Implementing Condition/Event Nets in the Circal Process Algebra

Antonio Cerone

Software Verification Research Centre
The University of Queensland, QLD 4072, Australia
Phone +61-7-3365-1651, Fax +61-7-3365-1533
`antonio@svrc.uq.edu.au`
`http://www.itee.uq.edu.au/~antonio/`

Abstract. We define a translation from Condition/Event nets to the Circal process algebra. Such a translation exploits the Circal feature of allowing the simultaneous occurrence of distinct actions. This permits us to give Condition/Event nets a semantics based on true concurrency, in addition to the interleaving-based semantics. In this way the true concurrency aspects of Condition/Event nets are preserved in the process algebra representation and can be analysed using the verification facilities provided by the Circal System. Systems modelled partly using Condition/Event nets partly using the Circal process algebra can also be analysed within the same verification environment.

1 Introduction

Petri nets and process algebras are very popular formalisms used for modelling and verifying concurrent systems. However, they express different modelling styles, have different underlying semantics and different mathematical and graphical representations, and are associated with different analysis techniques.

In this paper, we define a translation from Condition/Events nets (C/E nets) [20,19], a subclass of Petri nets where every place contains at most one token, to the Circal process algebra [13]. Among the many process algebras available we have chosen Circal because of its distinctive feature of having processes guarded by sets of actions, rather than by single actions, as in all other process algebras [2,12,14]. Having events consisting of simultaneously occurring actions allows the representation of true concurrency and causality, which is explicit in Petri net based formalisms such as C/E nets.

The paper is structured as follows. In Section 2 we motivate our approach. Section 3 is a brief introduction to C/E nets. In Section 4 we present the Circal process algebra and its implementation, the XCircal language. Our framework for modelling Petri nets is presented in Section 5. Techniques for analysing properties of C/E nets modelled in the Circal process algebra are presented in Section 6. Finally, Section 7 highlights the novelty of this work and discusses the extension of our approach to Place/Transition nets.

2 Background and Motivation

Condition/Events nets (C/E nets) [20,19], allow the modelling of finite state systems and are, therefore, as expressive as Finite State Machines (FSMs). However, FSMs have a global control, which describes a sequential behaviour, whereas C/E nets have a distributed control, which is defined by distinct tokens being in distinct places at the same time, so allowing an explicit representation of causality.

FSMs have been extended with higher level constructs, which allow the association of complex conditions or statements to states and transitions and the representation of structured data types, and with compositional mechanisms.

Process algebras [2,12,13,14] can be seen as mathematical formalisms for describing systems of concurrent, interacting FSMs. That is a way of giving compositionality to FSMs. The behaviour of the composite system is, however, still defined by a global FSM, where the concurrency among different components is expressed as non-deterministic interleaving. Therefore, the distributed aspects of the system specification do not appear in the behaviour of the composite system.

Petri nets, instead, do not support compositionality. Every attempt to introduce compositionality into Petri nets has resulted in a very restricted form of composition, which is not very useful in system design.

Both FSMs and Petri nets have visual representations, which make them attractive modelling tools also for those system designers who are not familiar with formal specification languages and techniques. Moreover, due to their different characteristics, FSMs and Petri nets are useful in modelling different aspects of the same system.

Let us consider, for example, a process control application. Petri nets are the appropriate formalism for specifying the distributed control of the whole system or the protocol that governs the communication among the system components. On the other hand system modes and system components might be easily specified by FSMs. A similar situation occurs in modelling asynchronous hardware [6,8]. Gate-level or CMOS-level components are easily specified by finite state machines, whereas the asynchronous handshaking control protocol is usually modelled by Signal Transition Graphs (STGs) [10], a subclass of Petri nets. In both examples the Petri net-based part of the specification is usually finite state, and may be modelled by C/E nets.

Among the many existing formal methods automatic tools [11], none is able to manage system specifications consisting of a combination of Petri nets and FSMs. Properties of the system which involve both aspects of the sub-systems modelled by Petri nets and aspects of the sub-systems modelled by FSMs cannot be analysed using such tools. This is the case for fundamental correctness properties, such as the correctness of the circuitry implementation, modelled by an FSM, of an asynchronous handshaking control protocol with respect to its specification, modelled by an STG. This property has been automatically verified by defining a translation from STGs to the Circal process algebra [13], the FSM-based formalism used to model the circuit implementation, and then feeding the specification and implementation models to the Circal System, an automatic verification tool based on the Circal process algebra [6,8].

3 Condition/Event Nets

Petri nets are an abstract formal model of information flow. They were introduced by Carl Adam Petri in 1962 [18]. The basic notion in *Petri net theory* is the notion of a *net*, which is usually described as a triple $\langle S, T, F \rangle$ [19]. Sets S and T are disjoint and are called the set of *places* and the set of *transitions*, respectively. $F \subseteq (S \times T) \cup (T \times S)$ is a binary relation called the *flow relation*.

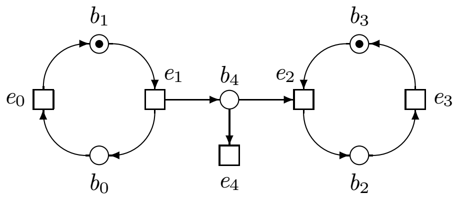
The basic notion of net is given a state, which is called a *marking* and is an assignment of *tokens* to places. Depending on the number of tokens that can be assigned to the same place and on the possible information carried by a token we can have different classes of Petri nets [19,20]. The dynamic behaviour of a Petri net is given by transition *firings*, which consume and produce tokens.

Condition/Event nets (C/E nets) are Petri nets where tokens are just markers for places and every place contains at most one token [20]. In a C/E net, places are called *conditions*, transitions are called *events* and markings, which are defined as the sets of places that are marked by one token, are called *cases*.

A C/E net is defined as a quadruple $N = \langle B, E, F, C \rangle$ such that $\langle B, E, F \rangle$ is a net and $C \subseteq B$ is a case.

A condition $b \in B$ is a *precondition* of an event $e \in E$ if and only if $(b, e) \in F$. A condition $b \in B$ is a *postcondition* of an event $e \in E$ if and only if $(e, b) \in F$.

An example of C/E net $N_1 = \langle B_1, E_1, F_1, C_1 \rangle$ is given in Figure 1(a). In the



(a)

	b_0	b_1	b_2	b_3	b_4	
e_0	c[0]	p[1]				e[0]
e_1	p[0]	c[1]			p[4]	e[1]
e_2			p[2]	c[3]	c[4]	e[2]
e_3			c[2]	p[3]		e[3]
e_4					c[4]	e[4]

(b)

Fig. 1. (a): A C/E net N_1 defining a producer-consumer system; (b): Correspondence between conditions and events of N_1 and Circal actions

pictorial representation, conditions are represented as circles, events are represented as boxes, with arrows from circles to boxes and from boxes to circles defining the flow relation. Tokens are represented by solid circles, each inside the place to which it is assigned.

C/E net N_1 defines an *unreliable producer-consumer system*. The producer consists of conditions b_0 and b_1 and events e_0 and e_1. When b_0 is marked by a token the producer is ready to produce, and event e_0 models the production of a message; when b_1 is marked the producer is ready to send, and event e_1 models the sending of a message. The consumer consists of conditions b_2 and b_3 and events e_2 and e_3. When b_3 is marked the consumer is ready to receive, and event e_2 models the receiving of a message; when b_2 is marked the consumer is ready to consume, and event e_3 models the consumption of a message. Condition b_4

models a 1-cell buffer between the producer and the receiver. Event e_4 models the loss of the message from the buffer.

4 The Circal Process Algebra

In this section we give a brief description of the the Circal process algebra and the Circal System [13,15].

Each Circal *process* has associated with it a *sort*, which specifies the set of actions through which it may interact with other processes. Every sort will be a non-empty subset of $\mathcal{A}$, the collection of all available actions.

The *behaviour* of a process is defined in terms of the `/\` constant and the operators of *guarding*, *external choice*, and *internal choice* (*behavioural operators*).

The `/\` constant represents a process which can participate in no communication. This is a process that has terminated or deadlocked.

In guarded process Γ `P` the `P` process may be guarded by a set Γ of simultaneously occurring actions. This is a key feature of Circal which greatly enriches the modelling potential of the algebra in contrast to process algebras such as CSP [12], CCS [14] and LOTOS [2], which only permit a single action to occur at one computation instant.

A name can be given to a Circal process with the *definition* operator (`<-`). Recursive process definitions, such as `P <-` Γ `P`, are permitted.

The `+` operator defines an *external choice*, which is decided by the environment where the process is executed, whereas the `&` operator defines an *internal choice*, which is decided autonomously by the process itself without any influence from its environment. Internal choices appear to an external observer as *non-determinism.*

The *structure* of a process is defined in terms of the operators of *composition* and *hiding* (*structural operators*).

Given processes `P` and `Q`, the term `P * Q` represents the process which can perform the actions of the subterms `P` and `Q` together (*composition*). Any synchronisation which can be made between two terms, due to some action being common to the sorts of both subterms, must be made, otherwise the actions of the subterms may occur asynchronously.

Term `P - a b` defines the *hiding* of actions `a` and `b` from process `P`.

The Circal process algebra has been extended with additional features and resulted in the XCircal language, which is implemented by the Circal System [13]. In XCircal the application of the structural operators (composition and hiding) needs to be explicitly enforced using the `~` operator. XCircal has the usual primitive data types, such as `int` for the integers and `bool` for the booleans. There are also two datatypes which define *events* and *processes*.

The `Event` data-type implements sets of actions. Atomic events, which are events consisting of a single action, must be declared before the use.

```
Event a, b, c[2]
```

The line above is the declaration of two events `a`, `b` and an array `c` of two events, `c[0]` and `c[1]`. An event consisting of the set of actions `a`, `b` is represented in

Circal as `(a b)`. Singletons may be shortened by removing the parentheses. Thus `a` is the same as `(a)`.

In the following an object of type `Event` will be called a *set of actions* (or an *action*, if it is a singleton) rather than an *event*, and the word *event* will be reserved to the C/E net concept to avoid ambiguity.

The `Process` data-type implements *processes*. A process must be declared before use. An array of processes may be declared. The composition of all the elements of an array `P` of processes is denoted by `*P`. The same operator `*` is used, with a different semantics, to denote the set of all elements of an array of actions. Therefore, given the declaration of `c` above, `*c` is a short form for `(c[0] c[1])`.

Other features of XCircal that are used in this paper are the following:

Control Structures. These have a syntax immediately derived from the programming language C.

Output. Values are sent to the standard output by the `print` function and process behaviours are formatted and sent to the standard output by the `display` function.

Testing Equivalence. The Circal System implements the testing equivalence defined by Moller [15] giving to the expression `P == Q` the result `true` if `P` and `Q` are equivalent, and `false` otherwise. The equivalence checking automatically enforces the application of structural operators without requiring the use of the `~` operator.

The composition operator of Circal provides synchronisation among an arbitrary number of processes as in CSP and LOTOS. This particular nature of the composition operator is exploited by the *constraint-based* modelling methodology [21], which has been used in several application domains such as communication protocols [5], safety-critical systems and asynchronous hardware [6,8].

When a process `P` is composed with a process `Q`, we say that `Q` *constrains* `P` if and only if there is a part of the behaviour of `P` whose restriction to the intersections of the sorts of `P` and `Q` is not consistent with the behaviour of `Q`.

The notion of constraining is used in synthesising complex behaviours by composing simple general behaviours with specific constraints. In Section 6 we will also see how the notion of constraining can characterise behavioural inclusion between processes.

5 Modelling Framework

5.1 Conditions and Cases

In C/E nets conditions can be seen as 1-cell buffers of tokens. A condition can be modelled in Circal by two processes representing the two states, *empty* and *full*, of the 1-cell buffer. We set the `CONDS` XCircal variable to be equal to the maximum number of conditions and we define two arrays `Empty` and `Full`, each consisting of `CONDS` processes. Then, for each i, processes `Empty[`i`]` and `Full[`i`]` model respectively the empty and full states of the $(i+1)$-th condition.

The transition from `Empty[`i`]` to `Full[`i`]` is triggered by the production of a token in the $(i+1)$-th condition, which is modelled by produce action `p[`i`]`. Analogously, the transition from `Full[`i`]` to `Empty[`i`]` is triggered by the consumption of a token from the $(i+1)$-th condition, which is modelled by consume action `c[`i`]`. Figure 1(b) shows how actions `p[`i`]` and `c[`i`]` are associated with conditions and events of the C/E net N_1 given in Figure 1(a). For example, action `c[0]` is in position $\langle e_0, b_0 \rangle$ of the table because an occurrence of e_0 consumes the token in b_0; action `p[1]` is in position $\langle e_0, b_1 \rangle$ of the table because an occurrence of e_0 produces one token in b_1.

The Circal code is given as follows.

```
Event p[CONDS], c[CONDS]
Process Empty[CONDS], Full[CONDS]
for(i=0;i<CONDS;i++) Empty[i] <- p[i] Full[i]
for(i=0;i<CONDS;i++) Full[i] <- c[i] Empty[i]
```

The first line of the Circal code above is the declaration of two arrays of sets of action, `p` and `c`; the second line is the declaration of two arrays of processes, `Empty` and `Full`; the other lines are the definitions of the processes that are elements of arrays `Empty` and `Full`.

Let us consider C/E net N_1 in Figure 1(a). The initial case of N_1 associates tokens only with b_0 and b_3. Thus the producer is ready to send, the buffer is empty and the consumer is ready to receive. Such an initial case is represented in Circal as follows.

```
InitCase1 = Empty[0] * Full[1] * Empty[2] * Full[3] * Empty[4]
```

Process `InitCase1` is the parallel composition of all the processes that define the states of the buffers that implement the conditions of the net. Processes `Full[1]` and `Full[3]` model the two fulfilled conditions, b_1 and b_3, respectively. Processes `Empty[0]`, `Empty[2]` and `Empty[4]` model the three unfulfilled conditions, b_0, b_2 and b_4, respectively.

5.2 Events and Sequential Semantics

In the first two decades of Petri nets' life, their semantics was based on the philosophy that a transition firing is considered to be instantaneous, that is to take zero time. Since time is usually considered as a continuous variable, the probability of any two or more firings happening simultaneously is zero [16]. This is the main argument to support an operational semantics, where transitions cannot fire simultaneously and a single execution of a net can be seen as an interleaving of markings or an interleaving of transitions or an interleaving of alternating markings and transitions [17]. The behaviour of a C/E net can thus be represented as a state graph, called a *reachability graph*, whose nodes are cases and whose arcs are labelled by single events [17].

Given a C/E net $N = \langle B, E, F, C \rangle$ and a case $\bar{C} \subseteq B$, an event e is *enabled* in $\bar{C}$ if and only if each of its preconditions is fulfilled and none of its postconditions is fulfilled. That is $\forall\, b \in B.((b,e) \in F \rightarrow b \in \bar{C}) \wedge ((e,b) \in F \rightarrow \; b \notin \bar{C})$.

If an event e is enabled in $\bar{C}$, then e may occur in $\bar{C}$. The occurrence of e generates a new case C' by consuming tokens from all preconditions of e and producing tokens in all postconditions of e. This is written as $\bar{C}[e\rangle C'$.

We set the EVENTS XCircal variable equal to the maximum number of events and we define an array e consisting of EVENTS actions. The declaration of the e array of actions is as follows.

```
Event e[EVENTS]
```

Since only one event may occur at any time, the behaviour of the net is given by a process consisting of a choice among all enabled events. For C/E net N_1 this is modelled by the EvSeq1 Circal process given as follows.

```
Process EvSeq1
EvSeq1 <- (c[0] p[1] e[0])      EvSeq1 +
          (p[0] c[1] p[4] e[1]) EvSeq1 +
          (p[2] c[3] c[4] e[2]) EvSeq1 +
          (c[2] p[3] e[3])      EvSeq1 +
          (c[4] e[4])           EvSeq1
```

Every possible choice in the definition of EvSeq1 corresponds to a row in Figure 1(b). For instance, the first choice corresponds to the first row, which is associated with event e_0, whose occurrence is modelled by action e[0]. Since the occurrence of e_0 consumes the token from b_0 and produces a token in b_1, then c[0], p[1] and e[0] are forced to occur simultaneously. Their simultaneous occurrence is expressed in Circal by (c[0] p[1] e[0]).

The state graph that defines the behaviour of N_1 can thus be modelled by process EvSeq1Sem, which consists of the parallel composition of InitCase1 and EvSeq1, followed by the hiding of the actions that represent the consumption and production of tokens.

```
EvSeq1Sem = ~(InitCase1 * EvSeq1 - (*c) (*p))
```

Process InitCase1 provides process EvSeq1 with the production and consumption actions that are feasible in the initial case. In this way InitCase1 constrains EvSeq1 to perform the second choice, which is the only one to be feasible in the initial case and corresponds to the occurrence of e_1.

We use the Circal command

```
display EvSeq1Sem
```

to print the behaviour of process EvSeq1Sem, which appears as follows.

```
S0 == e[1] S1
S1 == ((e[0] S2 + e[2] S3) + e[4] S4)
S2 == (e[2] S5 + e[4] S0)
S3 == (e[0] S5 + e[3] S4)
S4 == e[0] S0
S5 == (e[1] S6 + e[3] S0)
S6 == ((e[0] S7 + e[3] S1) + e[4] S3)
S7 == (e[3] S2 + e[4] S5)
```

Such a behaviour is a representation of the reachability graph of N_1.

Every state in the generated behaviour defines a case of N_1. Thus `S0` defines the initial case $C_1 = \{b_1, b_3\}$, `S1` defines $C_{1,1} = \{b_0, b_3, b_4\}$, `S2` defines $C_{1,2} = \{b_1, b_3, b_4\}$. One possible interleaving of alternating cases and events is then: $C_1[e_1\rangle C_{1,1}[e_0\rangle C_{1,2}[e_4\rangle C_1$.

5.3 Parallel Semantics

The sequential semantics of Petri nets resolves the concurrency and the causality expressed by the definition of the Petri net in terms of a non-deterministic interleaving of single event occurrences. However, the underlying assumption that the probability of any two or more events happening simultaneously is zero is too restrictive. A more modern view of Petri net semantics [19] is based on *true concurrency* and emphasises the causality and independence of events, important aspects of any net which are concealed by a sequential semantics. In our paper such a true concurrency semantics is called *parallel semantics*.

Given a C/E net $N = \langle B, E, F, C\rangle$ and a case $\bar{C} \subseteq B$, a set of events $T \subseteq E$ is *enabled* in $\bar{C}$ if and only if the three following properties hold:

– for every event $e \in T$, preconditions of e are fulfilled and no postcondition of e is fulfilled;
– the sets of the preconditions of the events in T are pairwise disjoint;
– the sets of the postconditions of the events in T are pairwise disjoint.

If a set of events T is enabled in $\bar{C}$, then all events in T may simultaneously occur in $\bar{C}$ generating a new case C'. This is written as $\bar{C}[T\rangle C'$.

In order to allow events to occur simultaneously in our Circal model of C/E nets, we need to represent the occurrence of different events as a parallel composition rather than a choice. We set `CONDS1` to the number of conditions in N_1 and we define an array `Cctrl` of processes, one process for every condition. Then we define process `EvPar1` as the parallel composition of the processes that are elements of `Cctrl`.

```
CONDS1 = 5
Process Cctrl[CONDS1]
Cctrl[0] <- (e[1] p[0]) Cctrl[0] + (c[0] e[0]) Cctrl[0]
Cctrl[1] <- (e[0] p[1]) Cctrl[1] + (c[1] e[1]) Cctrl[1]
Cctrl[2] <- (e[2] p[2]) Cctrl[2] + (c[2] e[3]) Cctrl[2]
Cctrl[3] <- (e[3] p[3]) Cctrl[3] + (c[3] e[2]) Cctrl[3]
Cctrl[4] <- (e[1] p[4]) Cctrl[4] + (c[4] e[2]) Cctrl[4] +
                                   (c[4] e[4]) Cctrl[4]
EvPar1 = ~(*Cctrl)
```

The `Cctrl[`i`]` process, which corresponds to condition b_i, consists of all possible choices of consumption of a token from b_i or production of a token in b_i. For instance, let us discuss the definition of process `Cctrl[4]`. A token can be produced in b_4 only by an occurrence of event e_1. This corresponds to the choice `(e[1] p[4]) Cctrl[4]`. A token can be consumed from b_4 either by an occurrence of event e_2 or by an occurrence of event e_4. This corresponds to the other two possible choices `(c[4] e[2]) Cctrl[4]` and `(c[4] e[4]) Cctrl[4]`.

The behaviour of N_1 can thus be represented by a graph whose nodes are cases and whose arcs are labelled by set of events occurring simultaneously. Such a graph is modelled by process `EvPar1Sem`, which consists of the parallel composition of `InitEvent1` and `EvPar1`, followed by the hiding of the actions that represent the consumption and production of tokens.

```
EvPar1Sem = ~(InitCase1 * EvPar1 - (*c) (*p))
display EvPar1Sem
```

The `display` command outputs the behaviour of process `EvPar1Sem` as follows.

```
S0 == e[1] S1
S1 == ((((e[0] S2 + (e[0] e[4]) S0) + (e[0] e[2]) S3) +
      e[2] S4) + e[4] S5)
S2 == (e[2] S3 + e[4] S0)
S3 == ((e[1] S6 + (e[3] e[1]) S1) + e[3] S0)
S4 == ((e[0] S3 + (e[0] e[3]) S0) + e[3] S5)
S5 == e[0] S0
S6 == ((((((e[0] S7 + (e[0] e[4]) S3) + (e[0] e[3]) S2) +
      (e[0] e[3] e[4]) S0) + e[3] S1) + (e[3] e[4]) S5) + e[4] S4)
S7 == ((e[3] S2 + (e[3] e[4]) S0) + e[4] S3)
```

Again, every state in the generated behaviour defines a case of N_1. Now `S0` defines the initial case $C_1 = \{b_1, b_3\}$, `S1` defines $C_{1,1} = \{b_0, b_3, b_4\}$, and `S2` defines $C_{1,2} = \{b_1, b_3, b_4\}$. One possible interleaving of alternating states and sets of events is then: $C_1[\{e_1\}\rangle C_{1,1}[\{e_0, e_4\}\rangle C_1$.

5.4 Read-Only and Overwrite Arcs

The semantics of C/E nets given in the previous section does not allow an event to occur when one of its postconditions is fulfilled. This causes the following implicit restrictions on C/E nets:

1. an event whose set of preconditions and set of postconditions have a non-empty intersection can never occur (*self-loop*);
2. an event with all preconditions fulfilled might be prevented from occurring by one of its postconditions being fulfilled (*contact*).

A semantics which does not associate an active behaviour to a self-loop is very limiting. It is often useful to model a system where the content of a condition can be read without consuming the token. In the C/E net N_2 in Figure 2(a), event e_5 models a test on condition b_4: we would like e_5 to occur when there is a token in b_4, but without consuming the token. This is impossible using the Circal implementation of C/E nets given above. However, we can modify the `Empty` and `Full` arrays of processes defined in Section 5.1 by allowing a simultaneous occurrence of `c[i]` and `p[i]` in `Full[i]`. Therefore, we replace arrays `Full` and `Empty` respectively with arrays `ReadFull` and `ReadEmpty` defined as follows.

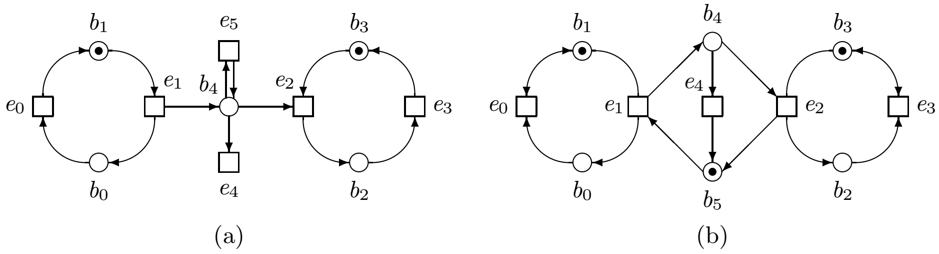


Fig. 2. (a): A C/E net N_2 defining a producer-consumer system with test on the buffer; (b): A C/E net N_3 defining a contact-free producer-consumer system

```
Process ReadEmpty[CONDS], ReadFull[CONDS]
for(i=0;i<CONDS;i++) ReadEmpty[i] <- p[i] ReadFull[i]
for(i=0;i<CONDS;i++) ReadFull[i] <- c[i] ReadEmpty[i] +
                                    (c[i] p[i]) ReadFull[i]
```

Term `(c[i] p[i]) ReadFull[i]` now allows an event to simultaneously consume and produce a token in the same condition, which is equivalent to a read-only operation on that condition. The initial case of N_2 allowing active self-loops is then given as follows.

```
InitCase2R = ReadEmpty[0] * ReadFull[1] *
             ReadEmpty[2] * ReadFull[3] * ReadEmpty[4]
```

The choice among all enabled events is modelled by the `EvSeq1R` Circal process given as follows.

```
Process EvSeq1R
EvSeq1R <- (c[0] p[1] e[0]) EvSeq1R +
   (p[0] c[1] p[4] e[1]) EvSeq1R +
   (p[2] c[3] c[4] e[2]) EvSeq1R + (c[2] p[3] e[3]) EvSeq1R +
   (c[4] e[4]) EvSeq1R + (c[4] p[4] e[5]) EvSeq1R
EvSeq1RSem = ~(InitCase2R * EvSeq1R - (*c) (*p))
```

The last term of the definition of `EvSeq1R` models that event e_5 (implemented by `e[5]`) simultaneously consumes a token (action `c[4]`) from and produces a token (action `p[4]`) to b_4.

In order to allow an event to occur even though at least one of its postconditions is fulfilled, we define arrays `OWFull` and `OWEmpty` as follows.

```
Process OWEmpty[CONDS], OWFull[CONDS]
for(i=0;i<CONDS;i++) OWEmpty[i] <- p[i] OWFull[i]
for(i=0;i<CONDS;i++) OWFull[i] <- c[i] OWEmpty[i] + p[i] OWFull[i]
```

Term `p[i] OWFull[i]` in the definition of process `OWFull[i]` now allows an event to produce a token in a fulfilled condition. When defining the initial case of a given C/E net, we will then use the `Full[k]` and `Empty[k]` processes for each non-overwritable condition b_k and the `OWFull[j]` and `OWEmpty[j]` processes for each overwritable condition b_j. For instance, if we want buffer overwriting in the C/E net N_1 in Figure 1(a), we need to make condition b_4 overwritable. The initial case is then represented in Circal as follows.

```
InitCase1OW = Full[0] * Empty[1] * Full[2] * Empty[3] * OWEmpty[4]
```

We will see an application of overwritable conditions in Section 6.1.

We can also have both read-only arcs and overwrite conditions by using the following processes.

```
Process ROWEmpty[CONDS], ROWFull[CONDS]
for(i=0;i<CONDS;i++) ROWEmpty[i] <- p[i] ROWFull[i]
for(i=0;i<CONDS;i++) ROWFull[i] <- c[i] ROWEmpty[i] +
                        p[i] ROWFull[i] + (c[i] p[i]) ROWFull[i]
```

The initial case is for C/E net N_2 in Figure 2(a), with read-only in self loops and condition b_4 overwritable is implemented in Circal as follows.

```
InitCase2RW4 = ReadEmpty[0] * ReadFull[1] *
               ReadEmpty[2] * ReadFull[3] * ROWEmpty[4]
```

6 Analysis of Properties

A correctness concept that can be readily characterised in Circal is the behavioural equivalence `P == Q` between two given processes `P` and `Q`, which is implemented by the Circal System.

However, in performing formal verification equivalence is often too strong a property. Analysing systems often consists of determining that certain properties hold, where these properties do not constitute a complete specification. Concurrent systems frequently require us to determine if the behaviour of a given process `Q` is included in the behaviour of another process `P`.

The constraint-based modelling technique supports a clear characterisation of such a behavioural inclusion. The behaviour of `Q` is strictly included in the behaviour of `P` if and only if the following three properties hold:

1. `P` and `Q` have the same sort;
2. `Q` constrains `P`;
3. `P` does not constrain `Q`

We want to prove that if `P` and `Q` are not equivalent and Condition 1 holds, then Conditions 2 and 3 hold if and only if the equivalence check `Q * P == Q` gives `true` as a result. If `P` and `Q` are not equivalent, then the equivalence above gives `true` as a result if and only if the part of the behaviour of `P` that is not consistent with the behaviour of `Q` disappears after the composition (that is Condition 2 holds) and the behaviour of `Q` is preserved after the composition (it is fully consistent with the behaviour of `P`, that is Condition 3 holds).

As an example of behavioural inclusion we would like to verify for the C/E net N_1 given in Figure 1(a) that the behaviour defined by the parallel semantics introduced in Section 5.3 strictly includes the behaviour defined by the sequential semantics introduced in Section 5.2. We first verify that the two semantics are not equivalent. The equivalence check

```
EvSeq1 == EvPar()
```

gives `false` as a result.

The two semantics have the same sort. Thus we check that the following equivalence is true.

```
EvSeq1 * EvPar() == EvSeq1
```

Notice that this result is independent of the initial case of N_1.

The behavioural inclusion checking allows the analysis of interesting properties of concurrent systems. Together with the equivalence checking and with other techniques for safety [5] and performance [6,8] analysis introduced in previous papers it may be used for the verification of systems in part modelled as C/E nets and in part directly modelled as Circal processes. In the next section, we present an example of verification of a simple property of C/E nets.

6.1 Contacts and Complementation

We have said in Section 5.4 that an event with all preconditions fulfilled might be prevented from occurring by one of its postconditions being fulfilled. This situation is called a *contact*. More precisely, in a C/E net there is a contact if and only if all preconditions and at least one postcondition of an event are fulfilled. A C/E net is *contact-free* if and only if a contact can never occur.

It might be considered not very elegant that the occurrence or non-occurrence of an event depends on both preconditions and postconditions [20, p.16]. This is an argument for avoiding contact situations in design. It is indeed possible to remove a situation of contact by adding additional conditions to the C/E net. In order to remove contacts, we introduce the concept of a *complement* to a condition.

In a C/E net a condition b' is a *complement* to a condition b if and only if for every event e the following properties hold:

- b is a precondition of e if b' is a postcondition of e;
- b is a postcondition of e if b' is a precondition of e;
- b' is unfulfilled in the initial case if and only if b is fulfilled in the initial case.

It is easy to show that if b' is a complement of a condition b, exactly one of the two conditions is fulfilled in each case. Therefore, adding a complement to a condition does not change the behaviour of the net.

A C/E net can be made contact-free by adding a complement to every postcondition that causes a contact.

In the C/E net N_1 in Figure 1(a) the occurrence of e_1 followed by the occurrence of e_0 results in a case with tokens both in b_1 and b_4. Event e_1 has its only precondition b_1 fulfilled, but it cannot occur because its postcondition b_4 is also fulfilled. This is a contact situation.

In order to detect the contacts in N_1 using the Circal System, we define the `OverWrite1` array of processes such that, for each i, process `OverWrite1[`i`]` is obtained by replacing in the initial case the `Empty[`i`]` and `Full[`i`]` processes respectively with the `OWEmpty[`i`]` and `OWFull[`i`]` processes defined in Section 5.4. For example process `OverWrite1[1]` is defined as follows.

```
OverWrite1[1] = Empty[0] * OWFull[1] * Empty[2] * Full[3] *
                Empty[4] * EvSeq1 - (*c) (*p)
```

We can now detect all contacts in N_1 through the following equivalence checks

```
for(i=0;i<CONDS1;i++){print(EvSeq1Sem == OverWrite1[i]); print" "}
```

which give as result: `true true true true false`. For each i, `OverWrite1[`i`]` is equivalent to `EvSeq1Sem` if and only if the `p[`i`]` action may not occur in `OWFull[`i`]`. Action `p[`i`]` may occur in `OWFull[`i`]` if and only if the token in b_i is overwritten by the occurrence of an event having b_i as a postcondition, that is when b_i being fulfilled causes a contact. Thus the equivalence of `OverWrite1[`i`]` and `EvSeq1Sem` holds if and only if there is no contact caused by b_i. Therefore, the equivalence checks above show that in N_1 contacts are only caused by b_4. This contact can be removed by adding a complement to condition b_4 as in C/E net N_3 given in Figure 2(b). Since b_4 was the only cause of contact, N_3 is contact-free. The Circal sequential model of N_3 is given as follows.

```
InitCase3 = Empty[0] * Full[1] * Empty[2] *
            Full[3] * Empty[4] * Full[5]
Process EvSeq3
EvSeq3 <- (c[0] p[1] e[0]) EvSeq3 +
          (c[1] c[5] p[0] p[4] e[1]) EvSeq3 +
          (c[3] c[4] p[2] p[5] e[2]) EvSeq3 +
          (c[2] p[3] e[3]) EvSeq3 +
          (c[4] p[5] e[4]) EvSeq3
EvSeq3Sem = ~(InitCase3 * EvSeq3 - (*c) (*p))
```

Now, if we build the `OverWrite3` process for N_3 in the same way as we have built the `OverWrite1` process for N_1 and we set `CONS3` equal to 6, then the following equivalence checks

```
for(i=0;i<CONDS3;i++){print(EvSeq3Sem == OverWrite3[i]); print" "}
```

give as result: `true true true true true true`. This proves that N_3 is contact-free. We can also check that N_3 has the same behaviour as N_1 through the following equivalence.

```
EvSeq1Sem == EvSeq3Sem
```

7 Conclusion and Future Work

We have presented a technique for implementing Condition/Event nets in a process algebra. We have chosen the Circal process algebra, which has the distinctive feature of allowing simultaneity of actions. We have exploited such a feature to implement a true concurrency semantics as an alternative to the implementation based on non-deterministic interleaving.

We have also seen that our implementation allows the design of systems modelled by C/E nets and the verification of general properties of C/E nets, such as equivalence and behavioural inclusion, or more specific properties, such as contact-freedom. Moreover, having an implementation of C/E nets within a process algebra allows the use of a unique verification engine in the analysis of systems which are modelled using different specification formalisms such as Petri nets and Finite State Machines.

All the Circal code used in this paper can be downloaded from the World Wide Web [3]. In order to run it it is necessary to install the Circal System [1]. Our implementation of C/E nets is exploited by the *Petrinette* graphical design and verififcation tool [9,3].

A natural continuation of this work is its extension to Place/Transition nets (P/T nets) and higher-level Petri nets. Circal allows the modelling of finite state systems, whereas P/T nets may have infinite states. Thus P/T nets cannot be translated into a finite state process algebra without an abstraction step which reduces the behaviour to a finite state system. Therefore, in order to have a complete translation, we must restrict our work to bounded P/T nets. However, the technique presented in this paper cannot be directly applied to bounded P/T nets. The main change to the modelling framework involves the implementation of places. Conditions in C/E nets have been implemented using 1-cell buffers, but the implementation of a place in a P/T net would require a buffer consisting of at least as many cells as the capacity of the buffer. Moreover we have to implement a test on the number of tokens in the buffer to check the enabling of transitions. The main problem is how to exploit the result of the test taking into account possible conflicts among transitions. The problem becomes even more complex if we allow the simultaneous firing of distinct transitions or even multiple concurrent firings of the same transition. Investigating all these problems is part of our current research.

Finally, we would like to point out that the Circal feature of allowing the simultaneous occurrence of distinct actions has eased the definition of the modelling framework presented in Section 5 and made possible the definition of the parallel semantics presented in Section 5.3. This is further evidence that the simultaneity of actions enriches the ability to model aspects of concurrent system which are believed hard to characterise in process algebra frameworks. Among these aspects we have already investigated priorities among actions [4,5], the modelling of dense time [7] and the verification of performance properties [6,8]. All such works exploit simultaneous actions in modelling systems and verifying properties. In our future work we plan to investigate the use of such a feature in further application domains, such as multitasking, security and mobility.

Acknowledgements

I would like to thank Colin Fidge and Nathan Spargo for helpful comments.

References

1. Circal System. Web Page, 2001.
   `http://www.acrc.unisa.edu.au/doc/circal/circal_distribution.html`.
2. T. Bolognesi and E. Brinksma. Intoduction to the ISO specification language LOTOS. *Computer Systems and ISDN Systems*, 14(1):25–59, 1987.
3. A. Cerone. Modelling Petri nets in Circal. Web Page, 2001.
   `http://www.itee.uq.edu.au/~antonio/Research/Misc/pntocir.html`.
4. A. Cerone, A. J. Cowie, G. J. Milne, and P. A. Moseley. Description and verification of a time-sensitive protocol. Technical Report CIS-96-009, University of South Australia, Adelaide, 1996.
5. A. Cerone, A. J. Cowie, G. J. Milne, and P. A. Moseley. Modelling a time-dependent protocol using the Circal process algebra. In *Proc. of HART97*, volume 1201 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1997.
6. A. Cerone, D. A. Kearney, and G. J. Milne. Integrating the verification of timing, performance and correctness properties of concurrent systems. In *Proc. of the Int. Conf. on Application of Concurrency to System Design (CSD'98)*, pages 109–119. IEEE Comp. Soc. Press, 1998.
7. A. Cerone and G. J. Milne. Specification of timing constraints within the Circal process algebra. In *Proc. of AMAST97*, volume 1349 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 1997.
8. A. Cerone and G. J. Milne. A methodology for the formal analysis of asynchronous micropipelines. In *Proc. of FMCAD00*, volume 1954 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2000.
9. A. Cerone and N. Spargo. Petrinette: A tool for the integrated analysis of Petri nets and finite state machines. Technical Report 01-39, Software Verification Research Centre, The University of Queensland, Brisbane, 2001.
10. T. Chu, C. Leung, and T. Wanuga. A design methodology for concurrent VLSI systems. In *Proc. of ICDD*, pages 407–410, 1985.
11. Formal methods. Web Page, 2001. `http://www.afm.sbu.ac.uk/fm/`.
12. C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
13. G. J. Milne. *Formal Specification and Verification of Digital Systems.* McGraw Hill, 1994.
14. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
15. F. Moller. The semantics of Circal. Technical Report HDV-3-89, University of Strathclyde, Glasgow, UK, 1989.
16. J. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, 1977.
17. J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.
18. C. A. Petri. Kommunication mit automaten. Schrift der IIM Nr. 2, Institut für Instrumentelle Mathematik, University of Bonn, 1962. *English translation*: Technical Report RADC-TR-65-337, Griffiths Air Force Base, New York, 1966.
19. W. Reisig. *Petri Nets — An introduction.* Springer, 1985.
20. W. Reisig. *A Primer in Petri Nets Design.* Springer, 1992.
21. C. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.