A Constructive Proof of the Turing Completeness of Circal

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-0214
September, 5 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 gives a proof of the Turing completeness of the Circal process algebra by exhibiting a universal program capable of mapping any Turing machine description into Circal specifications that effectively simulate the behaviour of the given machine.

1 Introduction

1.1 The Circal process algebra

1.1.1 Brief overview

The Circal process algebra was developed by G. Milne [3], being mainly targeted to control-oriented applications and system verification. Indeed, Circal offers a simple and efficient way of describing systems and their inherent notions of synchronisation and concurrency, along with some rules that allow the system to be exhaustively checked against given specfications.

A brief summary of the Circal operators is as follows:

- 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 can behave either as P or as Q, with the choice depending only on the process itself, and being 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 and that encapsulates the events of $\mathcal{S}$, which are then hidden externally.
- 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 the behaviour of Q, thus allowing recursive definitions.

1.1.2 Mapping Circal to reconfigurable logic

This process algebra has recently proven to be quite easily mapped into field-programmable gate arrays (FPGAs), as presented in [2, 4, 1]. This opportunity offers many applications, such as hardware accelerated model-checking, and the ability to embed complex controllers directly onto FPGAs.

However, one question remains unanswered regarding the usefulness of Circal: is the process algebra powerful enough to describe any system? This question was posed as the problem of proving the Turing completeness of the Circal process algebra. Although it has commonly been assumed, so far no proof has been demonstrated.

1.2 Turing machines

As this paper is targeted to a wide audience, we think that recalling a few notions about Turing machines would serve to aid the understanding of the rest of this paper. Readers who are familiar with Turing machines may skip this section.

The formal definition of a Turing machine if a tuple $\mathcal{M} = (Q, k, \Sigma, B, \delta, Q_{\text{accept}})$ where:

- Q is a set of states;
- $k \in \mathbb{N}$ is the number of tapes;
- Σ is an alphabet (a set of symbols);
- $B \in \Sigma$ is a special symbol, meaning “blank”;
- $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{-1, +1\}^k$ is the transition function;
- $Q_{\text{accept}} \subseteq Q$ is the set of accepting states.

The configuration of a Turing machine is given by the tuple $\mathcal{C} = (q, p, c)$ where:

- $q \in Q$ is the current state;
- $p = (p_1, \ldots, p_k) \in \mathbb{Z}^k$ is the position of the heads;
- $c : \{1, \ldots, k\} \times \mathbb{Z} \to \Sigma$ is the contents of the tapes.

And finally, in one step, a configuration $\mathcal{C} = (q, p, c)$ can lead to a configuration $\mathcal{C}' = (q', p', c')$ if and only if:

- $\delta(q, (c(1, p_1), \ldots, c(k, p_k))) = (q', (a_1, \ldots, a_k), (m_1, \ldots, m_k))$ where $\forall i \in \{1, \ldots, k\}$, $a_i \in \Sigma$ and $m_i \in \{-1, +1\}$;
- $\forall i \in \{1, \ldots, k\}$, $c'(i, p_i) = a_i$;
- $\forall i \in \{1, \ldots, k\}$, $p'_i = p_i + m_i$.

In a more informal way, one can just say that a k-tape Turing machine is a finite state machine coupled with k infinite tapes, along with a head on each of these tapes. At each step, the Turing machine polls the data written where the heads are on each tape, then writes new data, and moves its heads either left (-1) or right (+1) by one cell.

A Turing machine is said to effectively compute a function f when, given an initial configuration $(q_0, (0, \ldots, 0), \overline{n})$, where $\overline{n}$ is the encoded actual parameter n, the machine evolves from configuration to configuration until entering an accepting state $q \in Q_{\text{accept}}$, its configuration then being $(q, p, \overline{m})$, where $m = f(n)$.

One interesting result that has been proved is that any k-tape Turing machine is equivalent to a single semi-finite tape Turing machine (and therefore this type of Turing machine has the same “computing power” as general Turing machines).

1.3 Turing completeness

As Turing machines were presented as a “suitable programming model”, hence introducing a notion of computability, they became a reference for computational power, being considered to be the most powerful machines. Thus, the notion of Turing completeness qualifies the computing power of a given system: indeed, a system is said to be Turing complete if it can perform all the computations a Turing machine can.

A simple way to prove the Turing completeness of a programming language is to show that one can describe a universal machine for this language, that is a machine that, given the description and the initial configuration of a Turing machine, will output a program in that language, and that, according to the semantic rules of the language, will simulate the original Turing machine, effectively computing the same function.

Thus, in order to prove the Turing completeness of the Circal process algebra, this paper focuses on the realisation of a Turing machine in Circal, described in the next section.

2 Description of the Turing machine process

2.1 General overview of the machine

The implemented machine is a quite straightforward transposition of a single semi-finite tape Turing machine. Indeed, we can identify the two main parts:

- The finite state machine: this part is an automaton, that remembers in which state the machine is, and successively polls the head for the data on the tape, writes new data on the tape and moves the head, then changes its own state.
- The tape: which is trickier to represent, since the tape is in theory infinite. But hopefully, Turing machines don’t allow random access to any cell of the tape: the head can only move from a cell to its left or right neighbour. Therefore, it allows us to use a finite tape, that can be dynamically extended when the head needs to visit new cells. The head is directly embedded in the tape, the cell that is being read by the head being in a particular state.

2.2 Communications between the automaton and the tape

The communications between these two main parts are achieved by some dedicated events, whose description follow:

- e_a, $a \in \Sigma$: one of these events is sent by the tape when polled for its data, and another one is sent by the automaton to write data onto the tape;
- e_{left} and e_{right}: one of these events is sent by the automaton to move the head one cell left or right;
- e_{ready}: this event is sent by the tape when the head has successfully moved to the next cell.

2.3 The automaton process

The automaton process A is composed of a large monolithic process, mainly translated form the orignal automaton, each transition being replaced by a sequence of sub-states, in order to guarantee the sequentiality of the polling and writing of the tape, and the moving of the head.

Thus, we can have the following definition, where $q \in Q$, $a \in \Sigma$ and $\delta(q,a) = (q', a', m)$:

- $A_{\text{idle},q} \leftarrow e_{\text{ready}}\ A_{\text{reading},q}$: the automaton waits for the head to be correctly placed;
- $A_{\text{reading},q} \leftarrow \sum_{a \in \Sigma} e_a\ A_{\text{writing},q,a}$: the tape sends the symbol read by the head;
- $A_{\text{writing},q,a} \leftarrow e_{a'}\ A_{\text{moving},q,a}$: the automaton sends the symbol to be written by the head;
- $A_{\text{moving},q,a} \leftarrow e_m\ A_{\text{idle},q'}$ where $e_m = e_{\text{left}}$ if $m = -1$ and $e_m = e_{\text{right}}$ if $m = +1$: the automaton moves the head.

2.4 The tape process

The tape process is much more complex, as it involves the head passing from one cell to another, along with dynamic creation of cells. Thus, we have to deal with three main tape cells:

- the first cell, that does not have any left neighbour;
- the regular cells, that are in the middle of the tape, that have a left and a right neighbour;
- the end-of-tape cell, that is not actually a cell as it cannot contain data, but that can give birth to a regular cell on demand.

As one may notice, the space locality of the head along with the locality of its movements require local communications between the cells, so that a cell can actually “give” the head to one of its neighbours. As Circal only offers a finite number of events and global synchronisation between processes, we overcome that problem using abstraction. More details concerning the events we used are given in the next section, and section 2.4.2 describes the actual processes involved in the tape.

2.4.1 Internal communications between cells

Figure 1 illustrates the organisation and linking of processes in the tape.

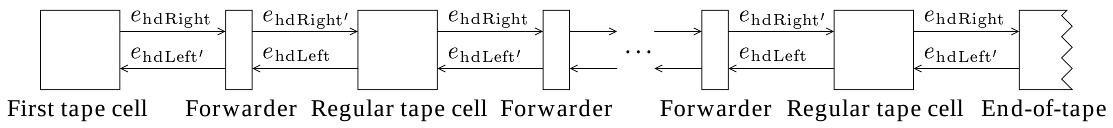

Figure 1: Schematic representation of communications in the tape

Except for the “forwarders”, whose role is detailed in the next section, the organisation of the internal communications of the tape is quite direct:

- e_{hdLeft} (and $e_{\text{hdLeft}'}$): these events are asserted between two adjacent cells when the head moves from the rightmost cell to the leftmost one;
- e_{hdRight} (and $e_{\text{hdRight}'}$): these events have the same role as previous ones, but for a left to right movement of the head.

2.4.2 The different tape processes

As the first tape cell is almost like a regular tape cell, except that it ignores all the events that move the head leftward, we only present here a definition of a regular cell C, where $a, a' \in \Sigma$:

- $C_{\text{idle},a} \leftarrow \sum_{e \notin \{e_{\text{hdLeft}'}, e_{\text{hdRight}'}\}} e\ C_{\text{idle},a} + \sum_{e \in \{e_{\text{hdLeft}'}, e_{\text{hdRight}'}\}} e\ C_{\text{head},a}$: unless it receives an event stating the arrival of the head, the cell stays in an idling state;
- $C_{\text{head},a} \leftarrow e_{\text{ready}}\ C_{\text{reading},a}$: the cell with the head signals the automaton that it is ready to start the "poll-write-move" sequence;
- $C_{\text{reading},a} \leftarrow e_a\ C_{\text{writing}}$: the cell then sends its symbol;
- $C_{\text{writing}} \leftarrow e_{a'}\ C_{\text{moving},a'}$: then it writes the new data to the tape;
- $C_{\text{moving},a'} \leftarrow e_{\text{left}}\ C_{\text{left},a'} + e_{\text{right}}\ C_{\text{right},a'}$: the automaton determines in which direction the head must move;
- $C_{\text{left},a'} \leftarrow e_{\text{hdLeft}}\ C_{\text{idle},a'}$: moving the head leftward;
- $C_{\text{right},a'} \leftarrow e_{\text{hdRight}}\ C_{\text{idle},a'}$: moving the head rightward.

As one can see, this process is again nothing but a state remembering the current symbol of the cell, and a sequence of actions to perform when the head visits this cell, matching those of the automaton.

The following is the description of the forwarder process F:

- $F_{\text{idle}} \leftarrow e_{\text{hdLeft}}\ F_{\text{left}} + e_{\text{hdRight}}\ F_{\text{right}} + e_{\text{ready}}\ F_{\text{idle}}$: when all forwarders are idle, they allow the automaton to start another "poll-write-move" sequence;
- $F_{\text{left}} \leftarrow e_{\text{hdLeft}'}\ F_{\text{idle}}$: forwarding leftward move event;
- $F_{\text{right}} \leftarrow e_{\text{hdRight}'}\ F_{\text{idle}}$: forwarding rightward move event;

As suggested by its name, a forwarder process only forwards local events. If its usefulness can be doubtful at first glance, it is important to notice that without forwarders, each tape cell process would have similarly named events for both left and right ports, and this problem cannot be solved easily in Circal.

Finally, the description of the end-of-tape process E is as follows:

- $E_{\text{idle}} \leftarrow \sum_{e \neq e_{\text{hdRight}}} e\ E_{\text{idle}} + e_{\text{hdRight}}\ E_{\text{forking}}$: unless it receives an event stating the arrival of the head, the process stays in an idling state;
- $E_{\text{forking}} \leftarrow (F_{\text{idle}} * ((C_{\text{head},B} * E_{\text{idle}}) - \{e_{\text{hdLeft}'}, e_{\text{hdRight}}\})) - \{e_{\text{hdLeft}}, e_{\text{hdRight}'}\}$: when forking, the end-of-tape generates a forwarder and a fresh tape cell (initialised with the blank symbol B).

In fact, here lies the basic structure on which is based the whole tape: indeed, two successive abstractions allow us to reuse the events to ensure local communications between cells.

3 Example

This section describes the implementation of a simple Turing machine. The Turing machine we have chosen for this example is a binary counter, that, with the alphabet $\Sigma = \{B, 0, 1\}$, on the input $< B\ \overline{n} >$ (where $\overline{n}$ is the binary representation of $n \in \mathbb{N}$, the leftmost bit being the least significant bit), successively computes $< B\ \overline{(n+1)} >, < B\ \overline{(n+2)} >, \ldots$

This Turing machine is interesting as an example, because it only has a few number of states and transitions, simplifying the automaton process, and it combines all the "tricky" features of our Turing machine implementation: indeed, it moves the head left and rightward, and sometimes has to extend the tape.

Thus, this machine only has two states (q_0 and q_1) and six transitions, as detailed in the following table:

$$\begin{array}{llll}
\delta : & (q_0, B) & \mapsto & (q_1, B, +1) \\
 & (q_0, 0) & \mapsto & (q_0, 0, -1) \\
 & (q_0, 1) & \mapsto & (q_0, 1, -1) \\
 & & & \\
 & (q_1, B) & \mapsto & (q_0, 1, -1) \\
 & (q_1, 0) & \mapsto & (q_0, 1, -1) \\
 & (q_1, 1) & \mapsto & (q_1, 0, +1)
\end{array}$$

Figure 2 illustrates the graph representation of the corresponding automaton. As one can see, its structure mainly resembles the structure of the original Turing machine automaton, each transition being in fact represented here by a sequence of actions (events) of the form "poll-write-move" (e_B - e_1 - e_{left} for example).

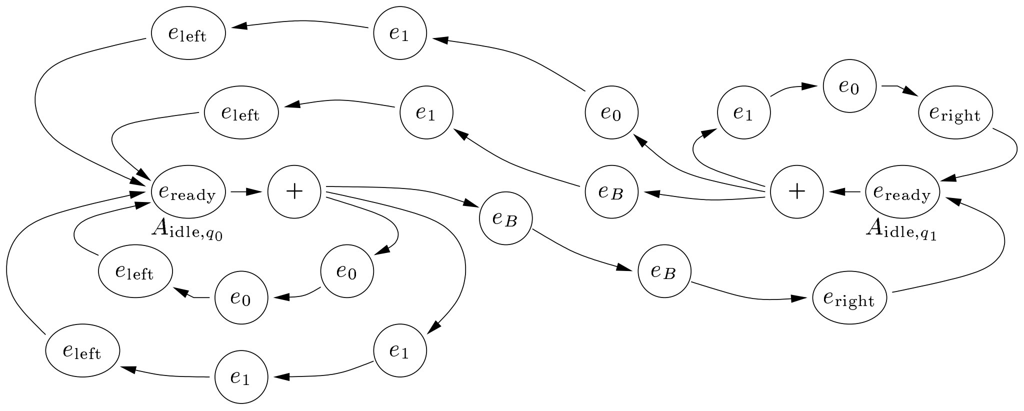

Figure 2: Graph representation of the automaton process A

The table in figure 3 is a simplified execution trace of the processes, in parallel with the behaviour of the original Turing machine. For the sake of readability, only the automaton (A) and the tape cells (C) processes are represented, the forwarder (F) and end-of-tape (E) processes being "hidden". Concerning the reading of the tape (rightmost column), the position of the head is represented by underlining the corresponding cell.

Process states	State	Action	Tape			
$A_{\text{idle},q_0} * C_{\text{head},B} * C_{\text{idle},0} * C_{\text{idle},1}$	q_0	idle	$\underline{B}$	0	1	
$A_{\text{reading},q_0} * C_{\text{reading},B} * C_{\text{idle},0} * C_{\text{idle},1}$	q_0	reading	$\underline{B}$	0	1	
$A_{\text{writing},q_0,B} * C_{\text{writing}} * C_{\text{idle},0} * C_{\text{idle},1}$	q_0	writing	$\_$	0	1	
$A_{\text{moving},q_0,B} * C_{\text{moving},B} * C_{\text{idle},0} * C_{\text{idle},1}$	q_0	moving	$\underline{B}$	0	1	
$A_{\text{idle},q_1} * C_{\text{right},B} * C_{\text{idle},0} * C_{\text{idle},1}$	q_1	moving	$\underline{B}$	0	1	
$A_{\text{idle},q_1} * C_{\text{idle},B} * C_{\text{head},0} * C_{\text{idle},1}$	q_1	idle	B	$\underline{0}$	1	
$A_{\text{reading},q_1} * C_{\text{idle},B} * C_{\text{reading},0} * C_{\text{idle},1}$	q_1	reading	B	$\underline{0}$	1	
$A_{\text{writing},q_1,0} * C_{\text{idle},B} * C_{\text{writing}} * C_{\text{idle},1}$	q_1	writing	B	$\_$	1	
$A_{\text{moving},q_1,0} * C_{\text{idle},B} * C_{\text{moving},1} * C_{\text{idle},1}$	q_1	moving	B	$\underline{1}$	1	
$A_{\text{idle},q_0} * C_{\text{idle},B} * C_{\text{left},1} * C_{\text{idle},1}$	q_0	moving	B	$\underline{1}$	1	
$A_{\text{idle},q_0} * C_{\text{head},B} * C_{\text{idle},1} * C_{\text{idle},1}$	q_0	idle	$\underline{B}$	1	1	
$A_{\text{idle},q_1} * C_{\text{idle},B} * C_{\text{head},1} * C_{\text{idle},1}$	q_1	idle	B	$\underline{1}$	1	
$A_{\text{idle},q_1} * C_{\text{idle},B} * C_{\text{idle},0} * C_{\text{head},1}$	q_1	idle	B	0	$\underline{1}$	
$A_{\text{idle},q_1} * C_{\text{idle},B} * C_{\text{idle},0} * C_{\text{idle},0} * C_{\text{head},B}$	q_1	idle	B	0	0	$\underline{B}$
$A_{\text{idle},q_0} * C_{\text{idle},B} * C_{\text{idle},0} * C_{\text{head},0} * C_{\text{idle},1}$	q_0	idle	B	0	$\underline{0}$	1
$A_{\text{idle},q_0} * C_{\text{idle},B} * C_{\text{head},0} * C_{\text{idle},0} * C_{\text{idle},1}$	q_0	idle	B	$\underline{0}$	0	1
$A_{\text{idle},q_0} * C_{\text{head},B} * C_{\text{idle},0} * C_{\text{idle},0} * C_{\text{idle},1}$	q_0	idle	$\underline{B}$	0	0	1

Figure 3: Simplified execution trace of the Turing machine processes

4 Conclusion

In this paper, we have given an algorithm for the construction of any Turing machine using the Circal process algebra. This is therefore a constructive proof of the Turing completeness of Circal.

If this result was commonly assumed, mainly because of the nearness of Circal to other Turing complete process algebras, such as Milner's CCS or Hoare's CSP, its proof is still a good step forward, as we are now sure of the computational power of this process algebra. Indeed, many applications of Circal, especially using its FPGA embedded version, are on the verge of being developed, and ensuring that these applications will prove to be useful was a necessity.

Another point that needs to be underlined is that this Turing machine implementation in Circal only uses constructs that can be mapped in our hardware version. Thus, even if the interpreter needs to be extended to support these constructs, we now know that this interpreter will be capable of effectively handling any possible computation, and so will the applications developed above.

Future work for Circal will now be oriented towards the following milestones:

- extending the interpreter to allow it to run our Turing machine;
- developing some applications on top of the interpreter, such as hardware accelerated model-checking.

References

[1] Oliver Diessel and Usama Malik. An FPGA interpreter with virtual hardware management. In *Proceedings Reconfigurable Architectures Workshop 2002*, Los Alamitos, CA, April 2002. IEEE

Computer Society Press.

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

[4] 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`.