1. Introduction
===============

This Report deals with decompositions of asynchronous circuits into modular networks, and their verification, using the high-level specification language LOTOS and its associated toolbox CADP, as well as Petri nets and the related tool PETRIFY.

We assume some familiarity with [EDIS], dealing with related topics, as well as LOTOS/CADP and Petri Nets/PETRIFY (see [YE2001]).

We use A,B,C to denote Boolean input lines, and X,Y,Z to denote Boolean output lines. A Boolean line, e.g. A, may assume the value T(true), representing logic-1, and F(false), representing logic-0. We use corresponding lower-case letters, e.g. 'a', to denote level changes. E.g. a+ will denote an up-transition of line A (a change from logic-0 to logic-1) and a- a down-transition. The symbol 'a' will represent both an up- as well as a down transition of the line A.

As is usual in connection with asynchronous circuits, we are interested in their "edge" (level-change) based behavior, rather than in their level-based behavior. The description of a circuit will also take into account the restrictions imposed on the environment, in order to achieve a predictable circuit behavior.

In some of the applications, particularly CADP-based, lower-case symbols are replaced by their capitals.

2. Notations
============

In the sequel we use state graphs to describe various modules. The notation used is rather evident. If not stated otherwise, s0 is the initial state.

We append '?' to indicate inputs, and '!' for outputs.

k-cycles
========

cy2.sg[a?,z!] := (s0 a? s1 z! s0)
This 2-cycle is also denoted by *[a?,z!].

k-cycles, k>2, are defined and denoted similarly.

parallel connection
====================

we use '||' to denote parallel composition with synchronization on all shared labels, and '||||' for parallel composition without synchronization (see [YE2001]).

realization
===========

we write IMPL|= SPEC to state that IMPL is a realization of SPEC (see [YE2001]).

hiding
======

we use expre\{x,...,y} to "hide" the variables {x,...,y} in expre.

3. Basic Modules
================

Here we introduce the basic modules used in the sequel. For informal explana-
The usual XOR-gate has Boolean inputs A, B and Boolean output Z. The gate is stable iff \( Z \Leftrightarrow \neg(A = B) \). We impose the following restrictions on the environment:

(i) only one input may change at a time;
(ii) an input may change only if the gate is stable.

Thus the edge-based behavior of this gate, including its environment restrictions, may be represented by the following state graph.

\[
\text{XOR.sg:= (s0 a s1 z s0 b s1)}
\]

LOTOS Representation

\[
\begin{align*}
\text{specification XOR[A,B,Z]: noexit behaviour} \\
\text{XOR[A,B,Z]} \\
\text{where} \\
\text{process XOR[A,B,Z]:noexit:=} \\
\text{A;Z,XOR[A,B,Z] [] B;Z,XOR[A,B,Z]} \\
\text{endproc} \\
\text{endspec}
\end{align*}
\]

The corresponding omin-file (see [YE2001]) becomes:

\[
\begin{align*}
\text{des (0, 3, 2)} \\
\text{(0,A,1)} \\
\text{(0,B,1)} \\
\text{(1,Z,0)}
\end{align*}
\]

PETRIFY Representations

The above XOR module is represented in PETRIFY as follows.

\[
\begin{align*}
\text{.model xor.net} \\
\text{.inputs a b} \\
\text{.outputs z} \\
\text{.graph} \\
p0 a b \\
p1  \\
p1 a b \\
p1 z \\
z p0 \\
\text{.marking (p0)} \\
\text{.end}
\end{align*}
\]

In the above net model, the place p0 represents all stable gate states (i.e., \( Z \Leftrightarrow \neg(A = B) \)), and place p1 all unstable gate states (i.e., \( Z \Leftrightarrow (A = B) \)).

This net model evidently coincides with the above state graph representation.

A 3-input XOR-gate (XOR3 = 3-MERGE) is represented in PETRIFY similarly:
3.2 CEL = JOIN

A CEL-circuit is sequential. It has Boolean inputs A, B and Boolean output Z. It is stable iff \( \neg (A=B) \) or \( A=B=Z \). We assume the circuit is started in an \( A=B=Z \) state; also an input may change only if its value equals that of the output. The states \( \neg (A=B) \) are memory states. The output is the same as its previous value. Indeed, the CEL-circuit has a similar function as a SET-RESET flip-flop. The SET command is \( A=B=T \), and the RESET command is \( A=B=F \).

The CEL-circuit may be realized by means of a MAJORITY-gate implementing the Boolean function \( Z \leftrightarrow ((A \land B) \lor (A \land C) \lor (B \land C)) \), (also known as the full-adder carry-out function). In a MAJORITY-gate the output is evidently determined by the majority of the input values. A CEL-circuit can be obtained from a MAJORITY-gate by providing a feedback connection from Z to the input C, which consequently is no longer an independent input.

The CEL-circuit is easiest represented by its state graph, shown below, using PETRIFY style.

If the CEL-circuit is started in its \( A=B=Z=F \) state, it first behaves like an AND-gate. But once the \( A=B=Z=T \) state is reached, this similarity is no longer valid.

LOTOS Representation

**specification CEL[A,B,Z]:noexit behaviour**

```
CEL[A,B,Z]
where
process CEL[A,B,Z]:noexit:=
  A;B;Z;CEL[A,B,Z]
  [ ]
  B;A;Z;CEL[A,B,Z]
endproc
eendspec
```

The corresponding omin-file is shown below.
PETRIFY Representation

To convert the above state graph cel.sg into a Petri net, we issue the PETRIFY command: petrify cel.sg -o cel.net and get the net below.

```
.model cel.net
.inputs  a b
.outputs  z
.graph
   a z
   z a b
   b z
.marking {<z,a> <z,b>}
.end
```

Note that the above net is a marked graph. In the above representation all places, which are all implicit, have been eliminated. The following command will reinsert all implicit places:

```
petrify -ip cel.net -o cel.ipnet
```

The outcome is the following net cel.ipnet:

```
.model cel.ipnet
.inputs  a b
.outputs  z
.graph
   a p0
   z p1 p2
   b p3
   p0 z
   p1 a
   p2 b
   p3 z
.marking {p1 p2}
.end
```

CEL3 = 3-JOIN

A 3-input CEL-gate is easiest specified by the following marked graph.

```
.model cel3.net
.inputs a b c
.outputs z
.graph
   a z
   b z
   c z
   z a b c
.marking {<z,a> <z,b> <z,c>}
```
Try now the PETRIFY command: write_sg cel3.net −o cel3.sg.
The outcome is an 8-state state graph. This illustrates the fact that in
general, net representations are more concise than state graphs.

LOTOS Specification
===================

specification CEL3SP[A,B,C,Z]:noexit behaviour

CEL3SP[A,B,C,Z]
where
process CEL3SP[A,B,C,Z]:noexit:=
    (A;exit ||| B;exit)|||C;exit)>> Z;CEL3SP[A,B,C,Z]
endproc
endspec

The relevant omin-file is shown below.

***********
(0, 13, 8)
(0, B, 4)
(0, A, 5)
(0, C, 3)
(1, B, 7)
(2, A, 7)
(3, B, 2)
(4, A, 1)
(4, C, 2)
(5, B, 6)
(5, C, 1)
(6, C, 7)
(7, Z, 0)
***********

3.3 TOGGLE
==========

This is yet another basic module we are interested in.
It has one Boolean input A and two Boolean outputs Y,Z. Its behaviour may
be described, using a bidirectional edge representation, by (a;y;a;z)*
(where * denotes "repeat forever"), or by the unidirectional edge represen-
tation (a+y+a−;a−;y+a−;y−+a−)*. A TOGGLE-circuit is easily realized, using
conventional logic design techniques.

Below are the relevant LOTOS specification and the corresponding TOG.aut file.

specification TOG[A,Y,Z]: noexit behaviour

TOG[A,Y,Z]
where
process TOG[A,Y,Z]:noexit:=
    A;Y;A;Z;TOG[A,Y,Z]
endproc
endspec

TOG.aut
**********

(0, 4, 4)
(0, A, 1)
(1, Y, 2)
PETRIFY Representations

The following is a PETRIFY-based state-graph representation:

```
.model tog.sg
.inputs a
.outputs y z
.state graph
.s0 a s1
.s1 y s2
.s2 a s3
.s3 z s0
.marking {s0}
.end
```

A net representation can be obtained by means of the command:

```
petrify tog.sg -o tog.net
```

producing the following net

```
.model tog.net
.inputs a
.outputs y z
.graph
.p0
.p1 z
.p1 y
.p1 y z
.p1 a
.marking { p1 <z,y> }
.end
```

Note that PETRIFY tends to produce uniquely-labelled nets (see [YE2001]), whenever possible. However, the trace equivalence between tog.sg and tog.net is not immediately evident. A less concise, but more state-graph oriented representation, can be obtained by using the -er option. The command: petrify -er tog.sg -o ertog.net produces the following net:

```
.model ertog.net
.inputs a
.outputs y z
.graph
.a y
.y a/1
.a a
.a/1 z
.marking { <z,a> }
.end
```

The above net is multiple-labelled (see [YE2001]). Indeed, it is isomorphic to the state graph tog.sg.

3-TOGGLE

A 3-output TOGGLE (3-TOGGLE) can be specified by the behavior expression

```
(a;x;a;y;a;z)*
```

The corresponding LOTOS and PETRIFY descriptions are easily derived.
4.1 LOTOS/CADP-Based

Let IMPL and SPEC denote LOTOS-processes, representing edge-based descriptions of the implementation and the specification of an asynchronous circuit. We assume that the two processes share the same alphabet (i.e., set of observable events/actions), and that this alphabet is partitioned into inputs and outputs.

We say that IMPL realizes SPEC (notation: IMPL |= SPEC) iff the following conditions are satisfied.

Cond1: SPEC||IMPL is obs.equivalent to SPEC.
Cond2: IMPL is live-lock free.
Cond3: No "undesirable" outputs (see below). One way to verify this condition is as follows.

Let iIMPL be the process obtained from IMPL by replacing each output, say z, by i;z. Then SPEC||iIMPL is deadlock-free. Frequently this condition can be verified by preferable ad-hoc methods.

4.1.1 Informal Motivation

Cond1 ensures that IMPL is at least as powerful as SPEC. Any behaviour specified by SPEC can be performed by IMPL, disregarding i-transitions occurring between observable events.
Cond2 assures that IMPL does not enter a cycle of i-transitions.
Cond3 prevents "undesirable" outputs to occur in IMPL. Let w1 be an action sequence of IMPL, followed by an output z. Assume that w1 is obs.equivalent to an action sequence of SPEC. Then there exists such an action sequence w2 in SPEC, obs.equiv. to w1, such that w2 is followed by z in SPEC.

4.1.2 Verifying Conditions 1-3

Cond1:

Method (1). Generate SIMPL:= SPEC||IMPL. Then convert the LOTOS-program SIMPL.lotos into the LTS SIMPL.aut, using the command
caeasar -aldebaran SIMPL.lotos
Similarly convert SPEC.lotos into SPEC.aut.
Then apply the command
aldebaran -oequ SIMPL.aut SPEC.aut
expecting the output "TRUE".
Method (2). Generate SPEC.aut and IMPL.aut. Then obtain SIMPL.aut= SPEC.aut||IMPL.aut (see ALDEBARAN manual!).
Check obs.equivalence as above.

Cond2:

This condition can be checked by means of the following command:
aldebaran -live filename.aut

Cond3:

Generate iIMPL.lotos, or alternatively iIMPL.aut directly. Then check whether iIMPL||SPEC is deadlock-free, using the command
aldebaran -dead filename.aut
Other methods will be illustrated later on.

4.2 Petri-Net/PETRIFY - Based

We now assume that we are given a specification SPN of an asynchronous circuit in form of a labelled Petri net, as well as a decomposition (realization) of this specification into a network of basic modules.
Let IMPN be a labelled net describing such a decomposition. We wish to formalize the concept "IMPN realizes SPN" (notation: IMPN |= SPN), and then to show how PETRIFY may be used for the verification of such decompositions. Related concepts, but quite different techniques, are discussed in [EDIS].

4.2.1 The Concept of Realization

We now assume that we are given a specification SPN of an asynchronous circuit in form of a labelled Petri net, as well as a decomposition (realization) of this specification into a network of basic modules. Let IMPN be a labelled net describing such a decomposition. We wish to formalize the concept "IMPN realizes SPN" (notation: IMPN |= SPN), and then to show how PETRIFY may be used for the verification of such decompositions. Related concepts, but quite different techniques, are discussed in [EDIS].
Let SPN and IMPN be as above. We assume that the labels of the two nets are partitioned into three parts: input labels, output labels, and internal labels. Furthermore, the input and output labels of the two nets coincide.

Environment-Restrictions Net
============================
This net, denoted env.SPnet, is obtained from SPN by omitting all the edges incoming to transitions labelled by output labels.

Definition of IMPN |= SPN
==========================
IMPN |= SPN iff
(1) (env.SPnet || IMPN) obs.equiv. SPN
(2) IMPN does not contain a cycle of internal labels.

The parallel net composition (||) and the concept of obs.equivalence are defined in [YE2001].

In [YG99] the above definition is related to the corresponding automata-based definition, and the two definitions are proven to be equivalent.

5: Modular Decompositions (LOTOS/CADP-based)
=============================================
XORk-gates can easily be decomposed into XORj-gates, where j<k. A similar statement applies to CELk-gates. For details see [EDIS].

To illustrate our approach, we show how the decomposition of CEL3 into CEL2 modules can be described and verified.

Below is a lotos-file describing the above decomposition (i.e., realization).

File cel3impl.lotos
===================
specification cel3impl[A,B,C,Z]:noexit behaviour
  c3i[A,B,C,Z]
where
  process c3i[A,B,C,Z]:noexit:=
    hide R in
    cel[A,B,R]|[R]|cel[R,C,Z]
  endproc
  process cel[A,B,Z]:noexit:=
    A;B;Z;cel[A,B,Z]
    []
    B;A;Z;cel[A,B,Z]
  endproc
endspec

We wish to prove that cel3impl |= cel3.
To verify Cond1 we generate cel3impl || cel3.
This is done in the following file.

File cel3simpl.lotos
====================
specification cel3simpl[A,B,C,Z]:noexit behaviour
  c3i[A,B,C,Z]||c3sp[A,B,C,Z]
process c3i[A,B,C,Z]:noexit:=
  (see previous file)
endproc
process c3sp[A,B,C,Z]:noexit:=
  (A;exit ||| B;exit ||| C;exit)>>Z;c3sp[A,B,C,Z]
endproc
endspec

We then proceed as discussed in Section 4, Cond1/Method (1). Cond2 is similarly verified (see Section 4).
To verify Cond3, we replace in file cel3simpl.lotos the two ;Z entries by ;i;Z. We convert this extended file into its aut-file,
and verify the no-deadlock condition.

Decompositions of XOR_{k}, k>2 and CEL_{k}, k>3 can be specified and verified similarly.

6: Modular Decomposition (PETRIFY-Based)

We now wish to deal with the preceding decomposition, using PETRIFY.

For cel.net, as defined above, we set cel.net=cel[a, b, z].

Set SPN=cel3.net and IMPN=(cel[a, b, r] || cel[r, c, z]) \{r\}. Here \{r\} indicates that r is to be considered a "dummy" variable.

Generating env.SPnet

SPN=cel3.net is shown below.

```
.model cel3.net
.inputs a b c
.outputs z
.graph
  a z
  b z
  c z
  z a b c
.marking {<z,a> <z,b> <z,c>}
.end
```

To get env.SPnet we simply omit from cel3.net the three lines ‘* z’.

Also, since ‘z c’ is already included in cel[r, c, z] it should be omitted from env.SPnet

Generating IMPN || env.SPnet = eSIMP.net

```
.model eSIMP.net
.inputs a b c
.outputs z
dummy r
.graph
  #cel[a,b,r]
  a r
  b r
  r a b
  #cel[r,c,z]
  r z
  c z
  z r c
.env
  a b
.marking {<r,a> <r,b> <z,c> <z,r> <z,a> <z,b>}
.end
```

To hide the dummy variable r, we issue the command:

```
petrify -hide r eSIMP.net -o res.net
```

The resulting net is shown below:

```
.model res.net
.inputs a b c
.outputs z
.graph
  a z
  c z
  b z
  z c b a
.marking {<z,b> <z,c> <z,a>}
.end
```
Thus, res.net is isomorphic to the specification net cel3.net. Since IMPN contains only a single dummy transition, condition (2) of Section 4.2.1 is evidently satisfied. Thus the statement IMPN |= SPN has been proven.

7. Transition Counters
==================================
In this section we introduce the concept of "Modulo-N Transition Counter", and indicate methods of synthesis, using the modules XOR and TOGGLE. This section is mainly based on [EP92]. The synthesis methods referred to in this section, will be used in the sequel, to illustrate our verification method, outlined above. We write \( w^* \) to denote "repeat \( w \) forever".

7.1 Specification
-------------------
A modulo-N (transition) counter can be specified as follows:

- **Inputs:** A
- **Output:** Y, Z
- **Behaviour:** \( \text{cnt}.N[a,y,z] := ((a;y;)^{(N-1)}a;z) \)

where \( w^N \) denotes the sequential repetition of \( w \), \( N \) times. For example, \( \text{cnt}.3[a,y,z]=(a;y;a;y;a;z)^{**}(N-1)a;z) \)

Note that the module TOGGLE coincides with the modulo-2 transition counter.

7.2 - Decompositions
---------------------
In accordance with [EP92], the modulo-N counter, for even \( N>2 \), can be decomposed into a modulo-\( N/2 \) counter, a TOGGLE, and a XOR-gate, as shown below.

**Proposition 7.2.1**

\[
\text{cnt}.N[a,y,z] = ((\text{cnt}.N/2[a,p,q] |[q]| \text{TOG}[q,x,z])
\]
\[
|\{p,x\}| \text{XOR}[p,x,y] \}
\]

Here, \( \{p,q,x\} \) indicates the "hiding" of \( p,q,x \), i.e., their declaration as dummy variables.

For odd \( N>2 \), the decomposition is as follows.

**Proposition 7.2.2**

\[
\text{cnt}.N[a,y,z]=((\text{cnt}.(N+1)/2[r,y,q] |[q]| \text{TOG}[q,s,z])
\]
\[
|\{r,s\}| \text{XOR}[a,s,r] \}
\]

Furthermore, the following decomposition rule is rather evident.

**Proposition 7.2.3**

Let \( N=N1xN2 \), where \( N1>2 \), \( N2>2 \).

Then \( \text{cnt}.N[a,y,z] = (\text{cnt}.N1[a,y,q] |[q]| \text{cnt}.N2[q,y,z]) \)

Although the above decomposition rules can easily be proven correct, we wish to use them for the purpose of illustrating our approaches to the formal verification of modular, asynchronous circuits.

7.3 - Verification of Modulo-3 Transition Counter (LOTOS/CADP)
===============================================================
This counter may be specified by the following state graph:

\[
\text{mod3tc.sg}:= (s0 \ a \ s1 \ y \ s2 \ a \ s3 \ y \ s4 \ a \ s5 \ z \ s0)
\]

The specification of this counter, using LOTOS, is provided in the following file.

File mod3cntsp.lotos
specification mod3count_sp\([A,Y,Z]\):noexit behaviour
\[Q[A,Y,Z]\]

where
process Q[A,Y,Z]:noexit:=
\[A;Y;A;Y;A;Z;Q[A,Y,Z]\]
endproc
endspec

Its implementation is shown below.

File mod3count.lotos

specification mod3count\([A,Y,Z]\):noexit behaviour
mod3count\([A,Y,Z]\)

where
process mod3count\([A,Y,Z]\):noexit:=
\[\text{hide } R,Q,S \text{ in } XOR[A,S,R] |[R,S]| (\text{toggle}[R,Y,Q] |[Q]| \text{toggle}[Q,S,Z])\]
endproc
process XOR\([A,B,Z]\) : noexit :=
\[A;Z;XOR\([A,B,Z]\)\]
\[]
\[B;Z;XOR\([A,B,Z]\)\]
endproc
process toggle\([A,Y,Z]\):noexit:=
\[A;Y;A;Z;\text{toggle}[A,Y,Z]\]
endproc
endspec

We now proceed to prove IMPL|= SPEC, where IMPL and SPEC denote
the above implementation and specification. Thus, we have to show
that Conditions C1,C2,C3 are satisfied.

Conditions C1,C2

C1 is easily checked, using either Method (1) or Method (2) of
Section 4. Also C2 can be checked as explained in Section 4.

Condition C3

In this example the application of the method discussed in Section 4
is not convenient. A reasonable alternative is to generate mod3count.
omin. Following the (unique) sequence \(A;Y;A;Y;A;Z\), leading from state
back to state 0, one immediately verifies that no undesirable out-
put is produced.

Using the above propositions, mod-N transition counters for \(N>3\) are
easily designed. Such counters can then be verified, following the
above example.

7.4 PETRIFY-Based Verification

Using PETRIFY, the state graph specification of this counter becomes:

File mod3cntsp.sg

------------------------
.model mod3cntsp.sg
.inputs a
.outputs y z
.state graph
s0 a s1
s1 y s2
s2 a s3
s3 y s4
We use PETRIFY to get the following specification net:

```plaintext
.model mod3cntsp.net
.inputs a
.outputs y z
.graph
  a z
  z y/1 p0
  a/1 p1
  y/1 p0 y
  p0 a/1
  y y/1
.marking { <z,y/1> p0 }
.end
```

From the above net we derive its environment-restrictions net envcnt3sp.net by omitting all connections to output transitions, as well as connections to places which lost their out-connections. The result is the following net "envres.net":

```plaintext
.model envcnt3sp.net
.inputs a
.outputs y z
.graph
  a
  p0
  y1 p0
  p0 a/1
.marking {p0}
.end
```

Our next task is to generate the parallel connection between the implementation net and the envres.net. Since this envres.net is multiple-labelled, we have to "split" (see [YE2001]) the corresponding transitions ('a' and 'y') in the implementation net. For this purpose, we first issue the command:

```bash
petrify -ip mod3cnt.net -o mod3cnt.ipnet
```

In order to insert all implicit places into the implementation net, the outcome is:

```plaintext
.model mod3cnt.ipnet
.inputs a
.outputs y z
.dummy r q s
.graph
  p3 p6
  p2 p9 p4
  p1 p9
  z p5 p7
  s p8 p7 p0
  a p8
  p0 z
  p1 q
  p2 z s
  p3 q y
  p4 y
  p5 s
  p6 s a
  p7 q
  p8 r
  p9 r
```
The outcome of the required parallel connection is shown in file cnt3res.net below:

```
.model cnt3res.net
.inputs  a
.outputs  y z
dummy  r q s
.graph
cnt3impl.net
  p3 p6
  p2 p9 p4
  y p1 p9
  l p1 p9
  p5 p7
  p8 p7 p0
  p8
  a/1 p8
  p0 z
  q
  z s
  p3 q y y/1
  p4 y y/1
  p5 s
  p6 s a a/1
  q
  p8 r
  r
envcnt3sp.net
  a
  r0
  y a
  z r0
  y/1 r0
  r0 a/1
.marking {r0 p4 p5 p6 p7 p9}
.end
```

To reduce the above net, we issue the command:
```
petrify -hide .dummy cnt3res.net -o cnt3res.rednet
```
The outcome is:

```
.model cnt3res.rednet
.inputs  a
.outputs  y z
.graph
  y y/1 p1
  a p0
  a/1
  a/1 z
  y/1 y p1
  p0 y y/1
  p1 a
.marking { <z,y/1> p1 }
.end
```

The above net is evidently isomorphic to the original specification net. To clarify this point further, we issue the command:
```
write_sg cnt3res.rednet -o cnt3res.sg
```
The resulting state graph, shown below, clearly coincides with our initial specification state graph.

```
.model cnt3res.sg
.inputs  a
.outputs  y z
.state graph # 6 states
```
One easily verifies that the above implementation contains no dummy cycles. Thus, we have indeed proven that the given implementation net is a realization of the given specification net.

8: Up-Down Counters
===================
8.1 UD-Counters without Outputs
==================================
We first consider up-down transition counters without outputs ("udcnt"). For example, udcnt3 may be specified by the following state graph.

udcnt3[a,b].sg:= (s0 a s1 a s2 a s3 b s2 b s1 b s0)

Here, 'a' denotes an up- or down-transition of the incrementing input A, and 'b' indicates a transition (edge, level change) of the decrementing input B.

udcntk[a,b].sg for other values of k is defined similarly.

8.1.1 CADP Representation
----------------------------
Using CADP udcnt3[a,b].sg may be specified by the following file udcnt3.aut:

************
des (0,6,4 )
(0,A,1)
(1,A,2)
(2,A,3)
(3,B,2)
(2,B,1)
(1,B,0)
************
8.1.2 PETRIFY Representation
=============================
We first use a concise, non-safe net representation, consisting of two places p0,p1 each with a capacity of 3. Initially, place p0 contains 3 tokens.

*******************
.model threecnt.net
.inputs a b
.outputs
.graph
p0 a
a p1
p1 b
b p0
.capacity p0=3 p1=3
.marking {p0=3}
.end
*******************

The following command converts the above net into a safe one.

petrify threecnt.net -o cnt3.net

The outcome is shown below.

*******************
.model cnt3.net
.inputs a b
.outputs
.graph
p0 a
a p1
p1 b
b p0
.capacity p0=3 p1=3
.marking {p0=3}
.end
*******************
.graph
a p0
b p1
1 b/1
1 p0
b/2 a/2
a/2 p1
b a/1
p1 a b/2
marking { <b/2,a/2> }

We can also convert the net threecnt.net into a state graph by means of the following command:
write_sg threecnt.net -o cnt3net.sg
We get as expected:

**********************
model cnt3net.sg
inputs a b
state graph # 4 states
s2 a s3
s2 b s1
a s2
s1 b s0
s0 a s1
s3 b s2
marking {s0}
end
**********************

8.2 UD-Counters with Outputs
============================
Adding outputs to the preceding counters, enables us to identify the various states of the counter, by adding suitable output logic.
We define:
udcount3.sp[a,b,y,z]:= *[a,y]||*[y,z]||*[z,b]
udcount3.imp[a,b,y,z]:= icel[z,a,y]||icel[b,y,z]

Proposition 8.1
===============
udcount3.imp == udcount3.sp
where '==' denotes 'observation equivalence'.

Proposition 8.2
===============
udcount3.sp\{y,z\} == udcnt3

GADP-Representation
=====================
udcount3.sp may be represented in LOTOS by the following file udcount3sp.lotos:

******************************************************************************
specification UDC3[a,b,y,z]: noexit behaviour
UDC3[a,b,y,z]
where
process UDC3[a,b,y,z]:noexit:=
  (CY2[a,y] || CY2[z,b])||[y,z]|| CY2[y,z]
endproc
process CY2[a,b]:noexit:=
  a;b;CY2[a,b]
endproc
endspec
******************************************************************************

The following LOTOS-program udcount3imp.lotos describes the above
implementation udcount3.imp.

 specification udcount3imp[A,B,Y,Z]:noexit behaviour
     udcount3imp[A,B,Y,Z]
 where
 process udcount3imp[A,B,Y,Z]:noexit:=
     ICEL[Z,A,Y] | [Y,Z] | ICEL[B,Y,Z]
 endproc
 process ICEL[A,B,Z]:noexit:=
     B;Z;CEL[A,B,Z]
 endproc
 process CEL[A,B,Z]:noexit:=
     A;B;Z;CEL[A,B,Z]
     []
     B;A;Z;CEL[A,B,Z]
 endproc
endspec

We may use CADP to convert the above two LOTOS-files into the corresponding
*aut files and then use ALDEBARAN to confirm that the two *.aut files
are indeed obs.equivalent.

PETRIFY Representation
--------------------

The specification of udcount3 is given by the following net:

***************
.model udcount3sp.net
.inputs a b
.outputs y z
.graph
  *[a,y]
  y a
  *
  *[y,z]
  z y
  *
  *[z,b]
  b z
  .marking {<y,a> <z,y> <b,z>}
.end

**********************

The implementation net is shown:

**********************
.model udcount3imp.net
.inputs a b
.outputs y z
.graph
  #icel[z,a,y]
  z y
  a y
  y a z
  #icel[b,y,z]
  b z
  z b
  .marking {<z,y> <y,a> <b,z>}
.end

**********************

The two nets are evidently strongly equivalent.
8.3 UD-Counters k=4
===================
The above considerations are easily extended to values k>3.
For k=4 we get:

\[
\begin{align*}
udcount4.sp &= *[a,x] || *[x,y] || *[y,z] || *[z,b] \\
udcount4.imp &= icel[y,a,x] || icel[z,x,y] || icel[b,y,z]
\end{align*}
\]

Proposition 8.3
===============
udcount4.imp == udcount4.sp

Proposition 8.4
===============
\(\text{udcount4.sp}\{x,y,z}\) == udcnt4
where udcnt4 is the obvious extension of udcnt3.

9: Pipeline Controllers
=======================
In this section we consider the control part of asynchronous pipelines, serving as FIFO (First-In First-Out) queues. In particular, we draw your attention to the well-known Turing-award paper [Sut89]. A pipeline latch control unit has IN-connections RIN?, AIN! and OUT-connections ROUT!, AOUT?. ('?' denotes input, '!' denotes output). The IN-connections (also known as LEFT- or PUT-connections) control the data input from the preceding cell, and the OUT-connections (also: RIGHT- or GET-connections) control the data output to the following cell. The above connections refer to bidirectional transitions (edges) and not to levels ("two-phase protocol").

The IN-connections always alternate, and so do the OUT-connections. Following [Sut89] we postulate the alternation of AIN! and ROUT!.

9.1 Specification of LCU
========================
The above LCU may be specified by

\[
\begin{align*}
LCUsp[RIN,AIN,ROUT,AOUT] &= (*[RIN,AIN] || *[ROUT,AOUT]) || *[AIN,ROUT] \\
&= udcount3.sp[RIN,AOUT,AIN,ROUT]
\end{align*}
\]

9.2 Implementation of LCU
========================= Below is a possible implementation of LCU (see [Sut89]):

\[
\begin{align*}
LCUimp[RIN,AIN,ROUT,AOUT] &= icel[ROUT,RIN,AIN] || icel[AOUT,AIN,ROUT] \\
&= udcount3.imp[RIN,AOUT,AIN,ROUT]
\end{align*}
\]

In view of Proposition 8.1, we evidently have

Proposition 9.1
===============
LCUimp == LCUsp

10: Pipeline Controls - Alternative Approach
=============================================
An alternative approach to pipeline controls was developed by J.Ebergen (see [VERDECT]). Here we refer to the IN-connections as PUT-connections, and rename OUT-connections as GET-connections.

10.1 ONE-PIPE
==============
The ONE-PIPE defined below replaces the above LCU.

\[
\begin{align*}
\text{ONE-PIPE}[rput?,aput!,rget!,aget?]:= (*[rput?;put;aput!] || *[put;rget!;aget?])\{\text{put}\}
\end{align*}
\]

Proposition 10.1
================ ONE-PIPE\{aput,rget\} == udcnt2[rput,aget]
where udcnt2.sg := (s0 a s1 a s2 b s1 b s0)

10.2 THREE-PIPE

Below is a specification of a 3-PIPE controller, and its implementation using ONE-PIPE modules.

THREE_PIPE specification

THREE_PIPE.sp[rput?,aput!,rget!,aget?] :=
[rput?,put1,aput!] || udcnt2[put1,trans2] || *[trans2,rget!,aget?]
{put1,trans2}

THREE_PIPE implementation

THREE_PIPE.imp[rput?,aput!,rget!,aget?] :=
(ONE_PIPE[rput?,aput!,req1!,ack1?]
||
ONE_PIPE[req1?,ack1!,rget!,aget?])
\{req1,ack1}

Proposition 10.2

THREE_PIPE.imp |= THREE_PIPE.sp

Proposition 10.3

THREE_PIPE.imp\{aput!,rget!} |= udcnt3[rput?,aget?]

11: k-TOGGLES

In this section we use LOTOS/CADP to verify decompositions of k-TOGGLES, for k > 2. For an extensive discussion of TOGGLE decompositions, see [EDIS].

11.1 k = power of 2

If this is the case, a k-TOGGLE may be realized by a tree of TOGGLEs. To illustrate, let k=4. A 4-TOGGLE is specified by the following LOTOS-file.

************************************************
specification TOG4sp[A,W,X,Y,Z]:noexit behaviour
TOG4sp[A,W,X,Y,Z]
where
  process TOG4sp[A,W,X,Y,Z]:noexit:=
  A;W;A;X;A;Y;A;Z;TOG4sp[A,W,X,Y,Z]
endproc
endspec
************************************************

A 4-TOGGLE can be implemented by the following tree-structure, consisting of three TOGGLEs.

************************************************
specification TOG4imp[A,W,X,Y,Z]: noexit behaviour
TOG4imp[A,W,X,Y,Z]
where
  process TOG4imp[A,W,X,Y,Z]:noexit:=
  hide R,S in
endproc
  process TOG[A,Y,Z]:noexit:=
  A;Y;A;Z;TOG[A,Y,Z]
endproc
endspec
************************************************
Let $\text{TOG4ver}:=\text{TOG4sp} \parallel \text{TOG4imp}$. We get that $\text{TOG4ver}$ and $\text{TOG4sp}$ coincide. From this result one easily obtains $\text{TOG4imp} \models \text{TOG4sp}$.

Below we show the LOTOS-file $\text{TOG4ver}.\text{lotos}$.

```
****************************************************
specification \text{TOG4ver}[A,W,X,Y,Z]:noexit behaviour
T4v[A,W,X,Y,Z]
where
  process T4v[A,W,X,Y,Z]:noexit:=
    \text{TOG4sp}[A,W,X,Y,Z] \parallel \text{TOG4imp}[A,W,X,Y,Z]
  endproc
  process \text{TOG4sp}[A,W,X,Y,Z]:noexit:=
    A;W;A;X;A;Y;A;Z;\text{TOG4sp}[A,W,X,Y,Z]
  endproc
  process \text{TOG4imp}[A,W,X,Y,Z]:noexit:=
    \text{hide } R,S \text{ in}
    \text{TOG}[A,R,S] \mid [R,S] \mid (\text{TOG}[R,W,Y] \parallel \text{TOG}[S,X,Z])
  endproc
  process \text{TOG}[A,Y,Z]:noexit:=
    A;Y;A;Z;\text{TOG}[A,Y,Z]
  endproc
endspec
****************************************************
```

The corresponding file $\text{TOG4ver}.\text{omin}$ is given below.

```
*************
des (0, 8, 8)
  (0, A, 2)
  (0, A, 4)
  (1, W, 6)
  (1, X, 1)
  (1, Y, 7)
  (1, Z, 0)
  (2, A, 3)
  (2, A, 5)
*************
```

Thus, $\text{TOG4ver}$ and $\text{TOG4sp}$ are indeed strongly equivalent.

11.2 Deriving a $(k-1)$-TOGGLE from a $k$-TOGGLE
============================================
This can be achieved by the inclusion of a XOR-gate, as illustrated in the following implementation of a 3-TOGGLE.

The specification of a 3-TOGGLE is given below.

```
**********************************************
specification \text{TOG3sp}[A,W,X,Y]:noexit behaviour
\text{TOG3sp}[A,W,X,Y]
where
  process \text{TOG3sp}[A,W,X,Y,Z]:noexit:=
    A;W;A;X;A;Y;\text{TOG3sp}[A,W,X,Y]
  endproc
endspec
*********************************************
```

The implementation of a 3-TOGGLE, consisting of a 4-TOGGLE and a XOR-gate, is shown below.

```
***************************************************************************
specification \text{TOG3imp}[A,W,X,Y]: noexit behaviour
\text{TOG3imp}[A,W,X,Y]
where
  process \text{TOG3imp}[A,W,X,Y]:noexit:=
```
hide R,S,B,Z in
endproc
process TOG[A,Y,Z]:noexit:=
A;Y;A;Z;TOG[A,Y,Z]
endproc
process XOR[A,B,Z]:noexit:=
A;Z;XOR[A,B,Z]| [ ] B;Z;XOR[A,B,Z]
endproc
endspec

The easily verifies that indeed TOG3imp|= TOG3sp.

11.3 k-TOGGLE Trees

If each output of an m-TOGGLE is connected to an n-TOGGLE, we evidently get a k-TOGGLE, where k=m*n.

References


[VERDECT] see [EDIS]