus (B) and Input Junction (IJ) blocks. Within a process, a series of Minterm (MT) blocks detects the combinations of event inputs a process can respond to. The sets of minterms that lead to particular next states from a given current state are summed in so-called Guard (G) blocks. Associated with each Initial State (IS) and Non-Initial State (NIS) block is a Request (R) block that determines whether any of the Guard block outputs are accepted in the current state. The process request signals are formed from the disjunction of Request block outputs in OR Tree (OT) blocks, and the Synchronization Logic (SL) blocks form the synchronization signal from the individual process request signals.

Each module type implements a particular combinational logic function using a specific spatial arrangement. Modules are specified by giving the exact function to be implemented, e.g., minterm number, input and/or output signal vectors, and their location in the array. To simplify the layout of the circuits, all modules are rectangular in shape and communicate via adjoining ports when they are abutted on the array surface. The module

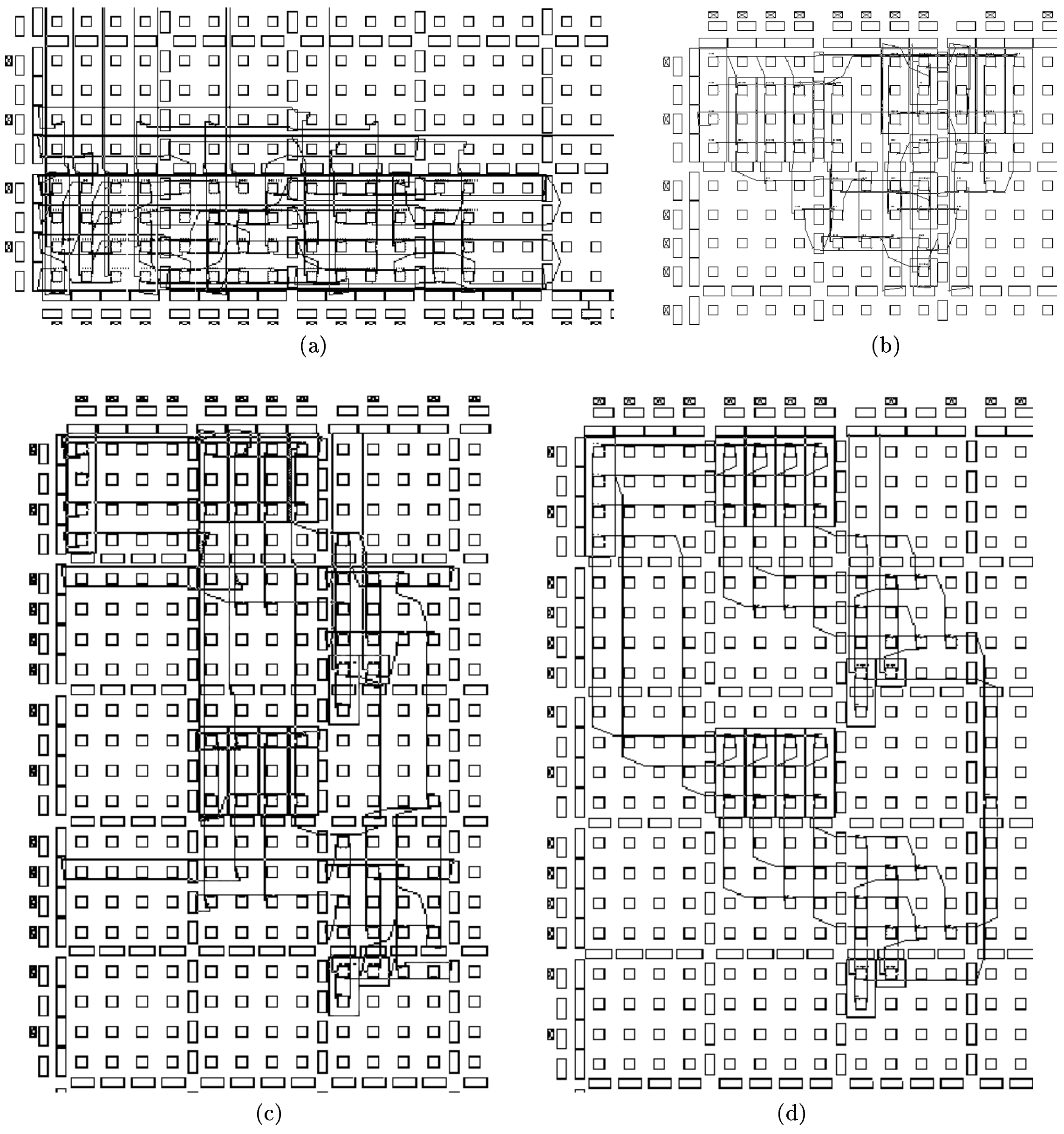

Fig. 6. (a) XACTstep APR of flattened netlist; (b) XACTstep APR with hierarchy preserved; (c) manual placement, XACTstep routing; (d) HCircal place and route.

arrangement for a typical system $P * Q * \ldots$ is depicted in Figure 7. For a complete description of the function, layout, and specification of the modules, please refer to [24].

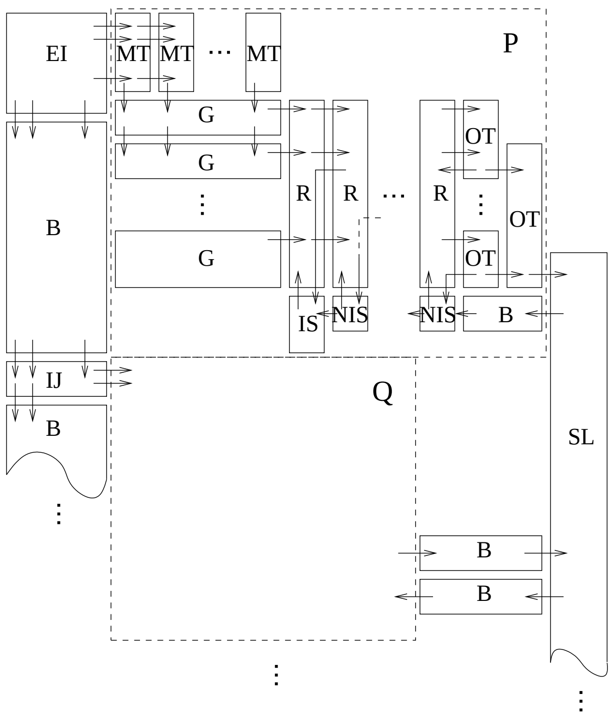

Fig. 7. Typical circuit module arrangement

The compilation approach places and routes the design entirely under program control in time proportional to the area of the circuit. A worst case assessment of the area required by the implementation assumes there are n environmental inputs, p processes, that the sort size of each process is $O(n)$, that the maximum number of minterms for any process is $k \leq 2^n$, and that no process has more than k states. In that case, the width of the design is $O(n + k \log n + \log p) = O(2^n \log n + \log p)$, and the height of the design is $O(p \times (n + k^2)) = O(2^{2n} \times p)$.

The results reported in this paper assume the underlying FPGA architecture is fine-grained (in the case of XC6200, a logic cell implements any 2-input function), and that local cell-to-cell routing is available. Current FPGA architectures, such as the Xilinx Virtex series [25], tend to be more coarse-grained, in that wider functions are supported at the logic cell level, and if they support partial reconfiguration, that also tends to be at a coarser grain size. One impact of these differences on our results are that current FPGAs need fewer cells in general to implement our circuit modules because more logic can be packed into a cell. For example, our designs make use of wide AND and OR gates that in the XC6200 compiler are mapped to balanced trees of 2-input gates. The corresponding functions in Virtex can be mapped to fewer configurable logic blocks (CLBs) since the additional carry logic resources and 4-input lookup tables provided allow 16-input gates to be mapped to a single CLB. Larger gates can be constructed by linearly combining CLBs via the carry chain.

The front-end of the XC6200 HCircal compiler produces a technology-independent description of the specified circuits in terms of a set of circuit module parameters. These modules are instantiated by the back-end in time proportional to their area. We assume that configuration time is proportional to the area of the circuit that is to be configured.

With XC6200 technology, which holds to this assumption, the time to generate the module bitstreams is of the same order as the time to load them. The Virtex FPGA series, on the other hand, supports a model of partial reconfiguration that requires an entire "frame" or column of configuration bits to be read/modified at a time. Configuration costs are thus proportional to the width of the circuit and the height of the FPGA. In order to minimize reconfiguration overheads, the model thus encourages logic to be laid out vertically into columns of logic cells. We are currently investigating the design of module generators that can produce dense mappings for the Virtex series which can also be configured rapidly. These module generators will call on the JBits API to perform device configuration [26].

6 Deriving modules from process descriptions

For each unique process that is to be implemented, a process template that consists of the modules comprising the process logic is constructed. The module parameters for a process template include positional offsets relative to the template origin. Once the size of the logic for each template is known, a copy with absolute offsets (final placement of modules) is made for each process that is to be implemented and the FPGA configuration is generated.

The steps in the derivation of the module representation of the required circuit or its equivalent FPGA configuration are as follows:

1. Identify the processes to be implemented by scanning the `Implement` statement in the specification.
2. For each process to be implemented:
   (a) Determine its sort by collecting the events guarding state transitions that are found by spanning the state tree from the initial state given in the `Implement` statement.
   (b) Compute the minterms needed by expressing the guards on state transitions in normalized form.
   (c) Form the disjunction of minterms that lead to the same next state for each current state. These groupings are formed in the so-called Guard blocks referred to in the previous section.
   (d) Compute the state logic, comprising:
      i. state registers — one flip-flop is allocated per state;
      ii. request logic, that forms the conjunction of the output of logic described in (2c) above with the corresponding current state; and
      iii. state transition logic that enables transition to the appropriate next state if the synchronization signal is asserted.
3. For each process composition:
   (a) Determine the absolute offsets for each process by stacking their logic blocks below one another and leaving room for buses carrying input signals on the left.
   (b) Compute the synchronization logic needed from the absolute process logic offsets.
4. Compute input registers and input broadcast buses from the process offsets and process sorts.

Currently the compilation is performed off-line and the configurations generated are static. In future implementations we plan to experiment with replacing modules at run-time to overcome resource limitations and implement dynamically changing process behaviours. Minor behavioural changes may simply involve replacing minterms or guard modules which could be done very quickly. The regular shapes and small sizes of modules may allow us to distribute them and finalize the module positioning at run-time in order to maximize array utilization.

7 Conclusions

Traditionally the specification of reconfigurable computing applications has utilized inherently structural hardware description languages such as VHDL [27] and Verilog [28], augmented programming languages such as C/C++ [6], Ruby [29, 30], and Lola [31], or schematic design entry, all of which take a low-level view of systems. The research reported here contributes to our understanding of appropriate high-level system description languages that are oriented towards the rapid design of highly concurrent systems. We require appropriate languages that permit both behavioural and structural description, as is done with high-level programming languages that are oriented towards the function of a program rather than its realization on the underlying hardware. Following the traditional programming language approach, suitable compilation techniques are also required if we are to bring reconfigurable computing to a mainstream user community, to make the technology more accessible, and to permit faster prototype turnaround. This will give developers who are experienced in algorithms and applications but who have limited understanding of the underlying hardware ready access to reconfigurable computing.

Specifically, this paper reports on the following results that contribute to the above goal.

We have shown how to realize Circal processes as circuits by developing a compilation technique that takes programs written in an abstract language that is oriented towards the hierarchical, modular description of systems of interacting, concurrently active processes and produces circuit descriptions. These circuits are mapped into concurrently active, synchronized blocks of reconfigurable chip logic that directly exploit the inherent parallelism of the underlying technology. Implementing system components as independent blocks of logic allows them to be generated independently, to be implemented in a distributed fashion, to operate concurrently, and to be swapped to overcome resource limitations. Our approach exploits the hierarchy and modularity inherent in abstract system descriptions given in a process algebra such as Circal.

We have shown how to instantiate a circuit by decomposing it into parametric modules that perform functions above the gate level. To simplify the layout, modules are mapped to rectangular regions that are wired together by abutting them on a chip. Since the modules completely describe the circuits to be implemented in a hardware–independent yet readily mapped manner, they could serve as a mobile description of Circal processes that can be transmitted and instantiated remotely.

The circuit model is readily derived from Circal behavioural specifications. A natural mapping from the circuit model allows modules to be generated quickly; mapping to different technologies should therefore be straightforward. The time to calculate, produce, and configure module logic is proportional to its area. Thus the time to instantiate a circuit is proportional to its area, which is a desirable property since it is the minimum time needed to configure the circuit onto the reconfigurable logic chip surface with current technology.

Current directions extending the work reported in this paper include the development of an interpreter that adapts to resource availability and that supports the run-time compilation of process descriptions whose structure may change dynamically [32]. One such language, called dsCircal, is presented in [13]. This work is targeting the Xilinx Virtex series FPGA and using JBits as the programmer interface. We also intend to assess the usability and descriptive power of process algebraic specifications for a number of applications such as the text filtering application discussed in [33]. This work involves developing strategies for enhancing the HCircal subset of Circal to support specific applications such as stream-oriented and data parallel computations. Our intent is to examine the feasibility of developing application-oriented languages "on top of" HCircal or dsCircal to assist in their rapid realization in reconfigurable logic.

Acknowledgments

We gratefully acknowledge the financial support of the Australian Research Council and thank Antonio Cerone for his contributions to the description of Circal in Section 2.

References

1. VUILLEMIN, J.E., BERTIN, P., RONCIN, D., SHAND, M., TOUATI, H.H., BOUCARD, P.: 'Programmable active memories: Reconfigurable systems come of age,' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1996, **4**, (1), pp. 56-69
2. LEMOINE, E., MERCERON, D.: 'Run time reconfiguration of FPGA for scanning genomic databases,' IEEE Symposium on FPGAs for Custom Computing Machines, FCCM'95, April 1995, (IEEE Computer Society Press, Los Alamitos, 1995), pp. 90-98
3. BUELL, D.A., ARNOLD, J.M., KLEINFELDER, W.J. eds.: 'Splash 2: FPGAs in a Custom Computing Machine,' (IEEE Computer Society Press, Los Alamitos, 1996)
4. ARMSTRONG, J.R., GRAY, F.G.: 'VHDL Design Representation and Synthesis,' (Prentice Hall, Upper Saddle River, 2000), 2nd edn.
5. PALNITKAR, S.: 'Verilog HDL: A Guide to Digital Design and Synthesis,' (Sun Soft Press, Mountain View, 1996)
6. BERTIN, P., TOUATI, H.: 'PAM programming environments: Practice and experience,' IEEE Workshop on FPGAs for Custom Computing Machines, FCCM'94, April 1994, (IEEE Computer Society Press, Los Alamitos, 1994), pp. 133-138
7. SNIDER, G., SHACKLEFORD, B., CARTER, R.J.: 'Attacking the semantic gap between application programming languages and configurable hardware,' Ninth International Symposium on Field Programmable Gate Arrays, FPGA'01, February 2001, (ACM Press, New York, 2001), pp. 115-124
8. MILNE, G.: 'CIRCAL and the representation of communication, concurrency and time,' *ACM Trans. Program. Lang. Syst.*, 1985, **7**, (2), pp. 270-298
9. MILNER, R.: 'Communication and Concurrency,' (Prentice-Hall, New York, 1989)
10. HOARE, C.A.R.: 'Communicating Sequential Processes,' (Prentice-Hall, Englewood Cliffs, 1985)
11. MILNE, G.: 'Formal Specification and Verification of Digital Systems,' (McGraw-Hill, London, 1994)
12. XILINX: 'XC6200 Field Programmable Gate Arrays' Xilinx, Inc., April 1997.
13. MILNE, G.: 'A model for dynamic adaptation in reconfigurable hardware systems,' Proceedings of the First NASA/DoD Workshop on Evolvable Hardware, July 1999, (IEEE Computer Society Press, Los Alamitos, 1999), pp. 161-169
14. SHAW, P., MILNE, G.: 'A highly parallel FPGA–based machine and its formal verification,' Second International Workshop on Field–Programmable Logic and Applications, FPL'92, September 1992, LNCS volume 705, (Springer-Verlag, Berlin, 1992), pp. 162-173
15. PAGE, I., LUK, W.: 'Compiling Occam into FPGAs,' Oxford 1991 International Workshop on Field Programmable Logic and Applications, FPL'91, (Abingdon EE&CS Books, Abingdon, 1991), pp. 271-283
16. JIFENG, H., PAGE, I., BOWEN, J.: 'Towards a provably correct hardware implementation of Occam.' Correct Hardware Design and Verification Methods, IFIPWG10.2 Advanced Research Working Conference, CHARME'93, May 1993, LNCS volume 683, (Springer-Verlag, Berlin, 1993), pp. 214-225
17. KIRSCHBAUM, A., RENNER, F.M., WILMES, A., GLESNER, M.: 'Rapid-prototyping of a CAN-Bus controller: A case study,' Seventh IEEE International Workshop on Rapid System Prototyping, June 1996, (IEEE Computer Society Press, Los Alamitos, 1996), pp. 146-151
18. FUMMI, F., SCIUTO, D.: 'A hierarchical test generation approach for large controllers,' *IEEE Trans. Comput.*, 2000, **49**, (4), pp. 289-302
19. DIESSEL, O., MILNE, G.: 'Compiling process algebraic descriptions into reconfigurable logic,' Reconfigurable Architectures Workshop 2000, RAW 2000, April 2000, LNCS volume 1800, (Springer-Verlag, Berlin, 2000), pp. 916-923
20. CERONE, A., MILNE, G.: 'A methodology for the formal analysis of asynchronous micropipelines,' International Conference on Formal Methods in Computer-Aided Design, FMCAD'00, LNCS volume 1954 (Springer-Verlag, Berlin, 2000), pp. 246-262
21. BAILEY, A., McCASKILL, G.A., MILNE, G.: 'An exercise in the automatic verification of asynchronous designs,' *Form. Methods Syst. Des.*, 1994, **4**, (3), pp. 213-242
22. BOLOGNESI, T., BRINKSMA, E.: 'Introduction to the ISO specification language LOTOS,' *Comput. Netw. ISDN Syst.*, 1987, **14**, (1), pp. 25-59
23. GUNTHER, B.K.: 'SPACE 2 as a reconfigurable stream processor,' 4th Australasian Conference on Parallel and Real–Time Systems, PART'97, September 1997, (Springer, Singapore, 1997), pp. 286-297
24. DIESSEL, O., MILNE, G.: 'HCircal: A hardware compiler for Circal,' Technical report ACRC–00–013, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 2000
25. XILINX: 'Virtex 2.5V Field Programmable Gate Arrays,' Xilinx, Inc., 2000
26. GUCCIONE, S.A., LEVI, D.: 'XBI: A java-based interface to FPGA hardware,' Configurable Computing: Technology and Applications, Proc. SPIE 3526, November 1998, pp. 97-102
27. BLIGHT, D.C., McLEOD, R.D.: 'VHDL for FPGA design,' Oxford 1991 International Workshop on Field Programmable Logic and Applications, FPL'91, (Abingdon EE&CS Books, Abingdon, 1991), pp. 246-254

28. SODERMAN, D., PANCHUL, Y.: 'Implementing C algorithms in reconfigurable hardware using *c2verilog*,' 6th Annual IEEE Symposium on FPGAs for Custom Computing Machines, FCCM'98, April 1998, (IEEE Computer Society Press, Los Alamitos, 1998), pp. 339-342
29. GUO, S., LUK, W.: 'Compiling Ruby into FPGAs,' Field–Programmable Logic and Applications, 5th International Workshop, FPL'95, August 1995, (Springer–Verlag, Berlin, 1995), pp. 188-197
30. SINGH, S.: 'Architectural descriptions for FPGA circuits,' IEEE Symposium on FPGAs for Custom Computing Machines, FCCM'95, April 1995, (IEEE Computer Society Press, Los Alamitos, 1995), pp. 145-154
31. WIRTH, N.: 'Hardware compilation: Translating programs into circuits,' *Computer*, 1998, **31**, (6), pp. 25-31
32. DIESSEL, O., MILNE, G.: 'Behavioural language compilation with virtual hardware management,' 10th International Workshop Field–Programmable Logic and Applications, FPL 2000, LNCS volume 1896, (Springer–Verlag, Berlin, 2000), pp. 707-717
33. GUNTHER, B.K., MILNE, G,J., NARASIMHAN, V.L.: 'Assessing document relevance with run–time reconfigurable machines,' IEEE Symposium on FPGAs for Custom Computing Machines, FCCM'96, April 1996, (IEEE Computer Society Press, Los Alamitos, 1996), pp. 10-17