Paper: [Yoe2001], Feb.2001

Title of Paper: Examples of LOTOS−Based Verification of Asynchronous Circuits

Author: Michael Yoeli, Prof. Emeritus
Dept. Computer Science
Technion, Haifa 32000, Israel
e−mail: myoeli@csa.technion.ac.il

Abstract: This paper illustrates the application of LOTOS/CADP to the verification of modular asynchronous circuits

1: Introduction

In this Report we illustrate the application of the high−level specification language LOTOS and its associated toolbox CADP to the verification of asyn−chronous circuits.
In [YG2001] we formulate the concept of realization (i.e., an implementation realizes a specification) using both automata theory as well as LOTOS/CADP and establish the relationship between the two approaches. There we also provide a brief introduction to Basic LOTOS (Control−oriented LOTOS; no data) using CADP.
Here we assume familiarity with Basic LOTOS and CADP.

2: Definition of Realization

Let IMPL and SPEC denote (LOTOS−)processes, representing edge−based descrip−tions of the implementation and the specification of an asynchronous circuit. We assume that the two processes share the same alphabet (i.e., set of obser−vable 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 con−dition can be verified by preferable ad−hoc methods.

3: Informal Motivation

Cond1 ensures that IMPL is at least as powerful as SPEC. Any behaviour speci−fied 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: 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
caesar −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, using *.exp and −exp2aut (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.

5: Module Descriptions
======================
We are concerned with the verification of modular asynchronous circuits.
Here we present a list of the modules (in LOTOS−style) we are interested
in. Related representations, and information about their decompositions,
can be found in [EDIS].
Note that we use bidirectional−edge based descriptions. We use A,B,C,D to
denote inputs, and X,Y,Z to denote outputs.

XORk Gates (=k−MERGE), k>1.
==========================
XOR2[A,B,Z]=A;Z;XOR2[A,B,Z] [] B;Z;XOR2[A,B,Z]
XORk, k>2, is defined similarly.

CELk Gates (=k−JOIN), k>1
=========================
CEL2[A,B,Z]=A;B;Z;CEL2[A,B,Z] [] B;A;Z;CEL2[A,B,Z]
CEL3[A,B,C,Z]=(A;exit ||| B;exit ||| C;exit)>> Z;CEL3[A,B,C,Z]
CELk, k>3, is defined similarly.
iCEL[A,B,Z]=B;Z;CEL2[A,B,Z]
CEL=CEL2

kTOGGLE, k>1
============
2TOGGLE=TOGGLE
TOGGLE[A,Y,Z]=A;Y;A;Z;TOGGLE[A,Y,Z]
3TOGGLE[A,X,Y,Z]=A;X;A;Y;A;Z;3TOGGLE[A,X,Y,Z]

6: Introductory Verification Examples
=====================================
The following two verification examples will illustrate some of
the concepts introduced in Section 4.

Example X1
==========
Let cy3[A,B,Z]= A;B;Z;cy3[A,B,Z].
We want to prove: IMPL |= SPEC, where IMPL=CEL[A,B,Z] and SPEC=cy3[A,B,Z].
Verifying Cond1:
We use a combination of methods (1) and (2).

File SIMPL.lotos
================
```
specification SIMPL[A,B,Z]: noexit behaviour
    SIMPL[A,B,Z]
where
process SIMPL[A,B,Z]:noexit:=
  cy3[A,B,Z] || CEL[A,B,Z]
endproc
process cy3[A,B,Z]:noexit:=
   A;B;Z;cy3[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
```

From the above lotos−file we derive the SIMPL.aut file (see Section 4).
The file SPEC.aut can be derived directly from SPEC=cy3[A,B,Z]:

File SPEC.aut
==========
```
des (0,3,3)
(0,A,1)
(1,B,2)
(2,Z,0)
```

We now issue the command: aldebaran −oequ SIMPL.aut SPEC.aut
and get: TRUE .

Verifying Cond2:
From the CEL.lotos file (cf. the above process CEL) we derive
the file CEL.aut.
We then issue the command: aldebaran −live CEL.aut
and get: no livelock.

Verifying Cond3:
In the CEL−part (only!) of SIMPL.lotos we replace Z by i;Z. We call the new
file iSIMPL.lotos. Then we get iSIMPL.aut. To check for deadlock, we issue
the command: aldebaran −dead iSIMPL.aut and get:
no deadlock states.

Example X2
==========
Let SPEC=cy3.aut and IMP=cy3i.aut
where
File cy3.aut
==========
```
des (0,3,3)
(0,A,1)
(1,B,2)
(2,Z,0)
```

File cy3i.aut
==========
```
des (0,4,3)
(0,A,1)
(1,B,2)
(2,Z,0)
(1,Z,0)
```

Proceeding as before, we get cy3.aut || cy3i.aut obs.equiv. cy3.aut.
Similarly, Cond2 is immediately verified.

To check Cond3, we generate iIMPL.aut = icy3i.aut.

File icy3i.aut
==========
```
des (0,6,5)
(0,A,1)
(1,B,2)
(2,i,4)
(1,i,3)
(3,Z,0)
(4,Z,0)
```

Next, we generate file iX2.aut = cy3.aut || icy3i.aut

File iX2.aut

```
============
des (0, 5, 5)
(1,i,2)
(3,i,4)
(0,A,3)
(3,B,1)
(2,Z,0)
```

Checking this file for deadlocks, we find that state 4 is indeed a deadlock. Thus Cond3 is not satisfied.

7: Module Decompositions

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 XORk,k>2 and CELk,k>3 can be specified and verified similarly.

8: 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".

8.1 Specification

A modulo−N (transition) counter can be specified as follows:
Inputs: A
Output: Y,Z
Behaviour: cnt.N[a,y,z]:= ((a;y;)**(N−1)a;z)*
where w**N denotes the sequential repetition of w, N times.
For example, cnt.3[a,y,z]=(a;y;a;y;a;z)*
Note that the module TOGGLE coincides with the modulo−2 transition counter.

8.2 − 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 8.2.1

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

Here, \{p,q,x} indicates the "hiding" of p,q,x, i.e., their replacement by 'i'.

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

Proposition 8.2.2

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

Furthermore, the following decomposition rule is rather evident.

Proposition 8.2.3

Let N=N1xN2, where N1>2, N2>2.
Then cnt.N[a,y,z] = (cnt.N1[a,y,q]|[q]|cnt.N2[q,y,z])\{q}

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.

8.3 − Verification of Modulo−3 Transition Counter

The specification of this counter 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:=
             hide R,Q,S in
       XOR[A,S,R] |[R,S]| (toggle[R,Y,Q] |[Q]| 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;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 0 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.

9: Pipeline Controllers

In this section we consider the control part of asynchronous pipe−lines, 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 [CT97] has IN−connections RIN?, AIN! and OUT−connections ROUT!,AOUT? (’?’ denotes input, ’!’ denotes out−put). The IN−connections (also known as LEFT− or PUT−connections) control the data input from the preceding cell, and the OUT−connect−ions (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 assume that the two sides are connected by the alternation of AIN! and ROUT!. In summary we get the following speci−fication of the latch control unit (LCU).

File LCUspec.lotos
==================
```
specification LCUspec[RIN,AIN,ROUT,AOUT]: noexit behaviour
           LCUspec[RIN,AIN,ROUT,AOUT]
where
   process LCUspec[RIN,AIN,ROUT,AOUT]:noexit:=
       (CY2[RIN,AIN]|||CY2[ROUT,AOUT]) |[AIN,ROUT]| CY2[AIN,ROUT]
   endproc
   process CY2[A,B]:noexit:=
     A;B;CY2[A,B]
   endproc
endspec
```
==

Here CY2[A,B] evidently means that A and B alternate.

The corresponding implementation (see [Sut89]) is represented by:

```
File LCUimp.lotos
=================
specification LCUimp[RIN,AOUT,AIN,ROUT]:noexit behaviour
     LCUimp[RIN,AOUT,AIN,ROUT]
where
   process LCUimp[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
```

It is easy to prove that LCUimp |= LCUspec.

9.1 Up-Down Counters

There exists an interesting connection between LCUspec and an up-down counter with the range 0-3. To see this connection, we reformulate LCUspec, hiding the signals AIN and ROUT. The relevant LOTOS-specification is shown in the file below.

```
File udc4.lotos
===============
specification UDC4[RIN,AOUT]: noexit behaviour
          UDC4[RIN,AOUT]
where
   process UDC4[RIN,AOUT]:noexit:=
            hide AIN,ROUT in
       (CY2[RIN,AIN]|||CY2[ROUT,AOUT]) |[AIN,ROUT]| CY2[AIN,ROUT]
   endproc
   process CY2[A,B]:noexit:=
     A;B;CY2[A,B]
   endproc
endspec
```

To relate the above lotos-file to an up-down counter, we define the following aut-file.

```
File udcnt4.aut
===============
des (0,6,4)
(0,RIN,1)
(1,AOUT,0)
(1,RIN,2)
(2,AOUT,1)
(2,RIN,3)
(3,AOUT,2)
```

The above file evidently defines an up-down counter with range 0-3. Let udc4.aut be the LTS corresponding to udc4.lotos, defined above. Using aldebaran, we may establish obs. equivalence between udcnt.aut and udc4.aut.

The preceding considerations can be extended to a cascade of LCUs. An interesting alternative approach to micropipeline control circuits

and the related up-down counters is presented in [VERD]/examples/ micropipelines. Most of this material is easily reformulated using LOTOS/CADP. The relevant propositions can then be proven within our framework.

between such control circuits and UP-DOWN counters is also elaborated. Most of this material is easily formulated using LOTOS/CADP, and the relevant propositions can then be proven within our framework.

10: References

[EDIS] http://edis.win.tue.nl/

[EP92] J.C.Ebergen and A.M.G.Peters, Modulo-N Counters: Design and Analysis of Delay-Insensitive Circuits, in: J.Staunstrup and R.Sharp (Editors), Designing Correct Circuits, Elsevier Science Publ., 1992, pp. 27-46.

[Sut89] I.E.Sutherland, Micropipelines, (Turing Lecture), Comm. ACM, 32(6), pp.720-738, 1989.

[YG2001] M.Yoeli and A.Ginzburg, LOTOS-based Verification of Asynchronous Circuits, Technical Report , Dept. of Computer Science, Technion, Haifa. http://www.cs.technion.ac.il/Reports/