Hardware Implementation of Algebraic Specifications

Giuseppe Mangioni¹

Dipartimento di Ingegneria Informatica e delle Telecomunicazioni
Facoltà di Ingegneria – Università di Catania
Viale A. Doria 6 – 95100 Catania (ITALY)
Giuseppe.Mangioni@diit.unict.it

Abstract. The growing complexity of hardware devices being developed and increasing time-to-market constraints have scaled up the risk of designing bug-affected devices, thus increasing interest in formal design techniques. This approach considerably improves early error detection, giving good guarantees on the effectiveness of the devices produced. Design requires techniques to be provided to synthesise the device, while preserving its features. The proposed technique defines a hardware translation of LOTOS specifications into RTL. To preserve LOTOS synchronisation semantics a handshake protocol is defined.

Keywords: High Level Synthesis, Formal Methods, Design Methodology.

1 Introduction

Hardware devices are becoming pervasive in modern technological society. They are present in most of tools, from household appliances to aircraft controllers. The correct behaviour of devices is often of critical importance. Nonetheless, the lifetime of devices is short; in fact, market requirements call for fast updating of devices in order to respond to new incoming technologies. All these issues have a great impact on the design approach; in fact, techniques providing faster prototyping, testing and validation become the key to success.

The adoption of a formal approach to hardware design can be considered a valid solution and this philosophy is commonly accepted by a number of researchers [1]. Using formal techniques, the design flow (commonly made up of specification, synthesis and testing phases) is rearranged in order to anticipate design validation. This is very interesting because the use of formal methods

¹ Corresponding Author. Email: Giuseppe.Mangioni@diit.unict.it.
relieves the designer of heavy testing phase.

In order to pursue this aim we propose a hardware design methodology based on the use of a Formal Description Technique (FDT) called Language Of Temporal Ordering Specification (LOTOS [2]). Our approach starts with the LOTOS specification of the device, than it is validated (by means of model checking or theorem proving) and finally, the specification of the device is synthesised into a Register Transfer Level Language (RTL). In this paper we discuss the synthesis algorithm used to translate a LOTOS specification into RTL.

In literature there exist other examples of synthesis of LOTOS specifications into hardware devices. In [3] a technique to transform a LOTOS specification into a VHDL specification is presented, but it is restricted to two-way synchronisation. In [4] a LOTOS synthesis algorithm that uses an EFSM model of a LOTOS specification is presented. This approach implements multi-way synchronisation among processes using a rendezvous table that models the synchronisation mechanism.

The novelty of our approach is the use of syntax-direct synthesis algorithm, that is for each LOTOS operator a corresponding hardware implementation is defined. This method has the following advantages:
- no intermediate HDL description is required;
- reduces the number of design paths and thus criticism of the design;
- allows us to evaluate the size and performance of the final device directly using a back-annotated high-level LOTOS specification.

In the following we discuss the synthesis algorithm showing the soundness of our approach through a case study. It has been chosen in the area of communication protocols because this is the field in which the designers are more confident with these techniques.

In section 2 the hardware design methodology is presented. Section 3 introduces the main features of LOTOS. Section 4 discusses the implementation model of the synchronisation. Section 5 introduces some definitions that will be used in the next sections. Section 7 describes the synthesis algorithm and in Section 8 a case study is shown.

2 Design Methodology

The hardware design methodology presented in this paper is based on three steps:
1. LOTOS specification of the system
2. Validation of the LOTOS specification
3. Synthesis of the specification in the hardware device

This design path allows early error detection, cheaper device testing, giving error-free results. The first step in the methodology is the specification of the system, which, as we said above, is carried out using LOTOS. The purpose of this phase is to express the requirements of the system given by the client in LOTOS. The specification can be built up with several refinement stages. The purpose of refinement is to give the specification in a form that is easily and effectively im-
implementable. During the specification phase, several styles are usually adopted. In [5] the problem of the style of specification to be adopted in the different phases of the design is discussed; the problem of transformation from one style of specification to another is also dealt with. Finally, it is important to note that LOTOS have been successfully used for hardware specification as illustrated in [6][7].

The second stage consists of verifying that the specification actually meets the system requirements. How to perform validation strongly depends on the formal technique adopted and what is to be validated. During the whole validation phase we need to verify that the behaviour described corresponds to what we want to specify (model validation). The model that we use for specification allows us to deal with the problem differently from traditional simulation methods. In fact, verification through simulation is carried out by using test patterns as inputs of the system, and by verifying that the outputs are the expected ones. This method, however, only allows partial verification of the behaviour of the system, because correctness is assured only for the verified test patterns. The use of a formal model gives the opportunity to exploit the mathematical basis to carry out more complete verifications. For example, we can verify the so-called safety properties, that is, the system, whatever the inputs might be, never ends up in undesired states. We can also verify the liveness properties, that is, a given desired configuration is adopted by the system. Besides, the combination of these two properties allows us to verify even very complex situations.

Tools for formal verifications can be divided into two categories: theorem proving-based tools and finite automata-based tools. The former are tools that assist (semi–automatically) the designer during mathematical demonstrations, meaning that they assure the correct use of the basic theorems of the model(see [8]). The second category of tools, which exploit the theory of automata, permit the verification process to be automatised, even if this approach is not often feasible due to problems related to state explosion (see [9]). To perform formal verification of LOTOS specifications, we use the CADP toolbox [10], which provides an integrated set of functions ranging from interactive simulation to exhaustive, model-based verification methods, and currently includes sophisticated approaches to deal with large case studies.

The third stage consists of implementation of the specification. The main purpose of the synthesis algorithm presented in the following sections is to obtain a device through direct synthesis into an RTL description of the specification processed during the previous phases of the methodology. This allows us to avoid intermediate translations into representations which have no formal basis and which therefore might not assure consistency of the final result with the specification input.

3 LOTOS

The basic idea behind LOTOS is that the behaviour of a system can be described by observing from the outside the temporal order in which events occur. In practice, the system is seen as a black-box which interacts with the environment by means of events, the occurrence of which is described by LOTOS behaviour expressions. The language has two components: the first is the
description of the behaviour of processes and their interaction, and is mainly based on the CCS[11] and CSP[12]; the second is the description of the data structure and expressions, and is based on ACT ONE[13].

In LOTOS distributed systems are described in terms of processes; the system as a whole is represented as a process, but it may consist of a hierarchy of processes which interact with each other and the environment. LOTOS models a process by its interaction with the environment. The atomic forms of interaction take the name of events.

The definition of a process in LOTOS is:

```
process <proc-id> <par-list> :=
  <behaviour-expression>
endproc
```

where:
- `<proc-id>` is the name to be assigned to the process;
- `<par-list>` is the list of events with which the process can interact with the environment;
- `<behaviour-expression>s` are the LOTOS expressions which define the behaviour of the process.

The recursive occurrence of a process-identifier in a behaviour expression makes it possible to define infinite behaviour (both auto-and mutual recursion are possible). A completely inactive process, i.e. one which cannot execute any event, is represented by stop. In the following paragraphs we will describe the basic operators by which it is possible to describe any system.

The action prefix represents the basic synchronisation. This operator produces a new behaviour expression from an existing one, prefixing it with the name of an event. If B is a behaviour expression and a is the name of an event, the expression a; B indicates that the process containing it first takes part in the event a and then behaves as indicated by the expression B.

The choice operator models the nondeterministic behaviour which takes place when two (or more) events are available. If B1 and B2 are two behaviour expressions, then B1 [] B2 denotes a process which can behave both as B1 and as B2. Choice between the two forms of behaviour is made by the environment.

To simplify the description of the system being specified, a number of derived operators are present in LOTOS. They can easily be rewritten in terms of the basic one but the specification become longer and difficult to understand. Some of the derived operators are described below.

Arbitrary interleaving represents the independent composition of two processes, B1 and B2 and is indicated as B1 ||| B2. If the two processes have some event in common, B1 ||| B2 indicates their capacity to synchronise with the environment but not with each other.

The parallel operator is indicated as B1 || B2 and it means that the two processes have to synchronise with each other in all events. B1 || B2 can take part in an event if and only if both B1 and B2 can participate. General parallel composition is a general way of expressing the parallel composition of several events and is denoted with the expression B1 [[a1,...,an]] B2.

The sequential composition of two processes, B1 and B2, is indicated as B1>>B2 and models the fact that when the execution of B1 terminates successfully B2 is executed. To mark successful termination, there is a special LOTOS process called exit. The introduction of types makes it
possible to describe structured events. They consist of a gate name which identifies the point of interaction and a finite list of attributes. Two types of attribute are possible: value declaration and variable declaration. Table 1 presents the permitted interactions.

Table 1. Type of interactions among processes

<table>
<thead>
<tr>
<th>process B1</th>
<th>process B2</th>
<th>sync condition</th>
<th>int. sort</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>g!E₁</td>
<td>g!E₂</td>
<td>value(E₁)=value(E₂)</td>
<td>matching</td>
<td>sync</td>
</tr>
<tr>
<td>g!E</td>
<td>g?x:t</td>
<td>value(E)∈domain(t)</td>
<td>passing</td>
<td>after sync X = value(E)</td>
</tr>
<tr>
<td>g?x:t</td>
<td>g?y:u</td>
<td>t = u</td>
<td>generation</td>
<td>after sync X = y = v , ∀ v ∈ domain(t)</td>
</tr>
</tbody>
</table>

4 Implementation model of synchronization

In this section the implementation model of the LOTOS synchronisation is illustrated. As discussed in Section 3, synchronisation between LOTOS processes takes place through interaction points called gate. Three kinds of synchronisation can be modelled in LOTOS:

1. synchronisation with no data exchange;
2. value declaration g!x, in which a gate is ready to synchronise itself offering a data item x. In this case we refer to the gate as a transmitter;
3. variable declaration g?y, in which a gate is ready to synchronise itself accepting data that will be stored in y. In this case we refer to the gate as a receiver;

In this paper we discuss the synchronisation model under the hypothesis that no more than one transmitter (with the same name) is involved in each rendezvous, whereas, several receivers can be involved. Synchronisation in which only receivers or more than one transmitter are involved lie beyond the scope of this paper; in fact, in describing hardware devices these cases are not significant.

The LOTOS choice operator describes the nondeterminism of the occurrence of a set of events. The synthesis algorithm solves the nondeterministic choice with a policy discussed in 7.1.

Each gate involved in an event is modelled with four signals (READY, ACK, ENABLE, DATA ENABLE) used to implement the multi-way rendezvous. These signals take on a different sense according to whether the gate is a transmitter or a receiver.

Receiver

Let \( a_i \), a gate, refer to:

1. \([a_i]\), an output signal through gate \( a_i \) (READY signal). When it is raised it denotes to the transmitter the receiver’s availability to take part in the synchronisation.
2. \([a_i]_a\), an input signal through gate \( a_i \) (ACK signal). When it is raised it denotes the transmitter’s availability to take part in the synchronisation.
3. \([a_i]_e\), an output signal (ENABLE signal). It is meaningful only in the case of nondeterminism in a choice. This signal is used to select one of several events offered.
4. \([a_i]\_d\), an output signal through \( a_i \) (DATA ENABLE SIGNAL). It enables data storing in \( x_i \).
Transmitter

Let \( a \), a gate, refer to:

- \( [a_i] \), an input signal through gate \( a_i \) (READY signal). It notifies to \( a_i \) the availability of all receivers to take part in the synchronisation. It is the result of a logic AND among all the READY signals emitted by the receivers.
- \( [a_i] \), an output signal through gate \( a_i \) (ACK signal). It is raised to notify to the receivers the transmitter availability to take part in the synchronisation. This signal is emitted after reception of the READY signal from all receivers involved.
- \( [a_i] \), an input signal through gate \( a_i \) (ENABLE signal). It is meaningful only in the case of nondeterministic choice. It is the result of a logic AND among all the ENABLE signals emitted by the receivers.
- \( [a_i] \), an output signal through \( a_i \) (DATA ENABLE SIGNAL). It enables reading of data \( x_i \).

Fig. 1. Interactions among transmitter and receivers

Now we will discuss our approach in the more general case, that is, one-to-many synchronisation, in which one transmitter and \( n \) receivers are involved. The synchronisation is solved in four phases (see figure 1):

1. all the receivers set the READY signal high to notify their availability to take part in the synchronisation. The transmitter READY signal is equal to the logic AND of all the receiver READY signals. When the READY signal is raised, the transmitter will be certain that all the receivers are ready to synchronise.

2. In this phase the transmitter raises the ACK signal to notify the receiver of the start of the rendezvous.

3. In this phase each receiver raises his ENABLE signal. The ENABLE signal of the transmitter, on the other hand, is the logic AND of all the receiver ENABLE signals. As said above, this signal is used to manage the selection among all the possible synchronisation event.

4. Each gate (both transmitters and receivers) that takes part in synchronisation raises its DATA ENABLE signal to allow the data transfer.
5 Notation

Each LOTOS behaviour-expression can be expressed in terms of a set of choice-expressions. For this reason, the synthesis algorithm is based on synthesis of a generic choice expression. In this section, we introduce some useful notation.

Given a choice expression:
- let \( N \) be the total number of branches;
- let \( M \) be the branches guarded by an event;
- let \( K \) be the transmitter (!) branches and \( M - K \) the receivers (?);
- let \( A \) be the finite set of atomic actions;
- let \( P \) be the finite set of LOTOS process names (including stop and exit).

**Definition**

1. We denote \( TRANS \) as the ordered set (with signature \( K \)) of 4-tuples \((a_i, B_i, x_i, \text{cond}_i)\) with \( i = 1..K \mid \text{cond}_i \rightarrow a_i ! x_i ; B_i \) is a choice transmitter branch, where:
   - \( a_i \in A \) denotes the event of the \( i-th \) branch;
   - \( B_i \) denotes the behaviour expression following action \( a_i \);
   - \( ! x_i \) denotes the value offering by variable \( x_i \) through gate \( a_i \);
   - \( \text{cond}_i \) denotes the Boolean guard of the \( i-th \) branch;

   In the following we denote with \( T_i \) each 4-tuple \( \epsilon TRANS \).

2. We denote \( RECVS \) as the ordered set (with signature \( M - K \)) of 5-tuples \((a_i, B_i, x_i, \text{type}_i, \text{cond}_i)\) with \( i = K + 1..M \mid \text{cond}_i \rightarrow a_i ? x_i : \text{type}_i ; B_i \) is a choice receiver branch
   where:
   - \( a_i \in A \) denotes the event of the \( i-th \) branch;
   - \( B_i \) denotes the behaviour expression following action \( a_i \);
   - \( ? x_i : \text{type}_i \) denotes a variable offering of the variable \( x_i \) of type \( \text{type}_i \);
   - \( \text{cond}_i \) denotes the Boolean guard of the \( i-th \) branch;

   Below we will denote each 5-tuple \( \epsilon RECVS \) with \( R \).

3. We denote \( PROCS \) as the ordered set (with signature \( N - M \)) of 2-tuples \((P_i, \text{cond}_i)\) with \( i = M + 1..N \mid \text{cond}_i \rightarrow P_i \) is the choice branch with process \( P_i \) instantiation, where:
   - \( \text{cond}_i \) denotes the Boolean guard of the \( i-th \) branch;
   - \( P_i \in P \) denotes the instantiation of a process.

   We will denote the 2-tuple \( \epsilon PROCS \) with \( PR_i \).

Using the above notation, a generic choice can be written as follows (where we use \( \Sigma \) instead of \([\text{}]):
Definition
- Given a 4-tuple \( T_i \in TRANS \), we define a function \( S^i_T \) which returns the \( i^{th} \) element of \( T_i \) with \( i = 1..4 \).
- Given a 5-tuple \( R_i \in RECVS \), we define a function \( S^i_R \) which returns the \( i^{th} \) element of \( R_i \) with \( i = 1..5 \).
- Given a 2-tuple \( PR_j \in PROCS \), we define a function \( S^i_{PR} \) which returns the \( i^{th} \) element of \( PR_j \) with \( i = 1..2 \).

Using the previous notation, each choice can be modelled using (1).

Value offering
A behaviour expression in which a gate \( a \) offers a value \( x \) and then behaves as process \( B \) (that is \( !a;B_x \)) is a sub-case of a generic choice and can be obtained by (1) with the following parameters:

\[ choice((a,B,x,true)),\{\},\{\} ) \]

Variable offering
A behaviour expression in which a gate \( a \) is able to accept a value \( x \) of type \( type \) and then behaves as process \( B \) (that is \( ?a;x:type;B \)) is a sub-case of a generic choice and can be obtained by (1) with the following parameters:

\[ choice(\{\},(a,B,x,type,true)),\{\} ) \]

Example
The following choice expression:

\[ \begin{align*}
[\text{guard1}] & \rightarrow a1!x1;B1 \\
[\text{guard2}] & \rightarrow a2?x2:type2;B2 \\
[\text{guard3}] & \rightarrow P1
\end{align*} \]

can be expressed by (1) with the following parameters:

\[ choice((a1,B1,x1,\text{guard1})),(a2,B2,x2,type2,\text{guard2}),(P1,\text{guard3})) \]
6 The target RTL language

The RTL language used throughout this paper is a language which can define the structure of a generic digital system. Any digital system is modelled using a functional block which receives information from the external environment by using signals and processes them, producing output signals (the response to the environment). Each functional block is implemented by the control unit and the processing unit. The first unit provides the signal to synchronise the operations performed by the second. The full system is based on a single clock which provides the synchronisation. The basic hypothesis is that the circuit must be stable before the clock cycle finishes. An RTL module is defined by the following sections:

- components: it contains the declaration of the components which make up the processing unit.
- control sequence: it defines the internal command sequence which must be emitted by the control unit.
- permanent assignment: it defines an operation which must be repeated every clock cycle.

The control sequence is made up of steps; each one is numbered and must be executed in a single clock unit. Each step is made up of one or more commands which are executed in parallel. All the commands belonging to a step are separated by ;. Therefore the control sequence has the following form:

\[ \begin{align*}
\text{i: } & \text{op1; op2; op3} \\
\text{j: } & \text{op4; op5}
\end{align*} \]

where \( i \) and \( j \) are the generic steps \( i \) and \( j \) and \( op_i \) are the commands.

The main constructs of the language are the assignment and the conditional. The first represents the transfer of a value between two registers. The right hand side of the operation can contain any Boolean operation. The two operators are represented as follows:

\[ \begin{align*}
\text{i: } & \text{targetRegister := sourceRegister} \\
\text{j: } & \text{targetRegister := sourceRegister1 and} \\
& \text{sourceRegister2 or ...} \\
\text{k: } & \text{if( c1; c2 ) then (op1; op2)} \\
\text{h: } & \text{if( c3; c4 ) goto (n; m)}
\end{align*} \]

To describe a direct connection between elements, the language allows us to describe the assignment of a value to a line; in this case the assignment is only valid for one clock cycle. It is described by the operator “=” and is also used to describe assignment to output lines.
7 Synthesis algorithm

7.1 Restrictions

In this subsection we present some of the hypotheses on which the synthesis process is based, and in particular which restrictions we impose in using LOTOS.

The basic element of LOTOS is the event, which consists of interaction between processes based on a rendez-vous mechanism. As shown in Table 1, three different types of interaction are present in LOTOS: value matching, value generation and value passing. The only one which is meaningful for our application is value passing because it has a correspondence with the physical reality of devices, thus it is the only one we have taken into account.

The instantiation of processes in LOTOS plays a fundamental role in the specification of systems; some situations which are syntactically correct cannot be used due to the static nature of hardware. The main limit imposed on the use of processes lies in the use of recursion and the form of the gate list. Mutual recursion must be avoided because it can generate a dynamic structure and therefore has no correspondence in hardware. Moreover, self-instantiation is only allowed if the gate list is not modified.

In the implementation of the choice operator we have to solve the nondeterminism typical of this operator because it cannot easily be implemented in either hardware or software. Moreover, in a choice all the guards related to a process instantiation must be mutually exclusive with respect to all other guards (we consider all branches without guards as if they were guarded by TRUE). More formally:

\[ \forall i, j, l \text{ where } i = 1..K, j = K + 1..M, l = M + 1..N \]
\[ (S_{PR}^2(PR_i) \neq S_{PR}^4(T_i) \text{ AND } S_{PR}^2(PR_i) \neq S_{R}^5(R_j)) \]

We are working on discarding some of the above limitations.

7.2 Translation of choice into RTL

In this section the synthesis of a generic LOTOS choice and some simple examples are presented. The proposed technique is valid under the restrictions discussed in sub-section 7.1.

As said in Section 3 a LOTOS specification is made up of several processes synchronising with each other and with the external environment. Moreover, a process can be expressed simply as a composition of generic choices. For these reasons, we focus on implementation of the generic choice and refer to all other processes (and the external environment) as the environment. The RTL module in figure 2 is the synthesis of a generic choice, parametrised with respect to the number and types of branches. The first instruction, labelled 1, is made up of two operations. The first
operation (equation 2) consists of raising the READY signal for all receivers present in the choice (phase 1 in figure 1).

\[
\bigcup_{i=K+1}^{M} [S^i_R]_r
\]  

(2)

This operation implements notification of receivers available to take part in a synchronisation.

\[
\text{choice}(\text{TRANS}, \text{RECVS}, \text{PROCS})_{\text{RTL}} =
\]

1: \(\bigcup_{i=K+1}^{M} [S^i_R]_r = 1;\)

\[
\text{if } (C_1; C_2; C_3; C_4) \text{ goto } (1; \bigcup_{i=1}^{K} 3i - 1; \bigcup_{i=K+1}^{M} (K + 2i); \bigcup_{i=M+1}^{N} (M + K + i + 1); )
\]

\[\vdots\]

for \(i = 1 \ldots K\)

\[\text{for } (3i-1): [S^i_T(T_i)]_a = 1\]

\[\text{for } (3i-1)+1: \text{if } \text{(not} [S^i_T(T_i)]_a, [S^i_T(T_i)]_a \text{) goto } (1; (3i - 1) + 2)\]

\[\text{for } (3i-1)+2: [S^i_T(T_i)]_r = S^3_T(T_i); \text{exit}(i)\]

\[\vdots\]

for \(i = (K + 1) \ldots M\)

\[\text{for } (K+2i): [S^i_R(R_i)]_r = 1\]

\[\text{for } (K+2i)+1: S^i_R(R_i) := [S^i_R(R_i)]_r; \text{exit}(i)\]

\[\vdots\]

for \(i = (M + 1) \ldots N\)

\[\text{for } (M+K+i+1): \text{exit}(i)\]

Fig. 2. RTL synthesis of the LOTOS choice operator

The second operation is a conditional goto on \(C_1, C_2, C_3, C_4, \ldots\), and is used to select which branch to activate. The term \(C_i\) is true iff the environment is not ready to take part in any synchronisation that this choice can achieve, and it is the negation of the logic or of three sub-terms \(C_i, C_i', C_i''\), i.e. not \((C_i' \text{ or } C_i'' \text{ or } C_i''')\) as shown in equation 3. While \(C_i\) is true the instruction 1 is re-executed.

\[
C'_i = \bigvee_{1}^{K} ([S^i_T(T_i)]_r \text{ and } S^3_T(T_i))
\]  

(3a)
\[ C_1^* = \bigvee_{i=K+1}^M ([S^1_R(R_i)]_a \text{ and } S^5_R(R_i)) \]  

\[ C_1''' = \bigvee_{i=M+1}^N (S^2_{PR}(PR_i)) \]  

In particular, the sub-term \( C_1' \) is true iff none of the transmitter branches of the generic choice can synchronise with any receiver belonging to the environment. The sub-term \( C_1'' \) is true iff none of the receiver branches of the generic choice can synchronise with any transmitters belonging to the environment. The sub-term \( C_1''' \) is true iff all the guards enabling process instantiation branches are false.

The term \( C_2 \) comprise as many sub-terms as there are transmitter branches (i.e. \( K \), see equation 4a). The i-th sub-term is true iff the i-th transmitter can synchronise with the environment (phase 1 in figure 1), in which case the related goto jumps to instruction \((3i-1)\). Each sub-term is a logic and between \( C_{2i} \) and \( C_{2i}' \).

\[ C_2 = \bigcup_{i=1}^K (C_{2i}' \text{ and } C_{2i}'') \]  

\( C_{2i}' \) is true when both a READY signal has been raised by the environment and the guard on the i-th branch (\( \text{cond}_i \)) is true. This means that the environment is ready to synchronise on the event offered by the i-th transmitter. Note that other choice branches may be ready to synchronise. In LOTOS selection of which event will take place is made in a non-deterministic fashion. To deal with non-determinism, we choose to select the branch with lowest index i. For this purpose we have introduced \( C_{2i}'' \).

\[ C_{2i}' = [S^1_T(T_i)]_a \text{ and } S^4_T(T_i) \]  

\[ C_{2i}'' = \Lambda \text{ not } ([S^1_T(T_j)]_a \text{ and } S^4_T(T_j)); \]  

The term \( C_3 \) comprises as many sub-terms as there are receiver branches (i.e. \( M - K \), see equation 5a). The i-th sub-term is true iff the i-th receiver can synchronise with the environment (phase 2 in figure 1), in which case the related goto jumps to instruction \((K+2i)\). Each sub-term is a logic and among \( C_{3i}' \), \( C_{3i}'' \) and \( C_{3i}''' \).

\[ C_3 = \bigcup_{i=K+1}^M (C_{3i}' \text{ and } C_{3i}'' \text{ and } C_{3i}''' ) \]  

\( C_{3i}' \) is true when both an ACK has been raised by the environment and the guard on the i-th branch (\( \text{cond}_i \)) is true. This means that environment is ready to synchronise on the event offered by the i-th receiver. Unlike transmitter branches (sub-term \( C_{2i}' \), equation 4b) synchronisation starts when the ACK is raised. \( C_{3i}''' \) assures that no transmitter has already been selected for synchronisation, whereas \( C_{3i}'' \) has the same meaning as \( C_{2i}'' \).
(5b)

\[ C'_{3i} = [S^1_R(R_j)]_a \text{ and } S^5_S(R_j) \]

\[ C^\sigma_3 = \bigwedge_{j=1}^{K} \text{ not}([S^1_R(T_j)]_a \text{ and } S^4_S(T_j)) \] (5c)

\[ C^\sigma_{3i} = \bigwedge_{j=K+1}^{j-i} \text{ not}([S^1_R(R_j)]_a \text{ and } S^5_S(R_j)) \] (5d)

The term \( C_4 \) comprises as many sub-terms as there are process instantiation branches, i.e. \( N - M \) (see equation 6a). The i-th sub-term is true iff the i-th process can be instantiated, in which case the related goto jumps to instruction \((M + K + i + 1)\). Each sub-term is a logic and among \( C'_{4i} \), \( C^\sigma_4 \) and \( C^\sigma_{4i} \).

\[ C_4 = \bigcup_{i=M+1}^{N} (C'_{4i} \text{ and } C^\sigma_4 \text{ and } C^\sigma_{4i}) \] (6a)

\( C'_{4i} \) is true when the guard on the i-th branch \( (cond_i) \) is true. Note that in this case no synchronisation takes place, so selection of the i-th process instantiation branch does not depend on the environment. \( C^\sigma_4 \) assures that neither transmitters or receivers have already been selected for synchronisation, whereas \( C^\sigma_{4i} \) has the same meaning as \( C^\sigma_{2i} \).

\[ C'_{4i} = ((S^2_P(R_i)) \] (6b)

\[ C^\sigma_4 = C^\sigma_3 \text{ and } \bigwedge_{j=K+1}^{M} \text{ not}([S^1_R(R_j)]_a \text{ and } S^5_S(R_j)) \] (6c)

\[ C^\sigma_{4i} = \bigwedge_{j=M+1}^{j-i} \text{ not}(S^2_P(R_i)) \] (6d)

In figure 2, the RTL instructions ranging from \((3i - 1)\) to \((3i - 1) + 2 \) with \( i = 1...K \) implement the i-th transmitter synchronisation protocol phases 2, 3 and 4, as described in Section 4. The instructions ranging from \((K + 2i)\) to \((K + 2i) + 1 \) with \( i = (K + 1)...M \) implement the i-th receiver synchronisation protocol phases 3 and 4. Finally the instructions \((M + K + i + 1)\) deal with process instantiation branches. In the following subsection the above formula is used to produce the RTL description of the simple examples presented in Section 5.

Value offering

\[ \text{choice} \{ \{a, B, x, true\} \}, \{ \} \} \] 

1: if (not(a_r and true); a_r and true) goto(1; 2)

2: \( a_d = 1 \)

3: if (not a_r; a_r) goto (1; 4)

4: \( a_v = x; \text{ exit}(1) \)

Variable offering
\[ \text{choice}(\{\}, \{(a, B, x, \text{type, true})\}, \{\})_{\text{RTL}} = \]

1: \( a_r = 1; \text{if (not (a and true); a and true)} \) goto(l, 2)
2: \( a_c = 1 \)
3: \( x := a_c; \text{exit(1)} \)

Example

\[ \text{choice}(\{(a_1, B_1, x_1, \text{guard1})\}, \{(a_2, B_2, x_2, \text{type2}, \text{guard2})\}, \{(P_1, \text{guard3})\})_{\text{RTL}} = \]

1: \( a_{1_0} = 1; \)
   \text{if (}
      \text{not((a_{1_0} and guard1) or (a_{2_0} and guard2) or ((guard3));}
      \text{(a_{1_0} and guard1);}
      \text{((a_{2_0} and guard2) and not (a_{1_0} and guard1));}
      \text{(guard3) and not (a_{2_0} and guard2) and not (a_{1_0} and guard1));}
   \text{) goto (l; 2; 5; 7) \)
2: \( a_{2_0} = 1 \)
3: \( \text{if (not a_{1_0}; a_{1_0}) goto (l; 4) \)
4: \( a_{1_0} = x_1; \text{exit(1)} \)
5: \( a_{2_0} = 1 \)
6: \( x_2 := a_{2_0}; \text{exit(2)} \)
7: \( \text{exit(3)} \)

7.3 Graphic Layout

As RTL is a representation of an actual device, it is possible to draw a netlist using proper blocks. Each block represents a choice operator, the only difference lying in the number of input/output terminals, which actually depends on the number of branches. Figure 3 shows the generic choice block; the signals are grouped as follows:

- **transmitter.** For each gate \((a_i)\) five signals are needed to implement the transmitter synchronisation model: \([a_i], [a_i], [a_i], [a_i], \text{cond}_i;\)
- **receiver.** For each gate \((a_i)\) five signals are needed: \([a_i], [a_i], [a_i], [a_i], \text{cond}_i;\)
- **processes/guards.** These groups collect all the signals needed to implement the synchronisation of a branch where a process instantiation is present. Only one signal is needed: \(\text{cond}_i;\)
- **control.** This group collects the signals needed to drive the block:
**IN.** When it is raised the choice starts; it is usually connected to the exit signal of another block;

**EXIT.** When it is raised the active choice has finished and the next is activated. They are equal to the number of branches;

**RST.** It resets the block;

**CLK.** The clock.

---

**8 Case Study - Firewire Protocol Synthesis**

P1394 (“High Performance Serial Bus”) [14] is the IEEE standard intended to be a pervasive means of hooking up modern digital devices. P1394 offers a standardised interface for assembling components for a given application. The architecture of P1394 is based on $n$ nodes connected through one high speed serial line (figure 4).

P1394 supports two different operating modes: asynchronous and isochronous.

In the asynchronous mode the transmitter node can send a message of arbitrary length to the receiver node after having gained control of the bus during the arbiter phase. An acknowledge message must be provided by the receiver to the transmitter to confirm reception of the message.

In the isochronous mode, the transmitter must transmit messages at a fixed rate. Moreover, no acknowledgement is required. This mode is used when a very high rate is required between two nodes, for instance to transmit videos from a camera to a computer.

In this section we illustrate direct synthesis of the LOTOS specification of the asynchronous mode. For this reason, we will briefly introduce this operating mode. A more accurate description can be found in [14].

The architecture of P1394 structures each node into three layers: the physical layer, the link layer and the transaction layer.
1. **transaction** is the upper layer (TRANS in figure 4). It implements three different services: *read*, allowing data to be read from another node; *write*, allowing data to be written on another node; *lock*, requiring another node to process some data and return the results.

2. **link** (LINK in figure 4) is the interface between the transaction and physical layers. It manages the transmission of packets, guaranteeing delivery. The Link layer provides two services:
   - *Data request/response*. Data request is the service provided to the transaction layer allowing it to request the transmission of a packet to a given node or all other nodes, or informing it of the reception of a packet by a given node. *Data response* is used to send the acknowledgment.
   - *Data indication/confirmation*. Data indication signals to the transaction layer that a new packet has arrived, whilst data confirmation signals reception of an acknowledgement.

   The link layer subdivides an input data packet stream into a sequence of subactions, separated by subaction gaps (see fig 5). There are two kinds of subaction: (1) *split transaction* made up of a single packet (*subaction n*) and (2) *concatenated response transaction* made up of two packets (*subaction n + 1*). Each packet in a subaction is enclosed by a *data start* and a *data end* frame. Before starting a subaction, the link layer must gain access to the bus.

   In short, the main goal of the link layer is to transform data coming from the transaction layer into a sequence of packets, guaranteeing their correctness (via a checksum) and correct delivery (via acknowledgement).

3. The **physical layer** is the lowest layer (PHY in fig. 4); it provides initialisation and bus management services. This layer is also responsible for analog/digital conversion.
8.1 P1394: Specification and Synthesis

The case study presented in this section is the synthesis of the asynchronous mode of the link layer. As discussed, the direct synthesis method presented in this paper is based on three steps: 1) LOTOS specification of the system, 2) validation of the LOTOS specification and 3) synthesis of the specification into the hardware device.

The next sections deal with steps one and three. Step two is beyond the scope of this paper, because it is discussed in detail in FDT literature.

The LINK Layer specification

The Link layer provides a transaction with the possibility to transmit a packet to another node (or all the other nodes) using a insecure channel. To guarantee successful delivery, reception is acknowledged by the receiver. The LOTOS specification of the link layer is partially derived from [15] where it is possible to find the complete E-LOTOS specification used to validate the protocol and also to find a protocol error. The P1394 LOTOS specification has been rewritten using a different level of abstraction more tailored to synthesis. In particular we focus on a single node, whereas [15] focuses on the global behaviour of a system.

The following is a brief description of the protocol behaviour. The link layer protocol supports three different modes: send mode, receive mode and send ack mode. Starting from the initial state the link layer can evolve into send or receive. Send describes the actions needed to transmit the packet, await the ack and then return to the initial state; receive describes the action needed to manage the reception of a packet and the related acknowledgement from the transaction layer. After the reception of an ack from the transaction layer, the system can activate two submodes: concatenated response or split response. The first mode requires an immediate response to be provided, whilst in the second mode the system returns to the initial state and the response will be provided later.

Table 2 lists all the states traversed by the link layer and the corresponding LOTOS processes. We use the names used in [15]. For the sake of simplicity we will show the application of the proposed synthesis algorithm to the link0 process, which describes the initial behaviour of the link layer.
Table 2. Processes implementing the link layer states

<table>
<thead>
<tr>
<th>mode</th>
<th>LOTOS processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>Link0</td>
</tr>
<tr>
<td>Send mode</td>
<td>Link4, Link4DH, Link4RH, Link4RD, Link4RE, Link4DRec, Link4BRec</td>
</tr>
<tr>
<td>Receive mode</td>
<td>Link2req, Link2resp, Link3, Link3RA, Link3RE</td>
</tr>
<tr>
<td>Send ack mode</td>
<td>Link5, Link6, Link7</td>
</tr>
</tbody>
</table>

process Link0 {[<gates list>>](buf: SIG_TUPLE): noexit is [is_void(buf)=true] → LDreq ?dest:Nat ?h:HEADER ?d:DATA; Link0[...](...) [] [is_void(buf)=false] → PAreq !fair; ( PAcon !won; Link2req[...](buf) [] PAcon !lost; Link0[...](buf) ) [] PDind ?p:SIGNAL; ( [p=start] → Link4[...](buf) [] [p!=start] → Link0[...](buf) ) endproc

To describe the behaviour of link0 we need a buffer (buf) which is initially empty, i.e. its value is void. The link0 process can receive a packet from the transaction layer through gate LDreq, or a packet from the physical layer through gate PDind. The actions the link layer has to perform are as follows:

1. LDreq. link0 prepares the packet and transmits it to the physical layer, also requesting access to the bus. The link layer starts the arbitrating phase by sending the value fair through gate PAcon and awaiting either a positive response, won, or a negative one, lost, through the same gate. If the link layer gains access it evolves to link2req; otherwise it returns to its initial state represented by link0.

2. PDind. The link layer can receive the value start, which causes it to evolve into link4; otherwise the signal will be ignored and system will return to its initial state (link0).

P1394: link0 synthesis

In this subsection direct synthesis of the link0 process is shown. Our synthesis approach is based on the definition of a mapping between basic LOTOS operators and the equivalent Register Transfer Level (RTL) description. The aim of next section is to make the synthesis technique clear. A more complete discussion can be found in [16] [17].

Synthesis results
Figure 6 shows the RTL implementation of link0, which is obtained applying the previously described synthesis rules. Figure 7 is a high level scheme of the synthesised link0 process. For the sake of clarity, the data part has been omitted.

There are three blocks:

```
module Link0 : 
IN : LD_req, PD_cmd
OUT : PAReq, PAcmd
1 : activate(choice _1)

module choice _1 :
0 : LD_req = 1; PD_cmd = 1;
   if ((not(PARreq and 'is _void(buf))) and (not (LD_req and is _void(buf))) and (not PD_cmd));
      (PAreq and 'is _void(buf));
   elseif ((LD_req and is _void(buf)) and (not (PAreq and 'is _void(buf))));
      (PD_cmd and (not (PARreq and 'is _void(buf))) and (not (LD_req and is _void(buf))));
   goto(0;1;4;6)
1 : PARreq = 1
2 : if (not PARreq; PARreq) goto(0;3)
3 : PARreq = fair; exit(1)
4 : LD_req = 1
5 : dest := LD_req; h := LD_req; d := LD_req; exit(2)
6 : PD_cmd = 1
7 : p := PD_cmd; exit(3)

module choice _2
0 : if ((not PAcmd) and (not PAcmd);
     PAcmd;
     (PAcmd) and not PAcmd)
   goto(0;1;4)
1 : PAcmd = 1
2 : if (not PAcmd; PAcmd) goto(0;3)
3 : PAcmd = won; exit(1)
4 : PAcmd = 1
5 : if (not PAcmd; PAcmd) goto(0;6)
6 : PAcmd = lost; exit(2)

module choice _3:
0 : if ((not (p == start)) and (not (p!= start)); (p == start);((p == start)) and not (p!= start)) goto(0;1;2)
1 : exit(1)
2 : exit(2)
```

Fig. 6. RTL description of link0
1. three-way choice (block 1), which implements reception of the requests LDreq, PDind and PAreq. LDreq and PAreq are guarded by the conditions is_void(buf)=true and is_void(buf) =false.

2. two-way choice (block 2), which synthesises the choice between PAcon!win and PAcon!lost;

3. two-way choice (block 3) implements the guards, allowing subsequent behavior to be selected depending on the value taken by p.

The circuit resulting from synthesis is made up of 22 flip-flops and 51 logic gates. Table 3 summarises the results of synthesis of the full system.

9 Conclusions

Designing hardware devices is a hard task due to system complexity, strict time-to-market constraints and risks linked to safety applications. Using formal techniques to help the designer develop error-free devices represents a good solution but several tools are needed to simplify this approach.

Fig. 7. Synthesis of Link0 process
Table 3. Synthesis results

<table>
<thead>
<tr>
<th>Process</th>
<th>#FF</th>
<th>#Gates</th>
<th>Process</th>
<th>#FF</th>
<th>#Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link0</td>
<td>22</td>
<td>51</td>
<td>Link4</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Link4DH</td>
<td>16</td>
<td>33</td>
<td>Link4RH</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>Link4RD</td>
<td>8</td>
<td>14</td>
<td>Link4RE</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>Link4DRec</td>
<td>11</td>
<td>23</td>
<td>Link4BRec</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>Link2req</td>
<td>30</td>
<td>55</td>
<td>Link3</td>
<td>14</td>
<td>33</td>
</tr>
<tr>
<td>Link3RA</td>
<td>12</td>
<td>26</td>
<td>Link3RE</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>Link5</td>
<td>10</td>
<td>20</td>
<td>Link6</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>Link7</td>
<td>6</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this paper we focus on the synthesis of specifications starting from LOTOS. First, we describe the structure of the hardware device implementing the LOTOS choice operator; then, thanks to the LOTOS features, the specification of the device being implemented is transformed in order to use only choice operators. Choice translation is described by a parametric RTL expression which covers all possible types of synchronisation. Finally, an example of application of the proposed synthesis technique is presented. Further study is needed to optimise the resulting devices in order to reduce the number of both wires and flip-flops.

References

1. Ramayya Kumar, Christian Blumenroehr, and Dirk Eisenbiegler.


