Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 162 (2006) 249–254
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Modelling Dynamically Changing Hardware Structure

George J Milne[1]

School of Computer Science and Software Engineering
The University of Western Australia
M002/35 Stirling Highway, Crawley WA 6009, Australia

Abstract

Techniques for modelling reconfigurable computing hardware using a process algebra are given. Dynamically changing hardware is modelled using a process algebra with dynamic sorts, where the sort changes through time. A programming technique based on this dynamic structure process algebra is outlined.

Keywords: process algebra, reconfigurable computing, dynamic structure hardware, circuit modelling

1 Introduction

Digital hardware is a natural application domain for process algebra given that circuits exhibit inherently concurrent behaviour, possibly massive concurrency. The impetus for modelling and verifying digital hardware resulted in the development of a process calculus with features influenced by concepts inherent in digital logic. This exercise was heavily influenced by the work of Robin Milner, on CCS [11] and its precursor [10], and resulted in the development of the circuit calculus known as Circal [6].

In 1990 a number of us at the University of Strathclyde in Glasgow wished to use field programmable gate arrays (FPGAs) as the enabling technology for a new computer architecture which could be programmed as a custom machine for simulating spatially distributed, highly concurrent applications. The underlying idea was fairly straightforward; we would map systems which exhibited large amounts of fine-grained concurrency and local interconnections onto an FPGA-based architecture. The resulting configuration provided us with a customised hardware simulator for physical systems, such as fluid flow, with a 1-1 correspondence between physical

[1] Email: george@csse.uwa.edu.au

components and an area of digital logic which implements its model. Such architectures are now known as *reconfigurable computers* e.g. [9,1]. It was felt that Circal was a suitable language with which to *prescribe* a system to be realised as hardware, rather than as a language to model previously designed circuitry. Process algebra allows us to describe systems of interacting automata, with each automaton being described by a process. This is the basis of techniques for compiling from process algebra to FPGA logic, described in [3,2]. The appropriateness of using process algebra to programme reconfigurable computers is examined in [5].

Reconfigurable computing differs from the classical von Neumann computing paradigm in that a program does not reside in memory but rather an application is realised directly in digital logic. As this logic is repeatedly programmable then the underlying FPGA platform may be instantiated to create different custom computer realisations for each distinct application. Furthermore, certain FPGA technologies are *dynamically reconfigurable* in that part of the FPGA logic may be reconfigured while another part is running. That is, our programmable hardware can be modified as it runs and, furthermore, it has the potential to be *self-modifying*, a concept which has traditionally been considered anathema in the software world, as described by Tony Hoare [4]. Dynamic reconfiguration thus allows the hardware structure to change at run-time, in contrast to traditional computing systems with fixed structure. The need to programme dynamically changing hardware structure has then lead to the introduction of *dynamic structure Circal* (dsCircal) [8].

Circal is *static* in that we may only model systems with a fixed structure; this is adequate for modelling many types of systems, such as digital hardware [7] and even road networks [9]. The notion of a fixed, static *sort* has been extensively used to represent the structure of concurrent systems with a fixed topology. The concept was introduced in [10], with this early work leading to the development of both CCS and Circal. The notion of sort is significant in Circal since the Circal composition operator is dependent on the sort of its operands.

A system of processes composed together adopts the structural synthesis convention that similarly named ports will link together. Processes change state following the synchronised occurrence of actions over all similarly named ports. Thus we have state changes caused by the occurrence of actions which guard the new states. We extend this concept to one where guarding actions can also control the change in process structure. That is, not just into a new state retaining the same structure, but into a new process with quite distinct structure. The structure can therefore change through time under the controlling influence of other processes in the environment. This is the concept of *dynamic sort*. Dynamic structure Circal then differs from Circal in that it allows the sort of a process to (possibly) evolve through time under the control of suitable actions. The sort of a process is explicitly represented as a vector of labels which is juxta-positioned with the corresponding process identifier, as with a simple process with static structure defined by:

$$P\,\langle a, b\rangle \overset{\text{def}}{=} aP_1 + bP_2 \tag{1}$$

Here we assume that renewal processes P_1 and P_2, which are defined elsewhere, have the same static sort, namely $\langle a, b\rangle$. But suppose we have a process named Q which "starts" with the same sort but which after event b changes into a process with sort $\langle c, d, e\rangle$, then we may have:

$$Q\,\langle a, b\rangle \overset{\text{def}}{=} aQ_1 + bQ_2\,\langle c, d, e\rangle \tag{2}$$

where

$$Q_2\,\langle c, d, e\rangle \overset{\text{def}}{=} (c, d)\Delta\phi + eQ\,\langle a, b\rangle \tag{3}$$

This captures a process Q_2 which responds to a simultaneous action (c, d) by becoming extinct (the null process Δ with the empty sort) and to an e event by recursing back to Q.

2 A Simple Dynamic Structure

Given a system of three processes A, B and C defined by:

$$A\,\langle a, p\rangle \overset{\text{def}}{=} aA + pA \tag{4}$$

where

$$B\,\langle p, t_1\rangle \overset{\text{def}}{=} pB + t_1B_1\,\langle q, t_2\rangle \tag{5}$$

$$B_1\,\langle q, t_2\rangle \overset{\text{def}}{=} qB_1 + t_2B\,\langle p, t_1\rangle \tag{6}$$

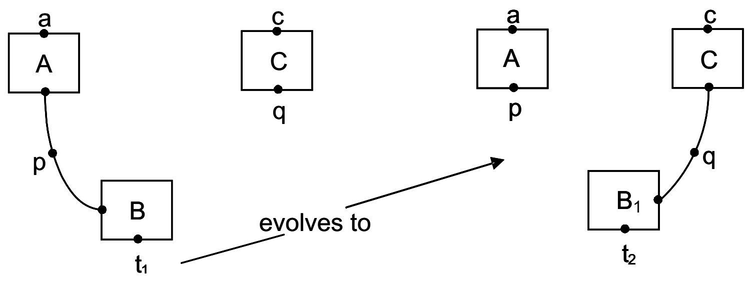
$$C\,\langle c, q\rangle \overset{\text{def}}{=} cC_1 + qC \tag{7}$$

A system constructed from these three processes may then be defined as follows:

$$SYS\,\langle a, p, c, q, t_1\rangle \overset{\text{def}}{=} A\,\langle a, p\rangle * B\,\langle p, t_1\rangle * C\,\langle c, q\rangle \tag{8}$$

Processes A and C have static structure, that is, their sort remains fixed through time. Only process B has dynamic sort which can change (initially) as a result of an externally generated action on port t_1. Action t_1, therefore acts as a *trigger action* causing process B to evolve into process B_1, whose sort is different. Process B_1 can react to action t_2, also generated externally, and toggle back to its original incarnation as process B. The change of structure, caused by trigger actions on process B, causes the structure of the system SYS to change. As B evolves into

B_1, as a consequence of action t_1, the p link disappears simultaneously with the appearance of a q link. Action t_2 in B_2 causes the reverse to occur. This evolution between two distinct system structures, and its potential to evolve back to the original configuration, is pictured in the following figure:



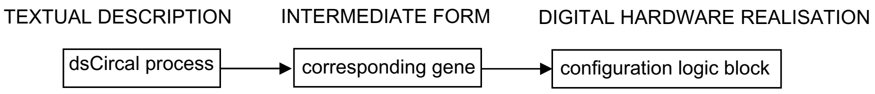
Notice that while our system has dynamic structure, this is in terms of changing interconnect over a fixed number of processes, three in this case. For dynamic reconfiguration in hardware we also require process creation, extinction and possibly mobility and so also the creation and destruction of their interconnecting links. The creation and destruction of processes may also take place under the control of actions occurring on specific control ports.

3 Programming Reconfigurable Hardware

In von Neumann computation the sequential behaviour of an executing processor generally manipulates input data into output data by following a sequence of activities described by the programming language. The program, its compiler and the design of the microprocessor which ultimately determines the behaviour of the executing code. In a concurrent world, one where computation is effected by a direct mapping into programmable digital logic, a similar situation also exists. In this case we may also use a language to describe computation with the language directly supporting the concurrent interactions over dynamic structures, as captured by the dsCircal syntax. Again code will execute but rather than imagine dsCircal fragments being compiled and executed on sequential computers, we will focus on the realisation of the encoded behaviour directly in digital logic

Blocks of FPGA logic cells will be allocated and configured to perform the functional behaviour of a fixed structure; that is, the behaviour captured by interconnected processes which do not change their structure for a given period of time. Hence they retain a fixed sort for that period of time. During that period quite distinct behaviour may be realised as the process evolves through a finite set of states. However, any interaction between this process and its environment will occur though the fixed set of ports denoted by its sort.

When our processes have dynamic sort then we can have a process P which changes into another Q with a totally different structure, following the occurrence of a guarding action. We may realise the behaviour encoded in such a dsCircal process as follows:



We may imagine that available FPGA logic is limited and that only the configuration block for process P may be realised. An encoding of sub-process Q will reside in a suitably identified memory location. When the enabling action occurs causing process P to evolve to the structurally (and behaviourally) distinct process Q, the configuration realising Q will be constructed from the encoding residing in memory. This new configuration will overlay the configuration block for P, hence our machine will be dynamically reconfigured as behaviour passes from P to Q with process P itself causing the reconfiguration as it evolves from P into Q. The data necessary to reconfigure a block of FPGA cells consists of a stream or sequence of bits. This bit stream is loaded into RAM where each memory location controls a corresponding part of the programmable gate array of the FPGA. This is used to select the logical functionality of the underlying, primitive configurable unit, often a gate or simple ALU, while also selecting the interconnectivity between such configurable logic building blocks. As with dsCircal, both the set of primitive agents, the configurable logic blocks, and their connective structure is dynamic and can change through time.

The process encoding residing at a location is known as a *process gene*, or *gene* for short. Just as a strand of DNA encodes or maps how proteins are constructed from sequences of amino acids, so our process gene encodes how the hardware building blocks of our underlying FPGA technology are constructed such that when they become active they realise the desired behaviour. The analogy with genetics is even stronger since DNA also encodes temporal information in that only part of a DNA strand may be responsible for the construction of a current protein and this protein may change under the control of the DNA encoding. The particular choice and ordering of occurrence of the amino acids determines how they combine and hence determines the structure of the constructed protein.

The compilation from process expression to realisation in FPGA hardware is pictured as follows:

A *process generator* function takes a process encoding gene and produces the hardware which directly realises the corresponding process behaviour.

Genes are active data and when interpreted result in an object, such as a block of configurable FPGA logic, which exhibits active behaviour. The ability to store genes at a location and to fetch a copy of the gene on demand permits us to communicate process encodings. Thus the active digital hardware implementations of two processes may communicate a third process from one to the other, by sending its identifier name to the receiving process. This identifier name is then used by the receiver to "nlock the corresponding location" and extract the contained gene. This gene may then be used by the receiver and a process generator to produce the implementation of the communicated process as a configuration logic block. This active data, namely the hardware realisation of the communicated process, will then be capable of execution as required.

References

[1] Barrie, P., P. Cockshott, G. Milne and P. Shaw, *Design and verification of a highly concurrent machine*, Microprocessors and Microsystems **16** (1992), pp. 115–123.

[2] Diessel, O. and G. Milne, *Behavioural language compilation with virtual hardware management.*, in: R. Hartenstein and H. Grunbacher, editors, *10th International Workshop on Field Programmable Logic and Applications (FPL 2000), Villach, Austria*, Lecture Notes in Computer Science **1896** (2000), pp. 707–717.

[3] Diessel, O. and G. Milne, *A hardware compiler realising concurrent processes in reconfigurable logic*, IEE Proceedings - Computers and Digital Techniques **148** (2001), pp. 152–162.

[4] Hoare, C., *Hints on programming language design*, Technical Report STAN-CS-73-403, Standford Artificial Intelligence Laboratory, Stanford University, Stanford (1973).

[5] Lee, G. and G. Milne, *Programming paradigms for reconfigurable computing*, Microprocessors and Microsystems **29** (2005), pp. 435–450.

[6] Milne, G., *CIRCAL and the representation of communication, concurrency and time*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 270–298.

[7] Milne, G., "The Formal Specification and Verification of Digital Systems," McGraw-Hill Company, Europe, 1994.

[8] Milne, G., *A model for dynamic adaptation in reconfigurable hardware systems*, in: A. Stoica, editor, *First NASA/DOD Workshop on Evolvable Hardware, July 19-21, 1999. Pasadena California. IEEE Computer Society Press*, 1999, pp. 161–169.

[9] Milne, G., P. Cockshott, G. McCaskill and P. Barrie, *Realising massively concurrent systems on the SPACE machines*, in: *Proceedings IEEE Workshop on FGPA's for Custom Computing Machines, IEEE Computer Society Press* (1993), pp. 26–32.

[10] Milne, G. and R. Milner, *Concurrent processes and their syntax*, Journal of the ACM **26** (1979), pp. 302–321.

[11] Milner, R., "Communication and Concurrency," International Series in Computer Science, Prentice-Hall International, New York, 1989.