SCCircal: a Static Compiler Mapping XCircal to Virtex FPGAs

Jérémie Detrey

ENS Lyon
46, allée d'Italie
69 364 Lyon cedex 07, France

jdetrey@ens-lyon.fr

Oliver Diessel

School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia

odiessel@cse.unsw.edu.au

UNSW-CSE-TR-0213
August, 23 2002

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia

Abstract

This paper describes the new version of SCCircal, a static compiler for XCircal targeted to Xilinx Virtex architecture. This compiler, written in Java, is now capable of providing a real FPGA implementation for almost any Circal process specification. Thus it supports hierarchy, abstraction and relabelling. This paper also introduces the notion of a process interface, provided to help the development of further extensions of this compiler.

1 Introduction

The Circal process algebra was developped by G. Milne [4], being mainly targeted to control-oriented applications and system verification. Indeed, Circal offers a simple and efficient way to describe systems and their inherent notions of synchronisation and concurrency, along with some rules that allow the system to be exhaustively checked against given specfications.

Then, together with O. Diessel, they developed a compiler [2] that could map Circal specifications onto Xilinx XC6200 FPGAs: the resulting core, fed with the external events, fully simulates the Circal process. But this version of the compiler only implemented a subset of the Circal process algebra, and thus was not able to represent some useful features such as abstraction or hierarchy.

Later, this compiler was ported to Xilinx Virtex architecture by K. So [5], using the JBits API. But this work was more aimed at porting the exisiting XC6200 compiler rather than extending it, and therefore, it still lacks some Circal features. Moreover, given that only a subset of the process algebra was implemented, the specification file format was not suitable for future extensions.

This paper presents the new version of the compiler, that is more or less an extension of the previous compiler. In order to provide better ways of specifying processes, the front-end was also extended, with a support for XCircal programs (see section 2 for further deatils about XCircal), XCircal being the format used by the software version of the Circal System. Our new compiler now supports most of the features offered by Circal, as described in section 7 of this paper.

2 The XCircal description language

2.1 Brief overview of Circal

The Circal process algebra offers some constructs for manipulating processes, that are fully described in [4]. However, here is a short list of those operators:

- Termination: Δ is a deadlocked process, that cannot evolve.
- Guarding: given a process P and a non-empty set of events $\mathcal{S}$, $\mathcal{S}\ P$ is a process that synchronises to performs all the events of $\mathcal{S}$ and then behaves as P.
- Choice: given two processes P and Q, $P + Q$ is a process that can behave either as P or as Q, depending on the environment.
- Non-determinism: given two processes P and Q, $P\&Q$ is a process that also can behave either as P or as Q, but here, the choice depends only on the process itself, and is independent of the environment.
- Composition: given two processes P and Q, $P * Q$ is a process that runs P and Q in parallel, with synchronisation occurring for shared events.
- Abstraction: given a process P and event set $\mathcal{S}$, $P - \mathcal{S}$ is a process that behaves as P, but also encapsulates the events of $\mathcal{S}$, that are then hidden from an external point of view.
- Relabelling: given a process P and two events a and b, $P[a/b]$ behaves as P, except that all occurences of event b are replaced by event a.
- Definition: given a process Q and an identifier P, $P \leftarrow Q$ defines P to have behaviour of Q, thus allowing recursive definitions.

2.2 XCircal: Circal embedded in XTC statements

XTC is a language developed by G.A. McCaskill [3]. It is mainly derived from C and C++, except that it is an interpreted language and it does not offer support for pointer arithmetic. Its main goal is to embed "interface languages", such as Circal for example.

Thus, the XCircal language is in fact an XTC implementation of the Circal description language, and its constructs were directly translated into XTC operators (such as `/\` for Δ, `<-` for $\leftarrow$, ...).

3 Overview of the compiler

The main design flow, from the input XCircal specification file to the output bitstream, is divided into three stages:

1. Parsing of the XCircal file: the parser reads the XCircal program, and generates an abstract syntax tree (AST) that is only an internal representation of the program.

2. Interpretation of the AST: the interpreter creates a contextual environment in which variables, functions and types are stored according to their scope, and each `implement` line adds the specified process to the implementation list of the specification.

3. Bitstream generation: each process of the implementation list is then recursively implemented in the bitstream, along with the blocks used to "interact" with the environment.

The compiler also produces a "state locations" `.loc` file, where it records the list of environmental events and the locations of all state registers on the chip. This file is only used by the testbench programs, that can thereby directly probe the status of the states, without having to parse and interpret the XCircal program each time, or having to have any knowledge of the global layout mapped to the FPGA.

The sections 4, 5, 6 and 7 of this paper detail more precisely the different stages of the compiler pipeline.

4 XCircal parser

As mentioned previously, the parser is the first stage of the compiler, and its goal is to parse the input XCircal file, and to generate an AST from it.

This parser was written in Java, using the JavaCC (Java Compiler Compiler) tool, to be able to specify the XCircal grammar more easily.

But given the limitations of JavaCC and the complexity of the XCircal grammar, we had to slightly modify the original language to avoid non-trivial ambiguous situations. In order to achieve this without altering too much the XCircal original grammar, we have only added the end-of-line semi-colons. In this way, only a few changes are required to convert an XCircal file into a format the parser will accept.

Of course, this parser is temporary, and although it suits our current needs, one of our projects is to adapt the current implementation of such a parser written in C by G.A. McCaskill [3], using the Java Native Interface (JNI).

5 XCircal interpreter

The interpreter is only a small layer above the parser, that linearly reads the AST, remembering the definitions of types, functions and variables and the scope of those definitions.

XCircal being an imperative language derived from XTC, itself derived from C, it has a fairly straightforward interpretation, and the elimination of pointer arithmetic spares us most of the tricky problems a C interpreter would have had to face.

In order to fit the requirements to see XCircal as a description language, it has been extended with an `implement` statement, that takes a process as a parameter, and allows the programmer to specify which processes he wants to implement. In this way, the `implement` statement only add a reference to the specifed process in the implementation list of the specification.

Each implemented process expression is then normalised and analysed in order to simplify its mapping onto the FPGA. This stage along with the internal format are detailed in next section.

6 Description of internal representation of processes

To represent processes internally, we adopted the natural oriented graph approach. Thus, each process is represented as an oriented graph, each node being a deadlock process, the guarding of a process, the choice between several processes, the composition of several processes or the abstraction of some events. Using this representation simplifies the task of the interpreter which simply applies the corresponding operator to the processes.

For example, the representation of the process S, defined as follows, is given in Figure 1.

$$
\begin{aligned}
S &\leftarrow (P * Q) - c \\
P &\leftarrow a\ P' + (a\ c)\ P \\
P' &\leftarrow c\ P \\
Q &\leftarrow b\ Q' + (b\ c)\ Q \\
Q' &\leftarrow c\ Q
\end{aligned}
$$

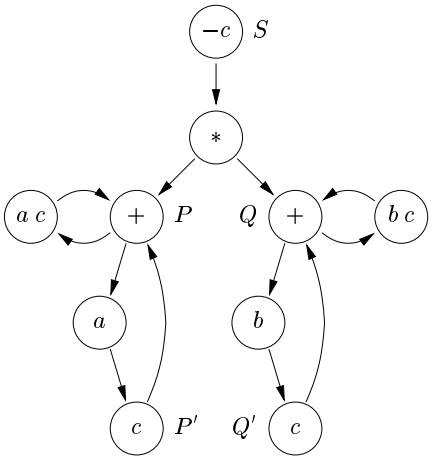

Figure 1: Oriented graph representation of the example process S

Moreover, the graph structure allows us to analyse it using classic depth-first or breadth-first traversing algorithms. For example, we can easily normalise each implemented process by merging similar nodes, or we can collect data to identify “finite state machine like” (FSM) processes. These analysis stages allow the compiler to simplify the graph in order to match some predefined patterns that will be mapped to the FPGA afterwards. Therefore, they are dependant on the mapping capabilities of the compiler and the features it offers. Thus, in order to add functionalities to the mapping stage, one will certainly have to slightly alter the normalisation and analysis phases to reflect the new changes.

7 Mapping the design to Virtex FPGAs

This phase of the design flow was in fact mainly adapted from K. So’s work [5] that targetted a Virtex XCV300 embedded in an Annapolis Microsystems Wildcard (Cardbus coprocessor). While some aspects of this design are Wildcard specific, it can easily be adapted to other Virtex based systems.

For the sake of simplicity, we will only focus on the new features that were added, assuming the reader is already familiar with the previous notions and fully understands them.

7.1 Overall layout

The overall layout of the design is pretty close to the original one, as depicted in Figure 2.

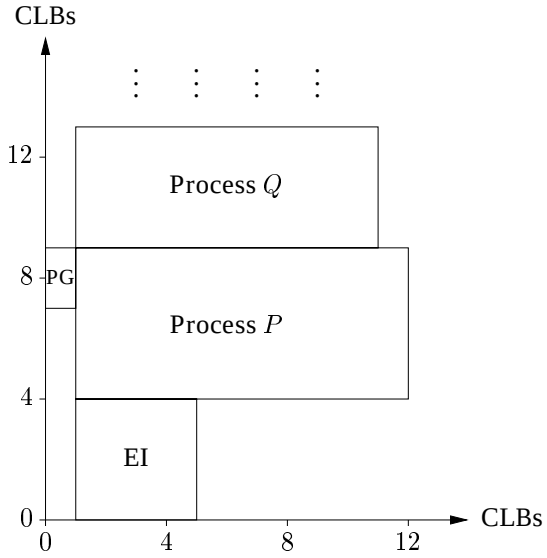

Figure 2: Overall layout of the desgin

Thus, the pulse generation (PG) module (needed by the Wildcard which does not support pulsing the FPGA chip) is still here to provide correct synchronisation between event loading and process transitions, allowing user interaction with the core. Of course, this synchronisation is incompatible with the intrinsically asynchronous aspect of the Circal process algebra, but given the fact that we are still at an early stage of development, synchronisation provides us with a simple solution, until we find a better method to overcome this problem.

The interface of the FPGA with the environment and the polling of events is embedded in the environmental inputs (EI) block. This block in fact buffers the 32-bit long input word (one bit per event) until the process transitions are triggered, and then another input word is read from the Wildcard LAD bus.

The only difference with the previous version of the compiler is that the event signals buses (ESB) and input junctions (IJ) have been eliminated as explicit structures. In fact, they are now embedded in each process block, that have to provide correct forwarding of the inputs to the next process block. Therefore, the process blocks are now stacked directly above the EI block.

A detailed description of these blocks is given in the next section.

7.2 Abstract process interface

Given the increasing number of different types of processes and the desire to represent hierarchical definitions, we have decided to define an abstract interface that is suited to any process, thus allowing us to recursively map processes to CLBs without having to worry about hierarchical compatibility issues.

This interface consists of a set of input and output ports, their location being completely arbitrary as the routing of the core is performed by the JBits router, JRoute. However it is recommended to observe the given locations of the ports while designing a new implementation of this interface, as this will greatly simplify the task of the router.

Figure 3 depicts this abstract interface.

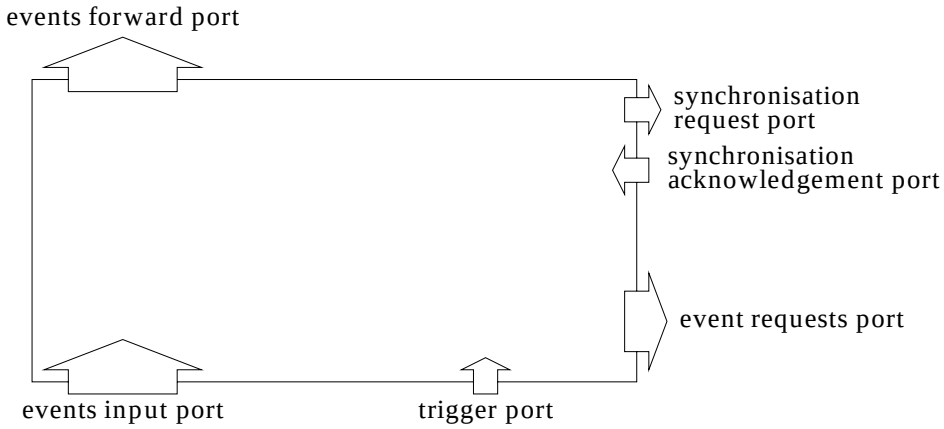

Figure 3: Abstract process interface

The role of these six ports is as follows:

- The events input port is connected to the events bus, and drives the wires of all current events, into the process block. Each of these events can then be used by the process, if it is in its sort, or is forwarded to the next process.
- The events forward port is therefore the port through which the events bus is forwarded. It has to drive exactly the same events as the events input port does, and in the same order. In this way, we can be sure that all sibling processes will receive the same input.
- The event requests port is an output port through which signals requesting the assertion of an event abstracted at a higher level are driven. Indeed, extending the idea submitted

by O. Diessel and G. Milne in [1], abstracted events will be asserted when all the processes that have them in their sort are in a state that can accept these events. The assertion of the abstracted events is then controlled by request signals that are driven through this port.

- The synchronisation request port is a single wire wide port, driving the synchronisation request signal to a higher level process. This signal states that the current process is ready to perform a transition, but is still waiting for synchronisation from its siblings.
- The synchronisation acknowledgement port then drives the answer to the request back to the process: one wire signaling whether the process can perform its transition or not. If the process does not need any synchronisation with other processes, this port will be directly connected with the synchronisation request port: the process will receive an acknowledgement each time it requests synchronisation.
- The trigger port is used for hierarchical needs, for example when a process evolves into a composed process. This issue is discussed in detail in section 7.6.

Of course, this is only an interface, and it does not actually describe the real logic embedded in the different process blocks. These points, along with the description of the four types of process blocks are described in the following sections.

7.3 “Finite state machine like” process block

7.3.1 Identifying FSM processes

An FSM process is a choice node, all its children being guarding nodes, all its grandchildren being choice nodes as well. An FSM process is nothing but a finite state machine, each choice node of the graph being a state and each guarding node representing a transition from one state to another.

7.3.2 FPGA mapping layout

Figure 4 illustrates the layout of the various modules used in an FSM process block, along with their interconnections. This implementation is a quasi-unaltered version of the process block presented in [5], so please refer to that paper for further details concerning most of the modules used here.

However, a brief description of the various modules and their role in the FSM process block follows:

- The input junction (IJ) block is only a connector that routes the wires corresponding to the events of the sort of the process form the events input port to the minterm block.
- The minterm (MT) block computes all the minterms that guard transitions. This is done with wide AND gates, implemented using the Virtex carry-chain. Those minterms are fed to the guard block.
- The guard (G) block then computes the guards for state transitions, by adding minterms using ripple OR gates.
- The requester (R) block uses the current state to determine which transitions are valid. The filtered guards are then fed to the disjunction block.

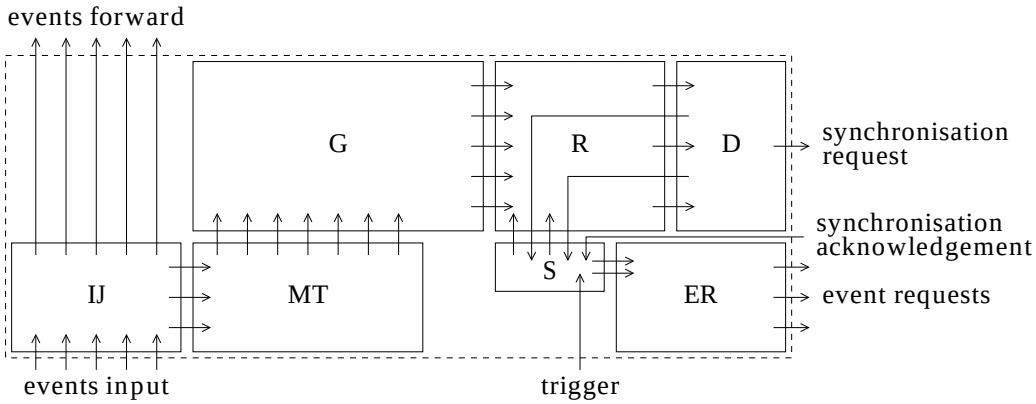

Figure 4: FSM process block layout

- The disjunction (D) block, given the guards, computes to which state each transition could lead, enables the corresponding state flip-flop, and sends an synchronisation request.
- The states (S) block is itself composed of an initial state (IS) module and several non-initial state (NIS) modules, that consist of registers remembering which state is active. On initialisation, the initial state is activated, unless the process is connected to a trigger net, in which case no state is activated until a trigger signal activates the initial state.
- The event requester (ER) block generates requests to assert abstracted events. The design of this block is detailed in the next section.

7.3.3 Event requester block

As mentioned earlier, the event requester block will generate one signal for each event abstracted at a higher level. This block relies on the idea developped in [1], where it is proposed that abstracted events should be generated by processes themselves that request these events whenever they can accept them. Thus, this block is in fact a set of ripple OR gates, one for each abstracted event, that combines the status of each state that can request this particular event.

This block is expensive as far as routing resources are concerned, and, assuming that only a few events are abstracted and only a small number of states can request these events, we have chosen to pack this block to avoid waste of area, instead of spreading it in order to respect a suitable routing resource usage. Of course, this drastic choice is not suitable in the long-term, and trying to come up with a better alternative will be a future objective.

7.3.4 Area issue

As the layout for this block has not changed from its original version, it is almost the same. Therefore, given a FSM process P such as:

- P has n_{state} states;
- its sort is $\mathcal{S}$;
- it has n_{guard} distinct guards ($n_{\text{guard}} \leq 2^{|\mathcal{S}|}$);

- it is able to perform n_{trans} transitions ($n_{\text{trans}} \leq n_{\text{state}} \cdot n_{\text{guard}}$);
- n_{abs} events are abstracted ($n_{\text{abs}} \leq |\mathcal{S}|$);
- in average, each abstracted event can be requested by n_{req} states ($n_{\text{req}} \leq n_{\text{state}}$);

then the area needed to implement P would be a rectangular area of dimension (h, w) where:

$$\begin{cases} h &= \max(\lceil |\mathcal{S}|/8 \rceil, \lceil n_{\text{abs}} \cdot \lceil n_{\text{req}}/3 \rceil / 12 \rceil) + \lceil n_{\text{trans}}/6 \rceil \\ w &= \lceil n_{\text{guard}}/2 \rceil + \lceil (n_{\text{state}} + 1)/4 \rceil + 4 \end{cases}$$

7.4 "Composition" process block

7.4.1 Identifying composition processes

The identification of this type of process is direct, as the compiler only has to ensure that the corresponding node is a composition node.

Each of the children of this node are then in turn mapped to the FPGA, and placed and wired according to the layout described in the next section.

7.4.2 FPGA mapping layout

Figure 5 illustrates the layout of a composed process block. Here again, the design is almost the same as in [5].

Here is a short description of this design:

- Each child P_i of the composition node is recursively mapped to a process block. Those blocks are stacked vertically and left-aligned, in order to have each events forward port abutting the events input port of the next process. The synchronisation acknowledgement and trigger signals are distributed to all the processes, and their synchronisation and event requests wires are respectively driven to a synchronisation logic block and an event generator block.
- The synchronisation logic (SL) block is implemented as a wide AND gate, that will send a request if and only if all the processes request synchronisation.
- The event generator (EG) block has the same role as the event requester block of an FSM process block has. However, as discussed in the following section, it is closer to a synchronisation logic block than to an event requester block.

7.4.3 Event generator block

The event generator block has to produce a signal for each abstracted event that is to be asserted. Thus, it has to combine the requests of the child processes in order to trigger a request signal for an event if and only if all the children request this event.

Therefore, the behaviour of this block is similar to the behaviour of a synchronisation logic block for each abstracted event. Hence, it is implemented in the same way, using wide AND gates.

But here again, this block is expensive in terms of routing resources, and this compacted layout relies on the assumption that the number of abstracted events is not too large. This issue will be investigated in the future.

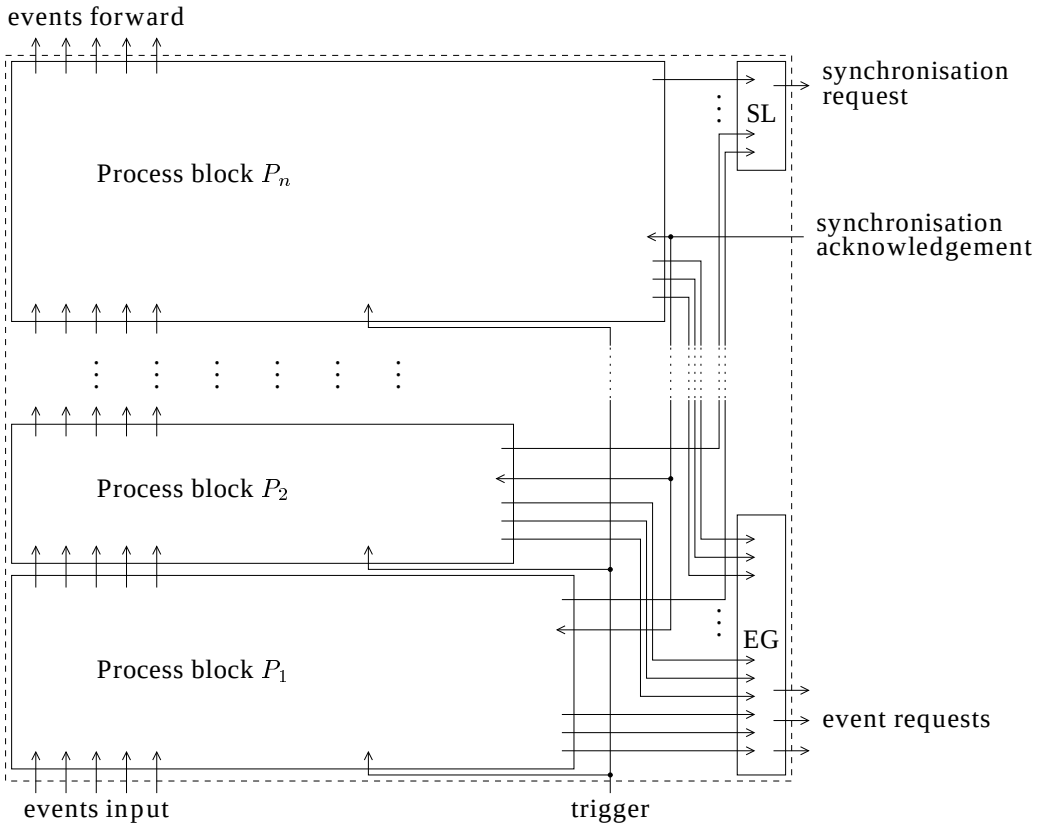

Figure 5: Composition process block layout

7.4.4 Area issue

Given a composed process P such as:

- $P \leftarrow P_1 * \cdots * P_n$;
- its sort is $\mathcal{S}$;
- each P_i occupies the area (h_i, w_i);
- n_{abs} events are abstracted ($n_{\text{abs}} \leq |\mathcal{S}|$);

then the area needed to implement P would be a rectangular area of dimension (h, w) where:

$$\begin{cases} h & = & \max\left(\sum_{i=1}^{n} h_i, \lceil n/8 \rceil \cdot (\lceil n_{\text{abs}}/2 \rceil + 1)\right) \\ w & = & \max\limits_{1 \leq i \leq n} w_i + 1 \end{cases}$$

7.5 “Abstraction” process block

7.5.1 Identifying abstraction processes

Again, the identification of this type of process is direct, as an abstraction process simply corresponds to an abstraction node in the process graph.

The unique child of this node is then recursively treated as another process, and placed and connected according to the layout described in the following section.

7.5.2 FPGA mapping layout

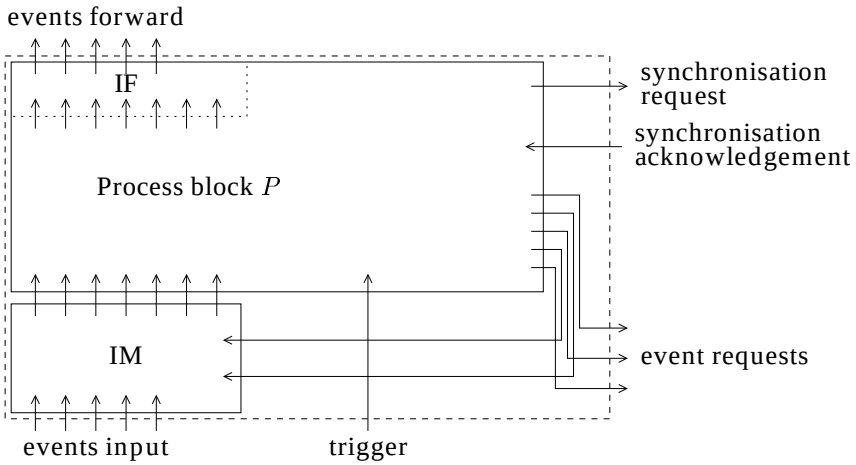

Figure 6: Abstraction process block layout

For this kind of process block, the design is quite straightforward:

- The input merger (IM) block adds the request signals of the events that are abstracted at the current level into one larger internal events bus.
- The child process P is then recursively implemented, and connected to the internal events bus (the output of the input merger block). Its synchronisation wires are forwarded through the corresponding ports of the abstraction process block. Its event requests bus is split into those events that are abstracted at the current level and whose wires are routed to the input merger block, and those that are forwarded to a higher level before being abstracted.
- The input filter (IF) restores the original events input bus by filtering out events that are abstracted at the current level.

7.5.3 Input merger block

In this implementation, any abstracted event that is requested will be asserted. Thus, the input merger only merges the events input bus and the event requests for the events that are abstracted at the current level. This block does not use any CLB resource, but still occupies area to provide enough routing resources for implementation.

Although this approach is quite straightforward, and directly adapted from the idea developed in [1], it nevertheless often creates deadlocks. Indeed, if one process can accept two different

abstracted events, both will be asserted, but if the process cannot accept these two events simultaneously, it will be deadlocked. An example of a process that would be deadlocked by our mapping strategy is given by S in the following example:

$$\begin{aligned} S &\leftarrow P - \{a, b\} \\ P &\leftarrow a\ P_0 + b\ P_1 \end{aligned}$$

This problem should also be investigated in the future, even if this implementation only applies when a system is being tested, since when it is deployed, the abstracted events would presumably be provided by other systems in the environment.

7.5.4 Input filter block

The input filter block is also a simple block, as it only acts as a terminator for the wires of the events that are abstracted at the current level, while forwarding the other wires to the next process.

One should notice that this block overlaps with the child process block because, as it is a connector, it does not use any CLB resource, and can therefore exploit the routing resources that would have been used by the event bus.

7.5.5 Area issue

Given an abstraction process P such as:

- $P \leftarrow P' - \mathcal{A}$;
- P' occupies the area (h', w');

then the area needed to implement P would be a rectangular area of dimension (h, w) where:

$$\begin{cases} h &= h' + \lceil |\mathcal{A}|/8 \rceil \\ w &= w' \end{cases}$$

7.6 "Hybrid" process block

7.6.1 Identifying hybrid processes

A hybrid process corresponds to a choice node in the process graph, and is in fact an FSM process, except that its nodes are not only choice nodes, but can be composition or abstraction nodes as well. This hybrid process notion is in fact the keystone of hierarchy, as it allows processes to behave as a finite state machine, and then to "evolve" into a composed process, as in the following example:

$$P \leftarrow a\ P' + b\ (Q * R)$$

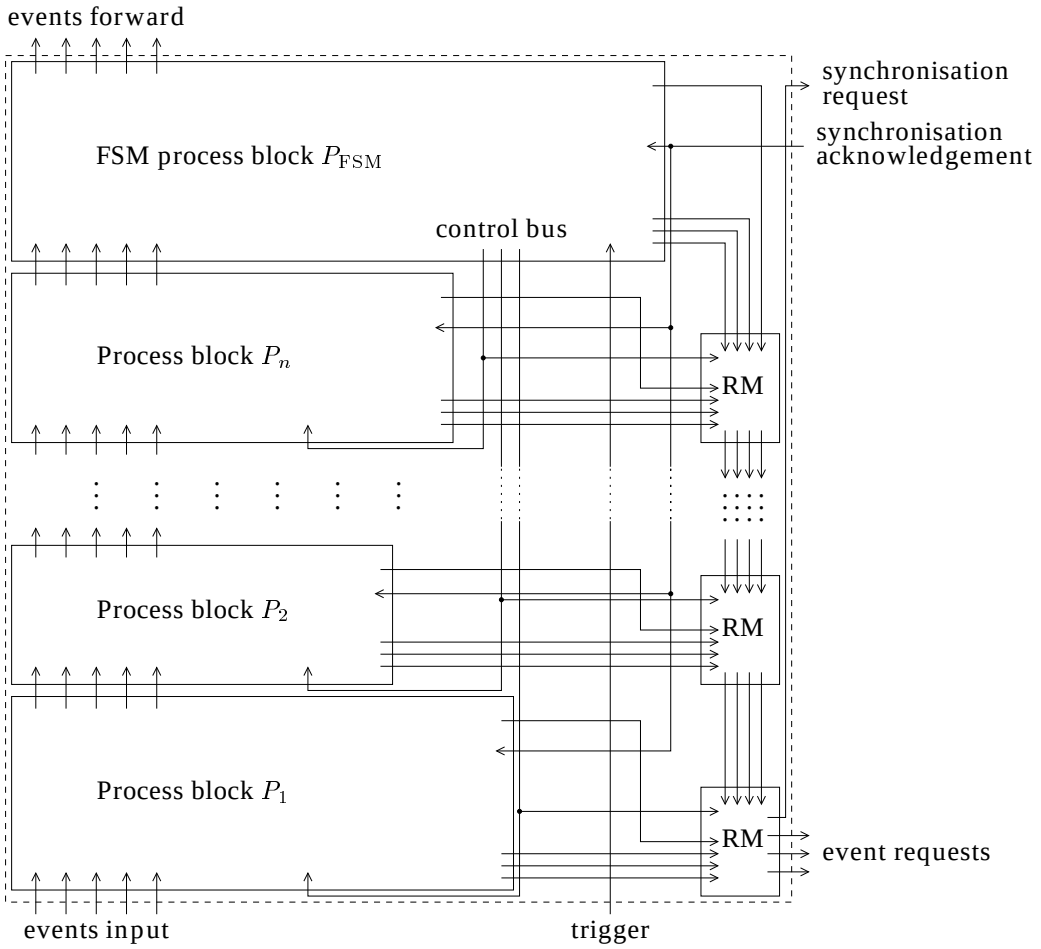

Figure 7: Hybrid process block layout

7.6.2 FPGA mapping layout

A hybrid process being mainly an FSM process, we first have to identify this process, called P_{FSM}, and then to compute the list of the "non-choice" states in P_{FSM}, say $P_1, \ldots, P_n$. These processes are then implementing according to the following layout:

- All the processes $P_1, \ldots, P_n$ are recursively implemented, and put in a "dormant state": in fact, no state is active, and thus no transition can occur. The only way to activate them is to send them a trigger signal.
- The process P_{FSM} is then implemented as a normal FSM process. The only difference here is the control bus. Each wire of this bus is connected to a state register that corresponds to a non-choice node. In this way, if P_{FSM} enters a non-choice state, it will send a pulse through this wire to "trigger" the corresponding process P_i, and will itself enter a "dormant state".
- The request multiplexer (RM) blocks only select the requests of the active process to drive them to the corresponding ports of the hybrid process.

7.6.3 Request multiplexer block

A request multiplexer block is a switch that initially drives both synchronisation and event requests from its north input (in other words, it only forwards its input) to the output, but if a trigger pulse is received, it will change its switch to drive the requests from its west input (selecting the requests of its corresponding process P_i).

In this way, when initialised, the chain of request multiplexer blocks forwards the requests of P_{FSM}, and as soon as a process P_i is activated by a trigger pulse, the corresponding request multiplexer block routes P_i's requests, effectively selecting these signals.

7.6.4 Area issue

Given a hybrid process P such as:

- the "non-choice" nodes of P are $P_1, \ldots, P_n$
- its sort is $\mathcal{S}$;
- P_{FSM} occupies the area $(h_{\mathrm{FSM}}, w_{\mathrm{FSM}})$;
- each P_i occupies the area (h_i, w_i);
- n_{abs} events are abstracted ($n_{\mathrm{abs}} \leq |\mathcal{S}|$);

then the area needed to implement P would be a rectangular area of dimension (h, w) where:

$$\begin{cases} h &= h_{\mathrm{FSM}} + \displaystyle\sum_{i=1}^{n} \max(h_i, \lceil (n_{\mathrm{abs}} + 2)/4 \rceil) \\ w &= \max\left(w_{\mathrm{FSM}}, \displaystyle\max_{1 \leq i \leq n} w_i\right) + 1 \end{cases}$$

8 Example

To give an illustration of these notions in a more concrete fashion, let us consider the following definition of process S, whose graph representation is given in Figure 8, and its mapping onto FPGA in Figure 9:

$$S \leftarrow P_0 - b$$

$$\begin{cases} P_0 &\leftarrow a\, P_0 + b\, P_1 + (a\, b)\, P_2 \\ P_1 &\leftarrow a\, P_2 \\ P_2 &\leftarrow (Q_0 * R_0) - c \end{cases}$$

$$\begin{cases} Q_0 &\leftarrow b\, Q_0 + a\, Q_1 \\ Q_1 &\leftarrow c\, Q_0 \end{cases}$$

$$\begin{cases} R_0 &\leftarrow b\, R_1 \\ R_1 &\leftarrow c\, R_0 \end{cases}$$

As one can notice, this process covers the four cases prensented earlier: indeed, nodes Q_0 and Q_1 form a FSM process (called Q), and so do nodes R_0 and R_1 with FSM process R. These two processes are then composed into $Q*R$, from which event c is abstracted to form process P_2. Thus, process P appears as a hybrid process, as it behaves as a finite state machine for states P_0, P_1 and P_2, except that P_2 is not a choice node but this abstraction node. Then, event b is abstracted from P to obtain the actual implemented process, S.

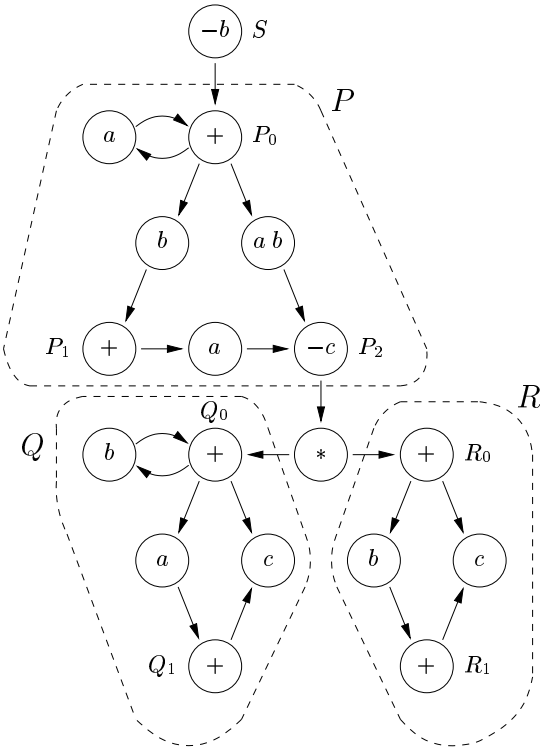

Figure 8: Normalised graph representation of the example process S

Given this hierarchical structure, the final layout is quite simple in that it reflects the hierarchy, as illustrated in Figure 9.

Thus, the main block for process S is an abstraction block, that merges the actual input event a with the abstracted event b, and then filtering b out just before the event forwarding port.

Descending in the block hierarchy, the next block is a hybrid block, corresponding to process P. In this hybrid process, the "controlling" finite state machine, say P_{FSM} is formed by states P_0, P_1 and P_2, with P_2 triggering the "controlled" process $(Q * R) - c$. One can notice the request multiplexer block that selects between either synchronisation and event requests of P_{FSM} and the ones of $(Q * R) - c$, according to which one of these two processes is active.

The implementation of P_{FSM} is also straightforward, as it is a regular FSM process block, except for the control wire that allows this process to trigger the activation of $(Q * R) - c$ when it enters state P_2. One should also notice the event requester block will drive any request for the abstracted event b.

Concerning $(Q * R) - c$, as it is also an abstraction, first comes the corresponding abstraction block, merging requests for the abstracted event c to the event bus, and forwarding requests for event b, as it is abstracted at a higher level.

Then follows a composition block corresponding to $Q * R$, with a synchronisation logic block combining the requests from Q and R, and an event generator block performing the same function for event requests.

Finally, at the lowest level, two FSM process blocks implement Q and R. Here, one can notice the trigger wire that has been driven all the way from P_{FSM} through the different levels of the hierarchy, to allow an activation of Q and R controlled by P_{FSM}.

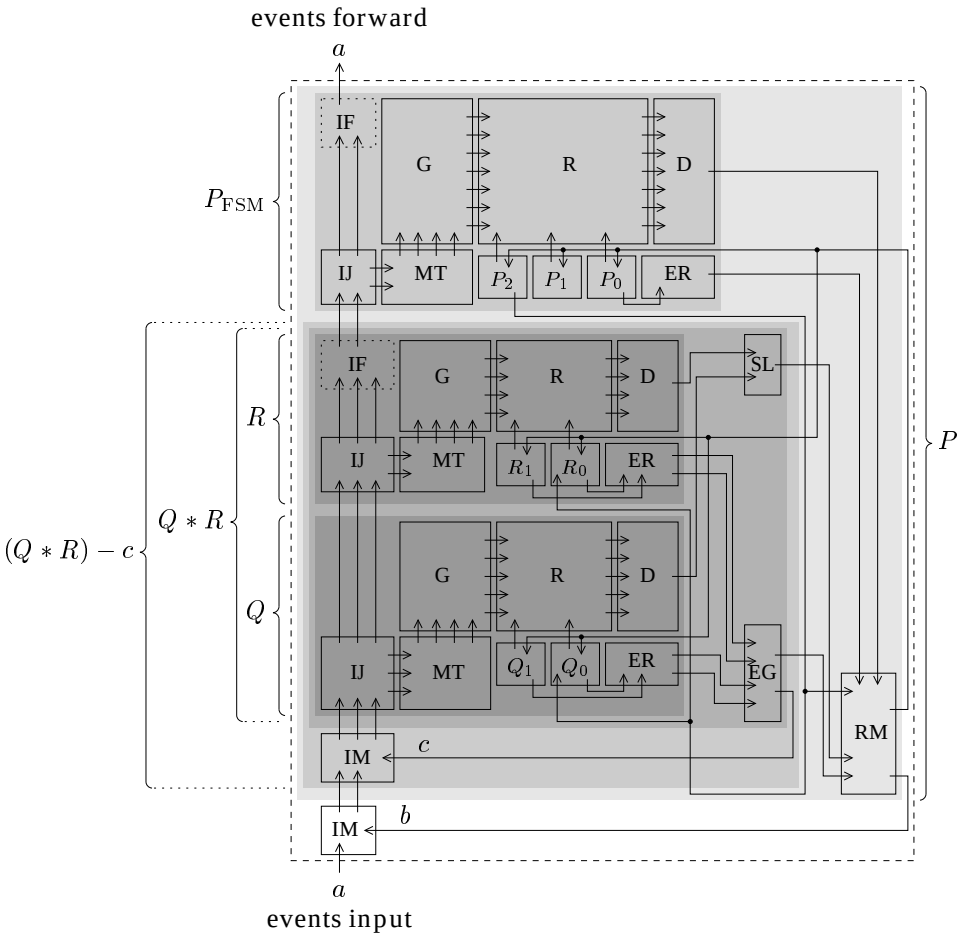

Figure 9: Layout of the example process S

9 Conclusion and future work

In this paper, we have presented a new version of our static Circal compiler targeted to Xilinx Virtex FPGAs. This compiler is now able to understand XCircal programs, to extract process specifications from them, and to map these specifications to reconfigurable logic. Indeed, except for non-determinism, all the features of the Circal process algebra are now supported and effectively mapped into equivalent hardware.

This work is still at an early stage, and a large number of issues remain to be investigated. Among them:

- The XCircal parser / interpreter is not complete, and in future should be replaced by the original XTC engine, interfaced with Java using JNI.
- The routing resource usage should be studied for new modules such as the event requester blocks, in order to come up with a better layout of this core, without sacrificing area for the most common cases.
- The generation of abstracted events should be modified to avoid the problem of deadlocked processes.
- Last but not least, the work achieved on the compiler should be ported to the interpreter, to be able to implement larger designs without having to worry about running out of area. This is crucial to being able to effectively create and destroy processes without having the associated area statically allocated.

References

[1] Oliver Diessel and George Milne. A note on the hardware implementation of the Circal abstraction operator. Draft, 1999.

[2] Oliver Diessel and George Milne. A hardware compiler realizing concurrent processes in reconfigurable logic. *IEE Proceedings — Computers and Digital Techniques*, 148(4):152 – 162, September 2001.

[3] George A. McCaskill. The XTC language reference manual. Technical Report HDV-14-91, Department of Computer Science, University of Strathclyde, 1991.

[4] George Milne. *Formal Specification and Verification of Digital Systems*. McGraw–Hill, London, UK, 1994.

[5] Keith So. Compiling abstract behaviours to Field Programmable Gate Arrays. Undergraduate thesis, School of Computer Science and Engineering, University of New South Wales, Oct. 2001. Available at `http://www.cse.unsw.edu.au/~odiessel/papers/kso-ugthesis.ps.gz`.