Paper: [YG2001], March 2001; replaces [YG98].

Title of Paper: LOTOS/CADP-based verification of asynchronous circuits

Contact Author: Michael Yoeli, Prof. Emeritus

Dept. Computer Science

Technion, Haifa 32000, Israel

e-mail: myoeli@csa.technion.ac.il

Second Author: Abraham Ginzburg, Prof. Emeritus

Dept. Computer Science, Technion

Abstract:

The paper introduces a precise formulation of the relationship between specification and realization of asynchronous circuits.

It demonstrates the suitability of the high-level specification language LOTOS, and its related toolbox CADP to the automated verification of modular, asynchronous circuits.

Chapter 0 - Introduction

This paper deals with the formal verification of asynchronous circuits. In particular, it demonstrates the applicability of the high-level specification language LOTOS to the verification tasks in question. Furthermore, we use the LOTOS-oriented toolbox CADP for the purpose of automating our verification approach.

An extensive literature is presently available dealing with asynchronous circuits. See, e.g., [BS94], which provides an extensive list of references, as well as the web-site [Async].

Asynchronous (clock-free) systems have important advantages over (globally clocked) synchronous systems, particularly in view of the present trend toward high-speed and high-density technologies. For more details, see [Sut89],[Async],[BS94],[KG97].

A large amount of the presently available literature deals with (correct- by-construction) synthesis. The literature on formal verification is rather limited. We mention [Yoe87],[Dill89],[McM95],[RCP95],[Roi97].

We find that using LOTOS and CADP provides us with tools which compete favorably with other published approaches to the verification problem under discussion, particularly with respect to modular, observationally non-deterministic, asynchronous circuits.

For information on LOTOS see Appendix A. Information on CADP is available at the web-site Appendix A/[inria]/.

LOTOS and CADP have proven powerful tools for the high-level specification of communication protocols, and their verification. In [FL93] and [TS94] the applicability of LOTOS to the formal specification of hardware systems was demonstrated. The application of LOTOS and CADP to the formal verification of asynchronous hardware, presented in this paper, appears to be new.

In Chapter 1 we discuss our basic approach to the modelling of asynchronous circuits, their specification and realization. In particular, we introduce a mathematically precise formalism, establishing the relationship between specification and realization.

In Chapter 2 we present a short introduction to Basic LOTOS (dealing with control concepts only) and the LOTOS-oriented toolbox CADP.

In Chapter 3 we introduce LOTOS-based descriptions of basic asynchronous components (modules) and of modular networks.

In Chapter 4 we discuss the LOTOS-based automated verification of asynchronous, modular networks.

Chapter 1 - Asynchronous Circuits : Basic Concepts of Realization

1.1 Introduction

In this chapter we discuss our approach to the modelling of asynchronous circuits, their specifications, and their realizations. In general, we describe circuit realizations, as well as their specifications, by means of their "dynamic behavior".

1.2 Dynamic (Edge-based) Behavior

The conventional approach to the behavioral description of digital circuits is based on observing the values of inputs and outputs at suitably chosen, discrete points in time. In a synchronous circuit, for example, the frequency of these observations is in accordance with the clock rate. In contrast to this "static" or "level-based" approach to behavioral descriptions, the "dynamic" approach applied in this paper, is based on observing "external events" of the circuit, i.e. changes of the value of an input or output. A dynamic description of the behavior of a digital system consists of listing all admissible sequences of external events, subject to restrictions imposed on the environment. We are interested in the order in time, in which the events occur, and not in the actual time intervals involved. The dynamic viewpoint is particularly applicable, and is indeed widely used, with respect to asynchronous circuits. This approach to the behavior of asynchronous circuits is quite similar to the way, processes/agents are described, e.g., in CSP, CCS, and LOTOS.

Note that we use the same symbol to denote a rising transition as well as a falling transition at a given node.

1.2.1 An Illustrative Example

To illustrate our approach to dynamic behavior descriptions, we consider a simple combinational gate, namely an Inverter. Assume such an inverter has a binary-level input A, and a binary-level output Z. Notice that this inverter is stable (i.e., no output change will occur, as long as the input does not change) iff $A \neq Z$.

We impose the following restriction on the behavior of the environment: an input change may occur only if the inverter is stable. This restriction is to ensure the proper, predictable behavior of the gate, independently of its propagation delay. If we were to admit an input change, while the gate is still unstable, the outcome would be unpredictable (an output pulse may or may not occur, depending on the actual timing conditions). Since we are interested in delay-insensitive behavior, such an environment restriction is well justified.

Thus we describe the dynamic behavior of this inverter by a finite automaton Inv.a with state set {q0,q1}. Here state q0 of Inv.a represents the two stable states of the inverter, and state q1 of Inv.a represents the two unstable states (i.e., A=Z). The alphabet of Inv.a is {a,z}. E.g., the symbol ’a’ denotes a level change at input port A, namely both an up-transition (from level 0 to level 1) as well as a down-transition. The state transitions of Inv.a are (q0,a,q1) and (q1,z,q0). The initial state is q0.

1.3 Circuit Transition Systems (CTS)

In this section we model the behavior of an asynchronous circuit by means of a modified version of an LTS (Labeled Transition System), called CTS, to be defined next. A CTS also includes information as to the restrictions imposed on the environment. These restrictions are intended to ensure the delay-insensitivity of the circuit, as illustrated in Section 1.2.1.

Definition 1.3.1

A *Circuit Transition System* CT is a 6-tuple

CT = (inCT, outCT, intCT, QCT, fCT, q0CT).

inCT is a finite set of *input symbols*. An input symbol represents an input node of the relevant circuit, as well as a change of logic level (i.e., from 0 to 1, or from 1 to 0) at the corresponding node. Similarly, outCT and intCT are finite sets of *output symbols* and *internal symbols*, respectively. Output symbols and internal symbols represent corresponding circuit nodes, as well as their level changes. The sets inCT, outCT, and intCT are mutually disjoint.

We set inCT$\cup$outCT$\cup$intCT = aCT.

QCT is a finite set of *states*.

fCT, the *next-state function*, is a partial function from QCT$\times$aCT into QCT.

q0CT$\in$QCT is the *initial state* of CT.

We postulate that all states in QCT are reachable from the initial state.

Let CT be a circuit transition system, as defined above. CT may be viewed as an acceptor automaton with aCT as its alphabet. We consider every state of QCT to be an accepting state, and denote by L(CT) the corresponding language.

Let w$\in$L(CT). We denote by w\intCT the restriction of w to the alphabet inCT$\cup$outCT, and set L(CT)\intCT = {w\intCT | w$\in$L(CT)}.

We denote by CT\ the automaton obtained from CT by replacing its internal symbols by ε. Evidently, CT\ may be non-deterministic, although CT has been defined as a deterministic system.

1.4 Specifications and their Realizations

Definition 1.4.1

A Specification S is a CTS, where intCT= ∅. We set

S = (inS, outS, QS, fS, q0S).

The language L(S) is defined similarly to the above definition of L(CT).

The following definition is intended to formalize the intuitive concept "CT is a realization of S".

Definition 1.4.2

Let S = (inS, outS, QS, fS, q0S) be a specification and CT a circuit transition system,

CT = (inCT, outCT, intCT, QCT, fCT, q0CT).

We say that CT is a *realization* of S (notation: CT |= S), iff the following conditions are satisfied.

Note: We use ';' to denote concatenation.

(1) inCT = inS

(2) outCT = outS

(3) L(S) ⊆ L(CT)\intCT

(4) assume w ∈ L(S), z ∈ outS, w';z ∈ L(CT) and w'\intCT = w.

Then w;z ∈ L(S) .

(5) assume w1;w2 ∈ L(S), and there exists w' ∈ L(CT), such that w'\ intCT = w1.

Then there exists w", such that w"\intCT = w2 and w';w" ∈ L(CT).

(6) let w ∈ L(S), w' ∈ L(CT), and w' \ intCT = w.

Then there exists a positive integer k such that for any word w" ∈ (intCT)*, w';w" $\in$ L(CT) implies length(w")<k.

Clearly, if CT\ is deterministic, Requirement (5) is implied by Requirement (3), i.e., Requirement (5) becomes redundant.

Informally, the above Definition 1.4.2 may be motivated as follows.
(a) the input and output symbols of CT are assumed to coincide with inS and outS, respectively (Requirements (1) and (2)).
(b) we expect the realization to be capable of producing for any signal sequence specified by L(S) a sequence which, after deletion of its internal symbols, will be equal to the above signal sequence. CT may even be capable of performing more than required by S (Requirement (3), cf.[Dill89]).
(c) the realization may not produce any "undesirable" outputs (Requirement (4)).
(d) Requirement (5) is rather evident; as mentioned above, this Requirement is redundant, if CT\ is deterministic.
On the other hand, one easily verifies that Requirement (5) implies Requirement (3) (see also Section 3.1.1).
(e) a word of the realization which reduces to a word in L(S), cannot be continued by an infinite sequence of internal signals (Requirement (6)).

1.5 Related Work

For related discussions of the topic of this Chapter, see [BS94]/Chapter 11, as well as [Dill89].

Chapter 2 - Introduction to Basic LOTOS using CADP

2.1 Introduction

In this chapter we provide a short introduction to Basic LOTOS in conjunction with the LOTOS-oriented toolbox CADP. LOTOS is a high-level specification language, mainly designed to specify communication protocols. Basic LOTOS deals with control aspects only, disregarding any data-processing.

Further tutorials on LOTOS and an overview on CADP are listed in Appendix A. We recommend that you first study this chapter, but also get copies of the documentation on CAESAR and ALDEBARAN, listed in App. A.

If you are familiar with other process algebras, e.g., CSP and CCS, our relevant comments will help. Otherwise just skip such comments.

2.2 Basic Concepts

Similar to most process algebras, LOTOS is concerned with two elementary concepts, namely "process" and "action" or "event". Usually, a process may perform a finite number of actions. After performing such an action, the process changes into another process. We write P[a>Q, to state that ’a’ is one of the actions, process P is capable of performing. After performing ’a’, P changes into Q.

2.3 Processes Formalized

The following is a formal way of introducing processes.

Definition 2.3.1

A *Directed Labelled Graph* G consists of a finite set V of *nodes* (or vertices), a finite set A of *labels*, and a finite set E of *(labelled) edges*, where $E \subseteq V \times A \times V$.

Definition 2.3.2

A *Process* P is a pair (G,v), where G is a directed labelled graph, and v is a node of G. With respect to P the elements of A are referred to as *actions* or *events*.

LOTOS uses ’i’ do denote an event which is not observable by an outside observer. Event ’i’ in LOTOS corresponds to ’’ in CCS, and to ’’ in automata theory.
We also define a particular process ’exit’, which is unable to perform any action. Another such process is ’stop’, indicating deadlock.

2.4 Observation Equivalence

Let ’a’ be any observable action of some process. We call an extension of ’a’ any sequence in i*ai*, where i* is any finite sequence of zero or more i’s.

Definition 2.4

Two processes are called *observation equivalent* iff any extension of an observable action, applicable to one process is matched by some extension of the same observable action, applicable to the other process, and the outcomes are again observation equivalent.

2.5 Some Basic LOTOS Operations

2.5.1 Prefix

The process P=a;Q is capable of performing only one action, namely ’a’, and then becomes Q. The operator ’;’ is called the *prefix* operator. Similar operators are available in CSP (->) and CCS (.).

Consider,e.g., the process Q=a;(b;(c;exit)), which may be written as Q=a;b;c;exit. The process Q will perform actions a,b, and c, sequentially, and will then terminate. Note that "exit" indicates a "successful" termination, e.g., no deadlock.

2.5.2 Choice

The choice operator on processes is denoted as []. P[]Q is the process which may behave either as P or as Q. It is similar to ,e.g., the choice operator in CSP.

Formally P[]Q defines the process R, specified as follows.

(1) If P[x>P', then P[]Q [x> P'

(2) If Q[y>Q', then P[]Q [y> Q'.

The following is an example of a nondeterministic process:

(a;b;exit)[](a;c;exit)

If 'a' is applied to this process, the process reached is uncertain: it may be either (b;exit) or (c;exit).

On the other hand, the following process is deterministic:

(a;b;exit)[](c;d;exit)

However, the process (a;exit)[](b;exit) is not observation- equivalent to (i;a;exit)[](b;exit). The distinction is the same as in CCS.

2.6 Using CADP

The above concepts may be demonstrated by means of CADP.

2.6.1 Labelled Transition Systems (LTSs)

LTSs play an important role within the CADP toolbox. An LTS may be viewed as a direct representation of a pair (G,v), where G=(V,A,E) is a directed labelled graph, and v is a node of G. In LTS-terminology the elements of V are referred to as *states*, v is the initial state, E is the set of *transitions*, and A is the set of *labels*.

2.6.2 ALDEBARAN

The ALDEBARAN part of CADP allows the efficient handling of LTSs. In the basic

ALDEBARAN format of LTSs each state is represented by a natural number. The initial state is always 0. The representation appears as file*.aut. The first line of this file is des(0,# of transitions, # of states). The other lines represent transitions in the evident way.

2.6.3 Examples

We assume you now have copies of the ALDEBARAN and CAESAR manuals (see Appendix A).
We will use the command

```
 'caesar -aldebaran filename.lotos'
```

to convert a LOTOS-program (filename.lotos) into an LTS representation (filename.aut).

By applying the command

```
            'aldebaran -omin filename.aut'
```

a reduced, observation-equivalent LTS is obtained. We shall denote this reduced version by filename.omin. We start with a few, simple examples, to illustrate the concepts introduced in Section 2.5. We recommend that you try similar examples on your own!

Example X.1 - prefix1

The file prefix1.lotos:

```
specification prefix1[a,b,c]:exit behaviour
             a;i;b;i;i;c;exit
endspec
```

Note that ’exit’ in the above specification heading indicates a terminating process, whereas ’noexit’ indicates that the process is not terminating.

The file prefix1.aut:

```
des (0, 7, 8)
(0, A, 1)
(1, i, 2)
(2, B, 3)
(3, i, 4)
(4, i, 5)
(5, C, 6)
(6, exit, 7)
```

Note that ALDEBARAN uses the label ’exit’ to indicate a transition to an exit state.

The file prefix1.omin:

```
des (0, 4, 5)
(0,A,2)
(2,B,3)
(3,C,4)
(4,exit,1)
```

The following two examples demonstrate the use of the operator [].

Example X.2 - choice1

File choice1.lotos:

```
specification choice1[a,b]:exit behaviour
        (a;exit)[](b;exit)
```

```
endspec
```

Henceforth we omit the *.aut files, which you can easily get on your own.

The file choice1.omin:

```
des (0, 3, 3)
(0,A,1)
(0,B,1)
(1,exit,2)
```

Example X.3 - choice2

File choice2.lotos

```
specification choice2[a,b]:exit behaviour
        (i;a;exit)[](b;exit)
endspec
```

The file choice2.omin

```
des (0, 4, 4)
(0,i,3)
(0,B,2)
(2,exit,1)
(3,A,2)
```

2.7 Additional LOTOS Constructs

2.7.1 Recursion

The following example illustrates the application of recursion to the construction of a simple two-state cyclic automaton.

Example X.4 - cycle

File cycle.lotos:

```
specification CYCLE [A,B] : noexit behaviour
    Q[A,B]
where
    process Q[A,B]: noexit:=
       A;B;Q[A,B]
    endproc
endspec
```

file cycle.aut = file cycle.omin

```
des (0, 2, 2)
(0, A, 1)
(1, B, 0)
```

2.7.2 Parallel Operators

LOTOS provides various facilities to represent concurrent processes. If P and Q are processes, P|||Q represents their parallel execution, without any synchronization, i.e., completely independent execution. On the other hand, P||Q represents their parallel execution, with complete synchronization on all observable(!) actions (but not on 'i'!). Note that the '|||' operator is known as the "interleaving" operator, and the '||' operator as the "full synchronization" operator.

If A1,...,An are some common, observable actions of P and Q, then

P|[A1,...,An]|Q

represents the parallel execution of P and Q, provided they synchronize on the listed actions A1,...,An. This operator is referred to as "selective composition" or "selective parallel" operator.

The above concepts will be illustrated by means of a few, simple examples.

2.7.3 Parallel Operators - Examples

The following example illustrates the use of the interleaving operator (|||).

Example X.5 - interleaving

The file interleaving.lotos:

```
specification interleaving[A,B,C,D]:exit behaviour
 (A;B;exit)|||(C;D;exit)
endspec
```

We leave it to you to get and analyze the corrsponding *.aut and *.omin files!

The following example deals with the selective composition operator.

Example X.6 - selcomp

File selcomp.lotos:

```
specification selcomp[A,B,C,D]:exit behaviour
        (A;B;C;exit) |[A]| (D;A;C;exit)
endspec
```

File selcomp.omin:

```
des (0,8,8)
```

```
(0,D,2)
(1,C,6)
(2,A,4)
(4,C,5)
(4,B,7)
(5,B,1)
(6,exit,3)
(7,C,1)
```

The full synchronization operator (||) is illustrated in the following examples.

Example X.7 - fullsyn

File fullsyn.lotos:

```
specification fullsyn[a,b,c]:exit behaviour
        (a;i;b;i;i;c;exit) || (a;i;i;b;c;exit)
endspec
```

File fullsyn.omin:

```
des (0,4,5)
(0,A,2)
(2,B,3)
(3,C,4)
(4,exit,1)
```

The following example involves a deadlock.

Example X.8 - fullsyn1

File fullsyn1.lotos:

```
specification fullsyn[a,b,c]:exit behaviour
      (a;b;exit) || ((a;c;exit) [](a;b;exit))
endspec
```

We again leave it to you to obtain and study the relevant *.aut and *.omin files.

Chapter 3 - LOTOS-based Behavior Descriptions

3.1 Introduction

In Chapter 1 we introduced the basic concepts of realization, with respect to asynchronous circuits.

In this and the following chapter we represent a modular circuit and its specification by a suitable, LOTOS-based behaviour description.

3.2 The Concept of Module

We assume asynchronous circuits, composed of basic components ("modules"), which are more complex than simple gates. The correct behavior of such modules is to be ensured by proper circuit design (at the transistor level), rather than by logic design.

Such a module may be in a stable condition (i.e., no output will change, as long as no input occurs), or otherwise be in an unstable condition. We assume that each module adheres to the following "fundamental mode restriction":

(*) In a stable condition an input may be applied; in an unstable condition no input may be applied, but an output must occur.

3.2.1 Representation of Components

In this section we discuss methods of representing the dynamic behaviour of basic components (modules).

For such basic components we establish three methods of representation:

(a) By means of labeled transition systems (LTS);

(b) By using a simplified LOTOS-based notation;

(c) By means of proper LOTOS specifications (process descriptions).

3.2.2 LTS-Representation

Our LTS-representation of (the dynamic behaviour of) components is summarized in the following definition.

Definition 3.2.1

A component comp.a is defined by a 5-tuple (in,out,Q,f,q0), where

(a) in and out are finite sets of *input ports* and *output ports*, respectively.

(b) Q is a finite set of states.

(c) f is a partial function from $Q\times$ (in $\cup$ out) into Q.

(d) $q0 \in Q$ is the initial state.

Notice that this definition is a special case of Definition 1.3.1, where intCT=$\emptyset$.

3.2.3 A Simplified LOTOS-Based Notation

In this notation we use the major LOTOS operators, namely [], the three parallel operators, and >> (P>>Q denotes the process P followed by Q, where P is process terminating successfully). For convenience, we introduce the following modified notation:

(1) $ = exit

(2) If P is a process which terminates successfully, we denote by P* the process P* = P>>P* .

(3) If B is a behaviour expression and H is a set of events, participating in B, then

B\H is the behaviour obtained from B by hiding the events in H.

3.2.4 LOTOS Specifications

For the purpose of computer-aided verification, we use proper LOTOS specifications. The corresponding file will be denoted comp.lotos. By applying the CADP command ’caesar -aldebaran comp.lotos’, a file comp.aut will be generated, providing an LTS version of the original specification. The command ’aldebaran -omin comp.aut > comp.omin’ provides a reduced version (comp.omin), observation-equivalent to comp.aut. CADP also provides a facility to check observation equivalence between two comp.out graphs.

We illustrate the above representation methods by means of the following example.

3.3 The Two-Input XOR-Gate

We are concerned with a XOR-gate with binary inputs A and B, and binary output Z. The gate is *stable* iff (A xor B) =Z.

In the descriptions below we deal with the dynamic behavior of the gate.

3.3.1 LTS Representation

Definition 3.3.1

The LTS XOR.a is defined as the 5-tuple

XOR.a = (inX,outX,QX,fX,q0X), where

inX={a,b}, outX={z}, QX={0,1}, q0X=0, and

fX = {(0,a,1),(0,b,1),(1,z,0)}.

Note that the labels a,b,z denote ports ("gates"), as well as events occurring at these ports.

Note further, that this component adheres to the "fundamental- mode" restriction.

3.3.2 The Simplified LOTOS-based Notation

Let XOR.b be the representation in this notation. Then XOR.b = [(a;$[]b;$)>>z;$]* .

Alternatively, XOR.b can be represented recursively as follows:

XOR.b = a;z;XOR.b [] b;z;XOR.b

3.3.3 The LOTOS Specification

The file XOR.lotos

```
specification XOR[A,B,Z]: noexit behaviour
   QXOR[A,B,Z]
where
   process QXOR[A,B,Z]: noexit :=
     A;Z;QXOR[A,B,Z]
     []
     B;Z;QXOR[A,B,Z]
   endproc
endspec
```

After applying the relevant CADP commands (see above) we get:

file XOR.omin

```
des (0,3,2)
(0,B,1)
(0,A,1)
(1,Z,0)
```

Note that for the above example, the LTS representation coincides with fX of the comp.a representation.

3.4 Modular Networks

A modular network is obtained by suitably interconnecting a finite number of modules. The modules of such a network must adhere to the following restriction.

If p is an input of any module, there may be at most one module with p as output.

In such a case the output p of this particular module is connected to every input p occurring in any other module. Such an interconnection p may be declared either as an output port of the network, or as an internal port.

Such an interconnection is modelled in LOTOS by means of the partial synchronization operator |[p]|. In case the interconnection is declared as internal port of the network, the LOTOS hiding operator is applied.

Note that in this paper we do not consider forks as independent modules, but as "built-in parts" of the following modules.

Chapter 4 - LOTOS-Based Verification

4.1 The Basic Approach

In this chapter we relate the concepts of Chapter 1 to the LOTOS-oriented discussions of Chapter 3. We will frequently refer to the details of the definition of CT|=S (Definition 1.4.2).

4.1.1 The Major Requirement

Let CT be a circuit transition system. We denote by CT\int the LTS obtained from CT by replacing all its internal symbols by ’i’. Let CT.lot denote a LOTOS-process, which is observation-equivalent to CT\int.

Given a specification S, we denote by S.lot a LOTOS-process, observation-equivalent to S.

We now formulate a set of conditions which ensure that CT|=S holds. They can be checked using CADP.
The checking of (1) and (2) of the Definition of CT|=S is evidently trivial.

Condition C1

S.lot || CT.lot is observation-equivalent to S.lot.

Proposition 4.1

Condition C1 implies (3) and (5) of Definition 1.4.2.

Proposition 4.2

(5) of Definition 1.4.2 implies Condition C1.

Thus, Requirement (3) is a consequence of Requirement (5).

For a proof of the above two propositions see Appendix B.

4.1.2 The No-Livelock Condition

An automaton, such as CT.aut, is said to be livelock-free, iff it does not contain a cycle, all edges of which belong to intCT. In this case, CT.aut evidently satisfies (6) of the Definition 1.4.2. The above no-livelock condition is satisfied in many practical examples.

Consequently, we introduce

Condition C2

CT.aut is livelock-free.

Using CADP, we can verify this requirement by means of the following command:

```
aldebaran -live CT.aut
```

4.1.3 The No-Undesirable-Output Condition

Here we refer to (4) of the Definition 1.4.2.

This condition is satisfied if the following requirement is met:

Condition C3

Let iCT.lot be the process obtained from CT.lot by replacing each output, say z, by i;z. Then S.lot||iCT.lot is deadlock-free.

However, frequently ad-hoc approaches are preferable.

4.2 Verification Example VE.X4

In this section we deal with the following verification example:

VE.X4: X4.IMP |= X4.SP

Here, X4.SP specifies a 4-input XOR-gate (XOR4), and X4.IMP represents a possible implementation, containing three 2-input XOR-gates.

In the following we use the simplified LOTOS-based notation introduced in Chapter 3.

The Specification X4.SP

X4.SP= a;z;X4.SP [] b;z;X4.SP [] c;z;X4.SP [] d;z;X4.SP

Note that inXOR4={a,b,c,d} and outXOR4={z}.

The Implementation X4.IMP

X4.IMP=

((XOR[a,b,r]|||XOR[c,d,s]) |[r,s]| XOR[r,s,z])\{r,s}

4.2.1 Verification of Condition C1

Using the formal LOTOS-representations of X4.SP and X4.IMP, which are easily derived from the above simplified notation, and applying the toolbox CADP, one verifies that Condition C1 is indeed satisfied.

4.2.2 The No-Livelock Requirement

Using CADP, one verifies that Condition C2 is satisfied by X4.IMP.aut, the LTS-representation of X4.IMP.

4.2.3 The No-Undesirable Output Condition

Applying CADP, Condition C3 is easily verified.

This concludes the proof of VE.X4.

4.3 Conclusion

In additional publications we demonstrate the applicability of the above verification approach to various, more complex examples.

References

[Async] The Asynchronous Logic Home Page, at
http://www.cs.man.ac.uk/amulet/async/index.html

[BS94] J.A.Brzozowski and C-J.H.Seger, Asynchronous Circuits,
Springer-Verlag, 1994.

[Dill89] D.L.Dill, Trace Theory for Automatic Hierarchical Verification
of Speed-Independent Circuits,
Ph.D. Thesis, CMU 1988; also: MIT Press 1989.

[KG97] R.Kol and R.Ginosar, Future Processors will be Asynchronous,
TR EE#1099, Dept.E.E., Technion, Israel, 1997.

[McM95] K.L.McMillan, A Technique of State Space Search Based on Unfolding,
Formal Methods in System Design, pp.45-65, 1995.

[RCP95] O.Roig, J.Cortadella, and E.Pastor,
Verification of asynchronous circuits by BDD-based model checking
of Petri nets. LNCS #935, pp.374-391, 1995.

[Roi97] O.Roig, Formal Verification and Testing of Asynch. Circuits,
Doct.Informatica Thesis, Universitat Politecnica de Catalunya,
Barcelona, 1997 .

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

[Yoe87] M.Yoeli, Specification and Verification of Asynchronous Circuits,
Using Marked Graphs, in: Concurrency and Nets,

Advances in Petri Nets, Springer-Verlag, pp.605-622, 1987.

[YG98] M.Yoeli and A.Ginzburg, LOTOS-based Verification of Asynchronous Circuits, Tech.Rep., CS-Dept, Technion, 1998.

A Guide to LOTOS Literature **Appendix A**

A. WWW/FTP Sites

[utwente]/	http://wwwtios.cs.utwente.nl/lotos/
[uottawa1]/	http://www.csi.uottawa.ca/˜lotos/
[uottawa2]/	ftp://lotos.csi.uottawa.ca/pub/Lotos/
[upm]/	http://www.dit.upm.es/˜lotos/
[stir1]/	http://www.cs.stir.ac.uk/˜kjt//research/
[stir2]/	ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/
[inria]/	http://www.inrialpes.fr/vasy/cadp.html

B. Tutorials

[KJT96] TechRep Kenneth J. Turner. The Formal Specification Language LOTOS: A Course For Users. Department of Computing Science and Mathematics, University of Stirling, Scotland, 1996.
in: [stir2]/lotos_users.ps.gz

[LFH92] Logrippo, L., Faci, M., and Haj-Hussein, M. "An Introduction to LOTOS: Learning by Examples" . In Computer Networks and ISDN Systems, 1992. Available in FrameMaker or PostScript formats.
in: [uottawa2]/Papers/index.html

C. Applications to Hardware

[FL93] Faci, M. and Logrippo, L. "Specifying Hardware Systems In LOTOS" , revised version of the paper that appeared in the proceedings CHDL'93 April 26-26, 1993. Available in FrameMaker or PostScript formats.

in: [uottawa2]/Papers/index.html

[TS94] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Umit Uyar, editors, Proc. Formal Description Techniques VI, pages 71-86. North-Holland, Amsterdam, Netherlands, 1994. in:[stir1]/publications.html

Proof of Propositions 4.1 and 4.2 Appendix B

In this Appendix we prove Propositions 4.1 and 4.2, formulated in Chapter 4.

We start by recalling the basic concepts involved.

Let S be a specification, and CT a circuit transition system. Below we repeat item (5) of the Definition of CT|=S (Definition 1.4.2).

(5) Assume w1;w2∈L(S), w'∈L(CT), where w'\intCT = w1.

Then there exists w", such that w"\intCT = w2, and w';w"∈L(CT).

We denote by S.lot a LOTOS-process, observation-equivalent to S. CT\int is obtained from CT by replacing all internal symbols by 'i'.

CT.lot is a LOTOS-process, obs.-equiv. to CT\int.

Condition C1

S.lot || CT.lot is obs.-equiv. to S.lot.

In the following, S.lot, CT.lot, and S.lot||CT.lot refer to the corresponding LOTOS processes, as well as to the corresponding LTSs (automata).

Proposition 4.1

Condition C1 => Requirements (3) and (5) of Definition 1.4.2

Proof

We first prove C1 => (5).

w1;w2∈L(S) => w1;w2∈L(S.lot), (in view of the observational equivalence between S and S.lot).

Now consider the LTS (automaton) representing S.lot||CT.lot.

Its states may be viewed as ordered pairs (a,b), where a is a state of S.lot, b is a state of CT.lot, and there exists a word w∈L(S.lot) and a word w'∈L(CT.lot), such that w'\i=w (w'\i is obtained from w' by omitting all instances of 'i' in w'), and q0(S.lot)[w>a , q0(CT.lot)[w'>b.

Condition C1 implies that there exists a relation R satisfying the requirements of the relevant observation equivalence. Clearly,

q0(S.lot||CT.lot)=(q0(S.lot),q0(CT.lot)).

Notice that (q0(S.lot),q0(CT.lot)) is a state of S.lot||CT.lot, in view of the above considerations, with w=w'=λ.

Furthermore, q0(S.lot||CT.lot) R q0(S.lot).

Let now w1 and w' be as in (5). Also, let q0(S.lot)[w1>a and q0(CT.lot)[w'>b .

Then, (a,b) is a state of S.lot||CT.lot. In view of the relevant observation equivalence, we have (a,b)Ra. Furthermore, assume a[w2>c. In view of the above observation equivalence, there exists a state (a',b') in S.lot||CT.lot, such that a[w2>a'in S.lot, b[w">b' in CT.lot, and w"\i=w2. It follows that w';w" is applicable to q0(CT.lot), i.e., w';w"∈L(CT.lot).

Thus, (5) is established.

A similar reasoning shows that C1 => (3).

Proposition 4.2

(5) => Condition C1.

Proof

In order to establish Condition C1, it is necessary and sufficient to establish a relation R between the states of S.lot||CT.lot and the states of S.lot, which satisfies the requirements of observational equivalence. This relation R may be defined as follows.

(1) (q0(S.lot),q0(CT.lot))=q0(S.lot||CT.lot) R q0(S.lot)

(2) For any w1∈L(S.lot) and w'∈L(CT.lot), with w'\i=w1, let

q0(S.lot)[w1>a and q0(CT.lot)[w'>b

Then we set (a,b)Ra.

Evidently, domain R contains all the states of S.lot||CT.lot and range R contains all the states of S.lot.

Let now (a,b)Ra and assume that w2 is applicable to a. Then w1;w2∈L(S.lot), and qo(S.lot)[w1;w2>c, where a[w2>c.

(5) implies that there exists w", such that w"\i=w2 and w';w"∈L(CT.lot). Hence,

q0(CT.lot)[w';w">d, where b[w">d.

But (w';w")\i = w1;w2 . By definition of R, we have (c,d)Rc.

It follows that R is indeed an observational equivalence, i.e., Condition C1 holds.

Acknowledgement

The authors are very grateful to Dr.Hubert Garavel and Dr.Laurent Mounier for their valuable assistance, leading to improved versions of this and related papers.