Experiences with Specification and Verification in LOTOS: A Report on Two Case Studies

Carron Kirkwood*
and
Muffy Thomas
Department of Computing Science,
University of Glasgow, Scotland, U.K.
email: {carron, muffy}@dcs.gla.ac.uk

Abstract

We consider the problems of verifying properties of LOTOS specifications with specific reference to two case studies, one of which was proposed by an industrial collaborator. The case studies present quite different verification requirements and we study a range of verification and validation techniques, based on various behavioural congruences and preorders, which may be applied, also using some mechanised tool support. We consider the implications of the (formal) proofs which succeed or fail with respect to our desired properties, and draw some conclusions about the verification process.

1 Introduction

Over the last few years we have been studying some of the problems of verifying properties of formal specifications written in *LOTOS*[1], the ISO standardised language ([ISO:8807]) for concurrent, distributed, and nondeterministic systems. Some of the issues tackled include:

- which kinds of verification are needed, particularly for real case studies,
- how verification can be carried out,
- which techniques are best for different verification requirements,
- what degree of rigour is required,
- which kinds of proofs can be automated,
- the use of automated tools,
- how specification styles affect the verification tasks,
- what the results of verification reveal about the specifications, and
- what we can conclude about the state of the art in LOTOS verification.

*Funded by SERC grant "Temporal Aspects of Verification of LOTOS Specifications".

[1]*Language of Temporal Ordering Specification*

In this paper we give an overview of some of the work we have carried out, including the techniques and tools used, and two kinds of case studies considered. We describe our experiences with two particular case studies in some detail: a communications protocol and a control device for a dual mode radiotherapy machine. The former case study originates from a major U.K. defence contractor and the latter is an abstraction of a real machine. While some aspects of these two particular studies have been reported elsewhere, e.g. in [13] and [24], we use these two studies here to illustrate what we have learned about various aspects of the verification issues given above, as well as reporting on further work on the studies.

2 LOTOS

The reader is referred to the LOTOS standard [10] and [1] for an introduction to LOTOS. LOTOS consists of two parts: so-called *Basic* LOTOS, for describing interaction and flow of control, and ACT ONE, for describing equationally specified abstract data types. Basic LOTOS is very similar to CCS [21], drawing some aspects, particularly multi-way synchronisation, from CSP [9].

The semantics of LOTOS specifications are given by structured labelled transition systems (defined by inference rules). Various relations based on observable behaviour may be defined over these systems; we have found the most useful relations for verification purposes are **weak bisimulation** congruence and the testing relation **cred**. The full definitions of these relations may be found in [10].

Briefly, two processes are weak bisimulation congruent if they have the same observable behaviour in all contexts, ignoring occurrences of the internal action, and two processes are related via the **cred** preorder if, in all contexts, whenever one process passes a test, the other process does too. We choose these relations as the system under examination will probably have to interact with other systems, therefore it is important that it behaves in the same way in all contexts.

Two other relations commonly used are **strong bisimulation** equivalence and **trace** equivalence. We generally reject these; the former is often too strong because it does not give the internal action its unobservable status, and the latter is too weak because deadlock properties are not preserved.

In this paper, for brevity, we give specifications mainly written in Basic LOTOS. However, it is important to note that a good deal of our work has been concerned with Full LOTOS, i.e. specifications which also contain abstract data types, c.f. [24] and [15].

3 Verification and Validation

The terms verification and validation may be defined in a variety of ways in the literature. Although we do not feel that the distinction between them is crucial, we define them in the following, very general, terms.

verification Formal mathematical manipulation of properties using known truths and axioms. Typically, more generic properties such as mutual exclusion and freedom from deadlock are considered.

validation Proof by experimentation. Validation typically involves non-exhaustive analysis of a property specific to the particular example by testing and/or simulation methods, e.g. observation of the occurrence of a certain event.

Verification and validation are both concerned with specifying the system in the *desired* way (with respect to best practice), and with specifying the *desired* system (with respect to the informal requirements).

Given this understanding of verification and validation, what techniques do we have at our disposal to demonstrate that certain properties hold for a specification? Moreover, how may our perception of what needs to be proved alter with experience of a particular scenario?

3.1 Techniques

A verification technique used commonly in the process algebra literature is: given two descriptions of the same system, show that the two descriptions are related by one of the behavioural relations. Although we most often use a congruence relation (for the reasons given above), we can also use a preorder, which expresses some notion of partial specification, i.e. one description describes only some aspects of the system, while the other describes the full system. Examples of the use of these techniques are given in sections 5 and 6.4.

Another useful technique, which straddles the border between verification and validation, is *property testing*. This technique is based on the ability to express desirable (or undesirable) properties of a system as a *test* using process algebra. Although the underpinning of this technique is formal (it is based on testing equivalence [5]), it is often automated by simulation, or state reachability techniques, e.g. using LOLA [22]. We will give some examples of property testing and show the use of two different tools for carrying out proofs in section 6.

Finally, we briefly mention a technique which requires the introduction of another formalism. Process algebra is constructive and useful for expressing properties in which the ordering of events must be specified explicitly; however, we sometimes need to express properties such as safety and liveness directly, and this requires a non-constructive formalism such as temporal logic. A more extensive study of this topic may be found in [14].

3.2 Tools

Although there are many tools available for LOTOS, the functionality of most of them is related to editing, syntax-checking, compilation, etc. There are comparatively few tools which deal with the verification activities mentioned above. We consider some of the tools we might adopt below.

Several tools check equivalence between two process algebra expressions. Those we are aware of can be split into two groups: those based on semantic reasoning and those based on syntactic reasoning.

The first group includes, for example, AUTO [20], which is part of the LITE toolkit [19], and Cæsar [7]. We consider these tools to be based on semantic reasoning because they transform the specifications into graphs (labelled transition systems) and apply sophisticated and efficient partition algorithms to decide equivalence. These tools have the advantage of speed and full automation, but, as a special representation is used, the proof steps (assuming we *can* look at intermediate steps in the proof process) are not informative. In particular, if a proof fails, we may not gain any information as to *why* it failed. These tools are also prone to the state explosion problem, and cannot deal with infinite graphs (usually finiteness conditions are applied to the input, thus restricting the range of processes which can be analysed). A further disadvantage of these tools is that they are limited to Basic LOTOS and it is our intention to be able to reason about Full LOTOS processes. Any attempt to extend these tools to Full LOTOS will either result in loss of information, if data types are simply ignored, or in an explosion in the size (usually infinite) of the graph, if data types are encoded in the gate names to give a Basic LOTOS process.

Similar equivalence checking tools exist for other process algebras, for example, the Concurrency Workbench [3]. These tools can be used with Basic LOTOS because of the FC2 ASCII format which encodes the labelled transition system in a form readable by several tools.

The alternative to the graph-based approach is to use tools which are based on syntactic reasoning, i.e. manipulation of process algebra expressions via the inference rules of the transition system or equations of the chosen behavioural relation. Thus, we can avoid the need for a special representation for processes and provide proofs which are simple and easy to follow, being merely applications of the axioms of the relation. The main drawback of this approach is the lack of automation and high reliance on the skills of the user, who must frequently guide the proof. This may also be seen as a benefit, as such close interaction may lend additional understanding of the system under examination. Only a few tools are available for LOTOS which work on this principle, the main one being the tool of Inverardi and Nesi [6]. (Another one, [2], uses LCF to encode the inference rules, but it does not employ sophisticated rewriting techniques for proving the behavioural equivalences.)

We choose to adopt the syntactic approach for the reasons given above, but also because we wish to concentrate our efforts on one aspect of verification; a fair amount of work has been done already in the area of graph-based equivalence checking. We do not adopt the tool of Inverardi and Nesi because it deals only with finite Basic LOTOS. It is our intention to perform proofs over as much of the language as possible, and although the proof system of Inverardi and Nesi is claimed to be modular and easily extensible, this is only true if the developer is familiar with Prolog programming. Our intention is to develop a system in which the user only needs to know about the laws of the LOTOS relation used, even in the case that new laws or even relations have to be added.

In the first instance we adopt a general rewriting tool, on top of which we can build our specialised LOTOS rules. We use RRL [12] to develop sets of rules for LOTOS relations because it supports rewriting and (Knuth-Bendix) completion modulo associative-commutative axioms. Sadly, such tools often have poor user interfaces and we turn to PAM [17] (Process Algebra Manipulator) because of its generality and nice graphical interface.

The technique of property testing can be implemented by a simulation tool, but the user then has to perform the analysis of the behaviour manually. Full automation of property testing for LOTOS is given by LOLA [22], which supports symbolic computation. This tool is very valuable because, unlike PAM, it is also a prototyping tool which supports explicit testing (i.e. state exploration) and Full LOTOS.

Finally, we do not consider the automation of model checking logical formulae with respect to LOTOS processes here; see [14] for more detail on this for Basic LOTOS processes.

The specifics of the tools adopted are described in more detail in sections 5.3 and 6.4. Our aim in this paper is to describe our experiences with particular tools, rather than to provide an exhaustive comparison of LOTOS verification tools.

4 Case studies

The two case studies presented in the following two sections illustrate quite different verification requirements and approaches to discharging those requirements. In the first study, verification involves formalising and demonstrating *satisfaction* between two specifications, and in the second study it involves formalising and demonstrating that an *undesirable property* does not hold for a specification.

More specifically, the first case study involves the common verification problem of demonstrating that a desired relationship holds between a *given* specification and a *given* implementation, i.e. the very realistic case when the implementation has *not* been derived from the specification directly. This case study, see [13] and [15] for more details, considers an abstraction of a real communications problem for which the specification, implementation *and* verification requirements were laid down at the beginning. The problem was originally proposed by an industrial collaborator, but is interpreted here as a login protocol, since the real end-user application is confidential.

The second case study involves specifying the high-level behaviour of a control device for a linear accelerator, or medical radiotherapy machine. In this case study we use LOTOS to specify the high-level control processes, and then attempt to prove that the specifications permit safe, or possible unsafe, behaviours. Thus, an initial activity is to identify the safe, and unsafe, behaviours. Several specifications are developed in [24]; here, we concentrate on only two such specifications. (Note: verification of some of the actual assembler software controlling such a machine is contained in [23].)

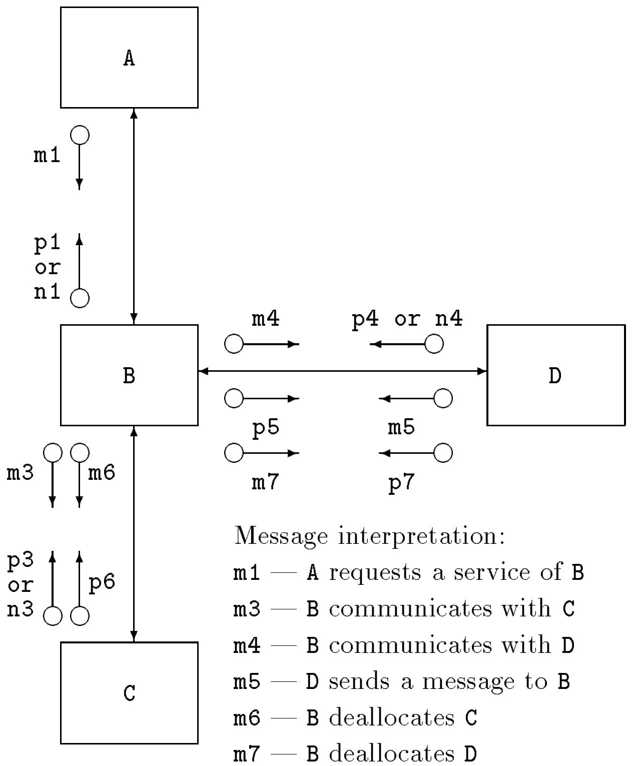
In each case study, for brevity, we abuse the LOTOS notation and omit various nonessential features (e.g. process gate parameters and termination type) where possible.

5 Login protocol

This example involves four entities, A, B, C and D, communicating as shown in figure 1, in which a box represents an entity, and a ○⟶ represents a message (sent in the direction of the arrow). Each message is labelled by mx, where x is a number in $\{1, 3, 4, 5, 6, 7\}$. Informal interpretations of the mx are also given in figure 1. Messages of the form px or nx are positive and negative acknowledgements, respectively, to the corresponding mx messages.[2]

We were initially given two formal descriptions of the system: a *specification* in the form of a group of protocols describing separately the interactions between A and B, B and C, and B and D, and an *implementation* given by a group of processes corresponding directly to A, B, C and D. We were also given an intuitive informal description of the system. Close inspection of the formal descriptions, given by a grammar, revealed inconsistencies with respect to the intuitive informal description; in order to have a formal description which matches our intuition we re-specify the system, using LOTOS for the new descriptions. Samples of the new LOTOS descriptions are given in the next section. We deliberately do not remove obvious inconsistencies between the specification and the implementation because one of the problems we wish to consider is the difficulty of reconciling the differences between a specification and an implementation

[2]Note that some messages only require a positive acknowledgment, while others require both positive and negative acknowledgments (see figure 1) — this is to do with the nature of the messages which they acknowledge, e.g. it does not make any sense to allow C to respond in a negative way to the message m6 "deallocate".



Message interpretation:

m1 — A requests a service of B

m3 — B communicates with C

m4 — B communicates with D

m5 — D sends a message to B

m6 — B deallocates C

m7 — B deallocates D

Figure 1: The Processes and their messages

not formally derived from that specification.

To help illuminate the system, we invented a possible interpretation of our own which provides some intuition as to what happens in the system, although it is not an exact match. We view the system as follows: A is a user wishing to log-on to a system with a username and a password. C takes a username and checks that it is valid. D takes a valid username, acknowledges receipt of the name, and then returns the corresponding password. B co-ordinates these activities to ascertain if A is a valid user and has supplied the correct password. B has an internal timer which "times out" if D does not reply to the communication within a previously set time limit. B must send deallocation messages to C and D when they are no longer required. Since we use Basic LOTOS to model the example, the password and username are not in the formal description of the system; a future extension might be to move to Full LOTOS and to include these features.

5.1 The problem in LOTOS

Communication in the system is governed by protocols P1, P2 and P3. Each protocol describes the interface between just two of the processes in the system, e.g. P2 describes the interface between B and C, completely ignoring the actions of A and D. This means the alphabets of the LOTOS processes P1, P2 and P3 are disjoint. We give an informal description of just one of the protocols, P2, below, together with its formal LOTOS description.

In the protocol P2, B sends m3 to C which must be acknowledged by p3 or n3. Following p3, B may or may not send m6 to C which must be acknowledged by p6.

```
process P2 :=
  m3; ( n3; P2
     [] p3; (P2 [] m6; p6; P2))
endproc
```

The implementation of the system is achieved by four interacting processes. As above we give only one of the processes, for reasons of brevity.

In a successful execution of B receive m1 from A, allocate C with m3 p3 and D with m4 p4, then set a timer because D must send m5 within some time limit. When m5 arrives cancel the timer and reply with p5. C and D are deallocated by m6 p6 and m7 p7 respectively. Finally signal the success of the transaction by sending p1 to A.

This sequence of actions may fail in a number of ways: either C or D could refuse to participate by returning negative acknowledgments (n3 or n4), or D might not send m5 within the time period, in which case the timer "times out". In these cases reply n1 to A. Deallocation of C and D occurs if and only if they originally agreed to participate in the transaction, i.e. if p3 and p4, respectively, were sent.

```
process B :=
m1; m3;
  (  n3; n1; B
   []p3; m4;
       (  n4; m6; p6; n1; B
       [] p4; set;
          (  timeout; m6; p6; m7; p7; n1; B
          [] m5; tcancel; p5; m6; p6; m7;
               p7; p1; B)))
endproc
```

Now we have formal descriptions of the specification and the implementation, we can try to verify that the implementation is correct with respect to the specification. Part of our problem is choosing the best way to express this requirement formally.

5.2 Formalising the verification requirement

Given that we have two different descriptions of the system, the verification requirement can be expressed as: does the implementation (the processes A, B, C and D) satisfy the specification (the protocols P1, P2 and P3)?, i.e.

$$(\mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \mathtt{D})\ \mathit{sat}\ (\mathtt{P1} \mid \mathtt{P2} \mid \mathtt{P3}) \qquad (1)$$

where *sat* stands for one of the LOTOS behavioural relations (probably a congruence relation, therefore the orientation of the equation doesn't matter) and the "|" operator denotes the general parallelism, $|[l]|$, operator of LOTOS. Variations of the events in the synchronisation list, l, of the parallelism operator give subtly different combinations of the components of the system, i.e. the combinator used in `A | B` may be different from that used in `C | D`, or `P1 | P2`.

An alternative approach is to consider reflecting the modular nature of the specification (the protocols are all disjoint) in our formulation of the conjecture to be proved; possibly also simplifying the proof process. This gives

$$(\texttt{A} \ | \ \texttt{B}) \ \mathit{sat} \ \texttt{P1} \tag{2}$$
$$(\texttt{C} \ | \ \texttt{B}) \ \mathit{sat} \ \texttt{P2} \tag{3}$$
$$(\texttt{D} \ | \ \texttt{B}) \ \mathit{sat} \ \texttt{P3} \tag{4}$$

and the correctness of the system as a whole is expressed by $((2) \wedge (3) \wedge (4))$.

As they stand, equations (2), (3) and (4) are not quite correct since the language of the left-hand expression may not be the same as that of the right-hand expression, e.g. `A | B` will use events not mentioned in `P1`. The most straightforward way of ignoring these events is to use the `hide` operator, but it may be possible to find an interpretation of *sat* which takes account of the extra events. As discussed in section 2, our choice of relations in LOTOS is restricted to weak bisimulation congruence, testing congruence and the **cred** preorder. None of these relations can abstract away information about the events in the way we require for this example, therefore in formulating our verification requirement we hide the extra actions.

A further problem, if we choose the modular formalisation of the verification requirement, is that we must be sure that satisfying the correctness of the parts is the same as satisfying the system as a whole. Since `P1`, `P2` and `P3` are concerned with distinct aspects of the system, it seems likely that the verification can safely be split into parts. In general, we can say that the success or otherwise of this formalisation depends on choosing the right methods of splitting up the system, hiding unimportant events, making individual proofs, and recombining the results.

¿From the above discussion we can see that there are several ways of expressing the verification requirement (if we consider the three different behavioural relations mentioned above as interpretations of *sat*), and that none of them seems either more or less appropriate than the others. Indeed, this is one of the problems of verification; ensuring that the formalisation of the verification requirement(s) reflects our intuitive picture of these requirements. To try to gain more understanding of the practical differences between different formalisations we approach the verification from an experimental point of view; we try to prove each conjecture given by different instantiations of equations (2), (3) and (4) above in turn, starting with the strongest behavioural relation and, if that fails, progressing to a weaker one.

More concretely, the general form of the conjectures are

$$\begin{aligned}\texttt{P1} \quad \mathit{sat} \quad & \texttt{hide CDevents in} \\ & (\texttt{A} \ |[\texttt{m1,p1,n1}]| \ \texttt{B})\end{aligned} \tag{5}$$
$$\begin{aligned}\texttt{P2} \quad \mathit{sat} \quad & \textbf{hide} \ \texttt{ADevents in} \\ & (\texttt{C} \ |[\texttt{m3,p3,n3,m6,p6}]| \ \texttt{B})\end{aligned} \tag{6}$$
$$\begin{aligned}\texttt{P3} \quad \mathit{sat} \quad & \texttt{hide ADevents in} \\ & (\texttt{D} \ |[\texttt{m4,p4,n4,m5,p5,m7,p7}]| \ \texttt{B})\end{aligned} \tag{7}$$

```
where
  CDevents = [m3, p3, n3, m4, p4, n4, m5, p5,
               m6, p6, m7, p7]
  ADevents = [m1, p1, n1, m4, p4, n4, m5, p5,
               m7, p7]
  ACevents = [m1, p1, n1, m3, p3, n3, m6, p6]
```

and we replace *sat* in these equations by either weak bisimulation congruence, testing congruence, or **cred** (and then we also reverse the order of the operands).

5.3 Use of tools

Initially our strategy was to automate the proofs by a general rewriting tool, namely RRL [12], tailored for use with LOTOS by including a set of rules expressing weak bisimulation congruence. The rules are derived from the laws for weak bisimulation congruence given in [10]. The exact details of this rule set may be found in [15].

The advantages of this approach are that no special implementation is required, that the system may be easily extended to other relations just by defining new rule sets, and that the proofs of equivalence are fully automated if the rule set is *complete*, i.e. confluent and terminating. Although we can obtain a complete rule set based on a *subset* of the equations in appendix B.2.2 of [10], too many aspects of weak bisimulation congruence are omitted.

For example, in hand proofs a particularly useful law is the one which allows us to convert instances of the parallel operator into instances of action sequence and choice. If we add rules expressing this law to our rule set it becomes non-terminating, leaving us with a semi-decision procedure for the relation. This was not unexpected, since weak bisimulation congruence is undecidable in general. This requires us to alter our proof strategy to include hand proofs in the case of RRL asserting that two processes are inequivalent.

During our first proofs with RRL we discovered several deficiencies in the approach. The main one was the inability to express recursion in the rewriting framework, but we also found RRL to be inflexible and difficult to operate. To solve both of these problems we adopted a new tool, PAM [17], the process algebra manipulator. PAM is a rewriting tool, parameterised on a definition of a process algebra, with some special features built in to make defining and reasoning about process algebra equivalences easier. PAM also has a nice graphical interface and is very easy to use. We have defined input files for PAM which allow us to manipulate processes using the equations of strong and weak bisimulation congruence, branching bisimulation congruence and testing congruence. We also have some rules relating to the reduction of processes with respect to the **cred** preorder, however, these reductions are sound only under certain conditions and should be used with care.

Essentially, PAM is an automated pencil and paper, so although it helps with the book-keeping of a proof, i.e. ensuring that axioms are applied correctly, recording which axioms are applied, and eliminating transcription errors, it can provide only partial automation of the proof because user guidance is required to determine which rule to apply next. A lesser drawback is that PAM is limited to Basic LOTOS; however, we hope that a new version of PAM, VPAM [18], will allow us to reason about Full LOTOS specifications. If we pursue this direction, a problem to be overcome is that VPAM assumes the CCS model of communication (two-way synchronisation) whereas LOTOS has multi-way synchronisation. Note that PAM does not suffer from this problem: the communication mechanism is completely user-defined.

Detailed descriptions of our experiences with RRL and PAM may be found in [15]; the next section gives an overview of our results.

5.4 Results

An initial attempt to show the specification is equivalent to the implementation, using RRL, fails, regardless of the relation chosen to replace *sat* in equations (5), (6) and (7). As mentioned above, this on its own is not a conclusive result since we only have semi-decision procedures for the relations. Further examination of the normal forms (i.e. the terms obtained by applying the axioms of the relation until no more can be applied) of the left and right hand sides of the equations highlights irreconcilable differences between them and, in fact, we can show by hand that none of these interpretations of the verification requirement holds. The system is *not* correct.

The reason for the failure of the proofs is that the protocols constitute a *partial* specification of the system, and within process algebra we have only limited ways of expressing this, i.e. using **cred** and/or `hide`. In this case this formalisation was not good enough, mainly because the use of `hide` introduced nondeterminism to the specification which was not present in the specification without the `hide` operator. One way to solve this problem is to adopt a different formalism and proof technique which *can* express partial specifications correctly, e.g. temporal logic. Using process algebra we have to construct the system, specifying exactly which actions occur and the order in which they occur. When applying the `hide` operator we then lose important information about why certain choices are made in the system (they become nondeterministic). On the other hand, a temporal logic formulation of the protocols allows us to concentrate on the events of interest and to express the notion that some other events may occur, but without specifying exactly which events occur, how many events occur, or the order of occurrence. Although our current work pursues that possibility [14], here our aim is remain within process algebra, allowing us to continue using the tools already developed.

Since the modular approach to expressing the verification requirements fails, we try the approach which compares the processes all combined with the protocols all combined, as expressed in equation (1). No relationship can be demonstrated here because there is no meaningful way in which to combine the protocols since they are only partial specifications of the system's behaviour. For example, we might consider using the interleaving operator, |||, since the protocols contain no common events; however, this results in a state explosion. The resulting process contains many traces which do not appear in (`A | B | C | D`), and which do not reflect our intuition about the behaviour of the system.

Another way of viewing this problem is to say that the environment in which we have placed the process is not restrictive enough. When we give a partial specification we are making assumptions about the environment in which that specification is placed, and when carrying out the verification we must ensure that that information is somehow included in the evaluation of the system.

For this example, information missing from the specification includes details of the timer and the criteria by which success or failure in the system is measured. These details can be specified as processes and added to the original protocols in a modular way using the constraint-oriented style of specification [26]. Since the full example is too large to be given here, we illustrate the procedure with just one of the new specifications.

Success or failure in the system can be expressed as

```
process system   :=    m5; p1; system
                    [] n3; n1; system
                    [] n4; n1; system
                    [] timeout; n1; system
endproc
```

We combine this process with the protocols, using an appropriate synchronisation list.

```
((P1 |[p1,n1,n3,n4,m5]| system) ||| P2) ||| P3
```

Combining the `system` process with `P1` first helps avoid the state explosion resulting from the interleaving of the three protocols.

Following a similar procedure for the other information missing from the specification, we obtain a new specification which can be shown to be equivalent to the original implementation; i.e. using RRL we can finally prove the verification requirement is satisfied, as long as we modify the specifications to give finite processes (by replacing recursive process references by `exit`).

Obviously it is not sufficient to be able to deal only with finite processes since concurrent systems typically display infinite behaviour. RRL is unable to deal with infinite processes, therefore we move to PAM with which we *can* prove the verification requirement is satisfied for the infinite processes given above. The move to infinite processes is not trivial as the synchronisation properties of the processes change when they are defined to be recursive. These problems were easily solved by a little trial and error.

It is interesting to note that our intuitive ideas about the verification requirement of this system were forced to change as a result of experimental formalisation and proof of the verification requirement. A large part of our work has been in the trial and error of finding out what we really wanted to prove, and what we actually could prove. Clearly, a difficult yet important task for the user of formal methods is to formalise the exact property (or properties) to be demonstrated, and one needs to be aware that there are no hard and fast rules about how to do this.

We now turn to the second case study.

6 Radiotherapy control device

This example concerns a dual mode linear accelerator and is motivated by the reports of the Therac-25 accidents [11]. The machine delivers two kinds of radiation therapy: electron and X-ray. The electron beam is used to irradiate the patient directly, using scanning, or bending magnets to spread the beam to a safe and therapeutic concentration. The X-ray beam is created by bombarding a metal shield, or beam flattener; the electrons are absorbed by the shield and X-rays emerge from the other side. Since the efficiency of producing X-rays in this way is very poor, the current of the electron beam has to be greatly increased when used for X-ray therapy.

Our aim is to specify the high level behaviour of a control device for such a machine in LOTOS, and to verify the machine can only operate *safely.*

6.1 The problem in LOTOS

There are events for altering the beam intensity and the shield position: `hb` - *high beam*, `lb` - *low beam*, `hs` - *high shield*, `ls` - *low shield*; and there are events for choosing X-ray or electron mode, and for firing the electron beam: `xr` - *X-ray mode*, `el` - *electron mode*, `fire` - *fire beam.*

The 'default' situation is that both the beam and the shield are low, i.e. the machine is ready to irradiate directly with the electron beam. Therefore, in order to operate in X-ray mode, the beam and shield are set to their respective high intensity and high positions; after firing in X-ray mode, the beam and shield are set back to their respective low intensity and low position. At any point during a radiation treatment, the process may be interrupted and another type of treatment may be chosen or the treatment restarted.

The formal specification of `Control_Device` is given below. The top-level process is `STARTUP` which calls the parameterised process `SETUP` to initialise the machine and set the beam and shield to low, before calling the main process `TREATMENT`. The process `TREATMENT` offers a choice between the X-ray mode, the electron mode, or termination by `exit`. The processes `XRAY` and `ELECTRON` specify the X-ray and electron mode behaviour (respectively); both processes call the process `TREATMENT` at the end of the 'normal' behaviour, and they may be interrupted, or disabled, by the process `TREATMENT` at any point during the 'normal' behaviour. All events are externally visible.

6.2 Verification

This specification can be deceptive: although it is quite short, the use of recursion and the disabling operator allows for *many,* possibly infinite, behaviours. The state explosion is quite dramatic and validation is not quite as simple as it might first appear.

```
process STARTUP :=
   SETUP[lb,ls] >>
     TREATMENT
endproc

process SETUP[ev1,ev2] :=
   (ev1; exit) ||| (ev2; exit)
endproc

process TREATMENT :=
      (xr; XRAY)
   [] (el;ELECTRON)
   [] exit
endproc

process ELECTRON :=
   (fire; TREATMENT)
      [> TREATMENT
endproc

process XRAY :=
   (SETUP[hb,hs] >> (fire; SETUP[lb,ls)
                 >>  TREATMENT)
      [> TREATMENT
endproc
```

Figure 2: Control Device

Our particular interest is in the *safety* of the specification. We note that with respect to verification, the word safety is overloaded. Here, we specifically mean that life is not endangered. However, we shall only be considering safety properties, in the other sense, i.e. properties which state that something bad should *not* happen, as opposed to liveness properties which state that something good *should* happen.

The greatest danger posed by the machine is the possibility of irradiating a patient directly with the high intensity electron beam; this unsafe property is characterised by:

The machine is unsafe when the electron beam is fired at high intensity and the shield is in the low position.

In order to verify that the specification is unsafe/safe, we must show that unsafe property does/does not hold; this requires a formalisation of the unsafe property with respect to the specified traces. We have explicitly chosen to define the *unsafe* property, since it is easier to identify the unsafe traces than the safe ones. Note also that we need not consider all traces over all events, but only those traces which are specified behaviours of the machine. One such unsafe trace has the property that it begins with the initialisation events (i.e. `lb` and `ls` occurring in any order), and there is an occurrence of `hb` which is neither preceded by a `hs` nor succeeded by a `lb` or `hs`. Thus, when the fire event occurs, the beam is high and the shield is low. Formally, these are traces prefixed by traces of the form: `((lb;ls)|(ls;lb)); (not(hb|hs))*; hb; (not(lb|hs|fire))*; fire` where we use the notation * for zero or more occurrences, | for choice, and `not(x|y)` to denote the choice of all events, *excluding* events `x` and `y`. For example, one such unsafe trace is: `lb;ls;xr;xr;hb;xr;hb;el;fire`. Since we have identified a class of unsafe traces, namely the traces are described by a regular expression, this suggests that property testing is the appropriate verification technique.

6.3 Property testing

Testing in LOTOS is a form of state reachability analysis on the underlying labelled transition system, i.e. do certain events occur on all, none, or some paths, or traces, from the root of the system. Testing can be carried out at the level of behaviour expressions, by expanding (possibly parameterised) behaviour expressions, with respect to a given equivalence; in our case, weak bisimulation congruence. *Property testing*, [22], is a more abstract, and especially useful, form of testing for a specific property. The given property is specified as a LOTOS process, the *test*, which concludes with a special user-defined test success event. The test process is then composed in parallel with the process under test, synchronising on the observable events in the test process, except the special test success event. Due to the multi-way synchronisation of the LOTOS parallel operator, if the process under test can perform the behaviour described by the test process, then eventually the special test event will be observed. The tool we use to automate property testing is LOLA [22], a simulation tool which operates by constructing and exploring the labelled transition system corresponding to the LOTOS expression.

Properties need not be defined by regular expressions; indeed, any context-free language of traces can be specified in LOTOS. Thus, property testing is a valuable validation technique which has a formal basis.

The LOTOS process `UNSAFETEST` in figure 3 gives the combined test and process under test, for this example. Clearly, if the test event `tok` can be reached, then the machine is *unsafe*. We would also like to be able to conclude that if it cannot be reached, then the machine is *safe*. Of course, these safety judgements are made with respect to the *unsafe* property defined above.

6.4 Proving the specification is unsafe

For automated assistance, we used two different tools: LOLA and PAM, employing slightly different implementations of the technique.

Results with LOLA. Since LOLA is a testing, or simulation tool, we carried out property testing by simply expanding `UNSAFETEST`, looking for occurrences of `tok` (using the `testexpand` command). Since `UNSAFETEST` specifies *non*terminating processes, an expansion depth is required when performing the expansions. A judicious choice of depth proved to be very important: too small and the test was rejected because insufficient process behaviour had been explored; too large and the heap was exhausted. In our case, on a Sun Sparc workstation, the test was rejected when depth<12 and memory was exhausted when depth>13! This experience confirms the experimental and possibly inconclusive nature of testing; *fortunately*, the test passed with depth=12. Thus, we may conclude that `Control_Device` is *not* safe.

```
process UNSAFETEST :=
          STARTUP
    |[fire,lb,hb,ls,hs,xr,el]|
   ((SETUP[lb,ls] >> TEST)>> tok;exit)
endproc

process TEST :=
   Not_hbhs >>
    (hb; Not_lbhs) >> (fire; exit)
endproc

process Not_hbhs :=
     fire;Not_hbhs
  [] lb;Not_hbhs
  [] ls;Not_hbhs
  [] xr;Not_hbhs
  [] el;Not_hbhs
  [] exit
endproc

process Not_lbhs :=
     ls;Not_lbhs
  [] hb;Not_lbhs
  [] xr;Not_lbhs
  [] el;Not_lbhs
  [] exit
endproc
```

Figure 3: Unsafe test

Results with PAM. We may also express property testing in terms of a relation between processes which allows us to use PAM for this technique.

To prove the device unsafe in PAM we use the following basic formulation of the property to be proved:

`tok;exit` **cred** `(hide [lb,hb,ls,hs,el,xr,fire]`
`in UNSAFETEST)`
= true

The use of **cred** expresses the notion that at least one trace of `UNSAFETEST` has the behaviour we are looking for (the occurrence of `tok`), although there may be lots of other behaviours. An equivalence relation is too strong in this case as it would also take account of these other behaviours. All events other than `tok` must be hidden in `UNSAFETEST` to make comparison with `tok; exit` possible.

To automate the proof, some further modifications to the specification are necessary because of limitations of PAM; in particular, parameter passing is not supported. The main modification, which is syntactic, is that instead of having a parameterised process `SETUP` as above, we have two processes `SETUPH` and `SETUPL` which are hardwired versions of `SETUP[hb,hs]` and SETUP[lb,ls] respectively. The other processes of the specification are also parameterised (in *correct* LOTOS), but the list of actual parameters matches the list of formal parameters therefore having no effect on the process behaviour, so we omit these freely.

Using PAM and algebraic manipulation of the property to be proved above, we are able to show that the relation holds, and that therefore the radiotherapy machine is unsafe. This serves to reinforce the result gained with LOLA.

6.5 Proving the specification is safe

A simple modification to the specification, to make it *safer*, is to remove the parallelism in the process SETUP, i.e. to replace the parallelism in SETUP by: ev2;ev1;exit and to replace SETUP[lb,ls] by SETUP[ls,lb]. Now consider verifying that this specification is safe.

Property testing was a good technique for the first specification because it was unsafe, i.e. it *passed* the unsafe test. But now we want to prove that the unsafe property *does not* hold, i.e. that the unsafe test is never passed. Property testing, in general, only provides a semi-decision procedure when the processes under investigation are infinite: in these cases, test rejection proves nothing conclusive as we can only try a finite number of test depths. In addition, we can never be sure that we have considered all possible test cases.

However, testing can be used to reason about infinite processes if they have a finite number of states. Namely, one way to prove that the test is never passed is to transform the combined specification into a set of recursive equations and then examine the non-recursive prefixes to ensure that the test event does not occur.

Experience with LOLA and PAM. Initial attempts with property testing were that for every depth tried, as expected, the UNSAFETEST test was not passed, and for depths>13 the result was inconclusive as the memory is exhausted. So, what next?

While a major drawback of LOLA is the inability to explictly manipulate expressions, a nice feature is its ability to recognise multiple occurrences of the same state. *Unfortunately*, initial attempts to find a finite number of states failed, and we were, at first, baffled by this. However, the computer *is* always right (or at least it was in this case!), and by changing to PAM, where the user is required to interact closely with the terms of the proof, we discovered the problem. In our specification, each expansion, or unfolding, of the expressions involving the disable operator, [>, introduces another occurrence of that operator.

Specifically, every expansion of TREATMENT[..] introduces a further TREATMENT[..][> TREATMENT[..]. We cannot simply replace the former by the latter using the laws of bisimulation because, in general, P [> P is *not* bisimilar to P. However, P1 [> P1 is bisimilar to (i.e. is a solution for) P when they are of a certain form; for example, when P = P1 [] a;(P1 [] b;P) and P1 = a;b;P1. Note that P is obtained here by the weak bisimulation congruence expansion law for [> applied to P1 [> P1.

A similar argument allows us to unfold the specification of TREATMENT[..], avoiding the use of [> altogether. The resulting specification (of the altered subprocesses only) is:

```
process ELECTRON :=
   (fire; TREATMENT)
[] TREATMENT[fire]
endproc

process XRAY :=
(TREATMENT
[]hb;(TREATMENT
     []hs;(TREATMENT
          []fire;(TREATMENT
                 []lb;(TREATMENT
                      []ls;TREATMENT)))))
endproc
```

Given this specification, we were able to find a set of recursive equations after expansion in both LOLA and PAM:

```
    UNSAFETEST = lb; ls; T
```

where T is

```
    TREATMENT
     |[lb,ls,hb,hs,xr,el,fire]|
    (TEST >> tok; exit)
```

After several unfoldings, we have

```
  T =    (i;stop)
      [] (xr;((i;stop)[]T))[](el;(fire[]T))
```

The test event tok clearly does not occur on any finite trace and so we conclude (by unique fixed points) that the test cannot be passed. Note that with PAM we draw this conclusion manually, because the implementation of the **cred** preorder does not allow us to prove conjectures of the form (P1 **cred** P2) = false, whereas LOLA is able to recognise duplicate states and conclude automatically (for a specific depth, of course) that the test is always rejected.

Further extensions to this case study include interaction with an operator and data types modelling shield and beam positions and error messages. For brevity, we present only the introduction of data types.

6.6 A specification in Full LOTOS

In this section we consider an extension to the specification which enables us to more explicitly reason about, and *avoid*, unsafe behaviour. Namely, we would rather be able to check that we are not in an unsafe state before firing, than have to be concerned with the *way* in which an unsafe state is reached. Full LOTOS is ideally suited to modelling this need for state, since values are not only offered with events, but also processes may be abstractions over values. Thus, our processes may be parameterised by the status of the beam and the shield and then, instead of reasoning about the traces leading up to an event, we need only consider the current status of the beam and the shield before firing.

We now proceed to parameterise the processes by the status of the beam and shield and to extend the specification to include the reporting of errors. We also use correct LOTOS syntax here, since it is now important to indicate the observable events and process functionality.

Datatypes. Three datatypes, in addition to the usual (library) type boolean, are required in this specification: the type `SHIELD`, the type `BEAM`, and the type `ERROR`. The specifications of these types are given in Figure 4.

The type `SHIELD` includes two constants: `up` and `down`, and a test for equality.

The type `BEAM` includes three constants: `high`, `mid`, and `low`, and a test for equality. Three values are included to model the possibility that the beam intensity may fall outwith the expected intensities. A more detailed specification would include many more discrete values for this type (perhaps it should be countably infinite), but for our purposes, one value which is neither `high` nor `low` will suffice.

Finally, we include a type of errors, to indicate that unsafe and nonstandard states have been reached.

Processes. The processes are similar to those in the first specification, with the addition of parameterisation over beam and shield values, but there are a few significant differences.

The first is that since we are no longer concerned with traces, but rather with the status of the beam and shield when a `fire` event occurs, the other events (i.e. those which alter the status of the beam and shield, and the choice of xray and electron mode) will be hidden.

The second difference is that we have dispensed with the process `SETUP` and perform the relevant set up events sequentially (as in the safer specification introduced earlier). Also, while LOTOS does permit value passing over process enabling (e.g. `P >> accept x:X in Q`), it does not permit value passing over process disabling. This is most unfortunate for us, but perhaps understandable because the semantics of such a construct would be quite complex; the values being passed would have to depend on the point at which the disable occurs. Thus, as in the safer specification, we have expanded processes `ELECTRON` and `XRAY`, using the weak bisimulation congruence expansion laws (and the "local" laws, i.e. the laws specific to this example, referred to earlier), though now we also include the appropriate values.

The final difference concerns the event `fire`. In processes `ELECTRON` and `XRAY`, the event `fire` is replaced by a process `FIRE`. This process is also parameterised by a beam and shield value and 'traps' the unsafe/nonstandard situations by calling the process `ERROR`, with an approriate error; this process then returns to `TREATMENT`.

6.7 Proving the specification is safe

Now the unsafe test is formalised very concisely by the event: `fire!high!down`. If there are traces which include this event, then the machine is unsafe. Using LOLA, we find a set of recursion equations, as before, and after examination, we conclude that the

```
type SHIELD is boolean
sorts shield
opns  up, down  : -> shield
_eq_ : shield, shield -> bool
eqns
ofsort bool
   up eq down   = false;
   up eq up     = true;
   down eq down = true;
   down eq up   = false;
endtype

type BEAM is boolean
sorts beam
opns high, mid, low : -> beam
   _eq_ : beam, beam -> bool
eqns
ofsort bool
   high eq low  = false;
   high eq high = true;
   low eq low   = true;
   low eq high  = false;
   high eq mid  = false;
   low eq mid   = false;
   mid eq low   = false;
   mid eq mid   = true;
   mid eq high  = false;
endtype


type ERROR is boolean
sorts error
opns  unsafe, nonstandard  : -> error
endtype
```

Figure 4: Beam, Shield and Error Datatypes

```
process STARTUP[fire]:exit(beam,shield):=
  hide lb,ls in
    lb; ls; TREATMENT[fire](low,down)
endproc

process TREATMENT[fire](b:beam,s:shield)
                    :exit(beam,shield):=
  hide xr,el in
       (xr; XRAY[fire](b,s))
    [] (el; ELECTRON[fire](b,s))
    [] exit(b,s)
endproc

process ELECTRON[fire](b:beam,s:shield)
                   :exit(beam,shield]:=
     (FIRE[fire](b,s) >>
        TREATMENT[fire](b,s))
  [] TREATMENT[fire](b,s)
endproc

process XRAY[fire](b:beam,s:shield)
                :exit(beam,shield):=
  hide lb,hb,ls,hs,xr,el in
     TREATMENT[fire](b,s)
  [] hb;(TREATMENT[fire](high,s)
    [] hs;(TREATMENT[fire](high,up)
      [] (FIRE[fire](high,up) >>
           (TREATMENT[fire](high,up)
        [] lb;(TREATMENT[fire](low,up)
          [] ls;
             TREATMENT[fire](low,down))))))
endproc

process FIRE[fire](b:beam,s:shield)
                             :exit:=
   hide err in
    [(b eq high) and (s eq down)]
                 ->ERROR[err](unsafe
 [] [(b eq high) and (s eq up)]
                 ->ZAP[fire](b,s)
 [] [(b eq low)]
                 ->ZAP[fire](b,s)
 [] [not(b eq high) and not(b eq low)]
                 ->ERROR[err](nonstandard)
endproc

process ZAP[fire](b:beam,s:shield):exit:=
   fire!b!s; exit
endproc

process ERROR[err](e:errnum):exit:=
   err!e; TREATMENT[fire](low,down)
endproc
```

Figure 5: Control Device with State

event `fire!high!down` *cannot* be reached. Thus we can conclude that this specification is safe.

6.8 Results

This study of the radiotherapy machine demonstrates one of the clearest benefits of using formal methods: uncovering design errors[3]. Accordingly, the proof of the existence of erroneous, or unsafe, behaviour of the machine was easier, and more convincing, than the verification of the safer specification.

This study also demonstrates the usefulness of a formally-based validation technique, i.e. property testing and LOLA: often the simplest of techniques takes us quite far in discharging proof obligations, though it did, ultimately, break down as an automated technique. When it did break down, we needed to change to a less automated tool such as PAM which allowed us to investigate the problem and then to recognise the transformation, particular to our specification, required.

We took two approaches to specification and testing: one was trace, or history, based and the other was state based. In the trace based approach, we considered only the traces which were specified as valid machine behaviours, i.e. they all begin with `lb` or `ls`. A more comprehensive approach to safety would be to consider *all* traces leading to unsafe behaviour and then relate them to the specified behaviour in order to construct a fault-tolerant specification. Also, we did not attempt to show, in a formal way, that our specification of the unsafe traces was a most general description. In the state based approach, it was much easier to see that we have a most general description of the unsafe behaviour, since it is captured succintly in one (structured) event. This example offers a good illustration of how the specification style can affect the ease and techniques of verification.

A further feature of this study is that the disable operator prevented a seemingly simple extension from a specification without values to one with values, since values cannot be passed over the disable operator. This suggests that this operator should be introduced into a specification with great care, and that further properties of `[>`, w.r.t. the various congruences and equivalences, should be studied (e.g. under what conditions does `P1 [> P1 = P` hold?). We have also demonstrated the need, sometimes, to use "local" laws: laws which do not hold for all processes, in general, but do hold for the ones under inspection.

7 Discussion and Future Work

We found that the majority of verification requirements were not generic properties, i.e. absence of deadlock or livelock, but properties specific to the application. Of course those properties may involve an instance of a generic property. Given this observation, it is important to note that although verification is a highly formal, rigorous activity, it is only as useful as the specifier/designer's intuition about the crucial features of the system. This means that there is also

[3]Unfortunately, neither formal methods nor good software engineering practice were employed by the designers of the Therac-25 software, which did permit fatal radiation overdoses.

much informal reasoning about what is really going on in the system and many failed proofs, relating to incorrectly formulated properties or to errors in the specification, before we obtain our final verification result.

Moreover, failed proofs are often more useful than successful ones in that they can reveal faults in the specifications. One is therefore led to ask what do final verification proofs reveal? — only correctness with respect to the particular properties which we had the foresight to consider important.

Equivalence and satisfaction requirements have traditionally been seen as the main focus for verification; and when (at least) two descriptions of the system are given, it seems obvious to try to relate them using a behavioural relation. However, since most of these relations are undecidable, the task can never be completely automated. Moreover, a good deal of effort has to be put into understanding the difficult theory underpinning the various equivalences and into determining how they relate to the actual verification requirements of that system. The issues are considerably more complex for LOTOS because of the interconnections between the data type and process parts.

Detailed understanding of the underlying theory is also important when a verification process fails; one must be able to distinguish between failure because a property does not hold, and insufficient theory to to demonstrate that it holds.

On the other hand, given a property which is easily formulated in terms of traces, property testing is a relatively easy and effective technique, even though it too is underpinned by the more complex theories of the bisimulation and testing relations. Of course property testing does not provide a complete solution, since not all properties can be described in this way.

A method of expressing more abstract properties of a system is to use a temporal logic. Although there is some use of logic for Basic LOTOS, for example [4] and [14], the problem of adding data types has not been fully addressed. (We are aware of only one reference to Full LOTOS and logic, namely [8].) To this end, we are currently developing a new semantics for Full LOTOS, which permits open processes (i.e. processes abstracted over data), and an associated temporal logic [16].

We did not address here in any depth how specification styles affect the verification tasks, mainly due to the lack of space. In [24] and [15] more examples of Full LOTOS specifications are given; these demonstrate how the verification process can become much harder with the introduction of data types.

We are continuing to explore these problems through further case studies using LOTOS including:

- the design of side-stick controllers in fly-by-wire aircraft [25];
- the implementation of a general semaphore by three binary semaphores;
- the equivalence of three specifications of the infamous stack, written in entirely different specification styles, including very different process/data boundaries;
- a simulation of a railway network including single and double track, and full four aspect signalling.

8 Conclusions

Our major conclusion from these studies is that there is not one notion, or concept, of verification for LOTOS, nor is there one equivalence/relation which captures all the verification requirements. Moreover, there are numerous techniques available for discharging the requirements.

Formal verification remains an extremely difficult task, with few models to follow. A great deal of time, and experience, is required to determine just *what* is to be verified, and then *how* it is to be verified.

Validation methods generally have better tool support; this means there is a quick gain in confidence of correctness. Perhaps this is enough, since verification typically takes longer, requires more specialist knowledge, and often fails to show what one set out to show in the beginning. However, we do believe that formal verification has a particularly vital role with reference to high integrity systems. In pursuit of evidence for this conviction, we presented some of the specifications (from the second case study) to colleagues for informal verification; in nearly all cases, the unsafe behaviour was not detected. Thus, some formal verification is clearly necessary when considering applications and problems of this nature.

To increase the uptake of verification methods, more guidance for verifiers, in the way of more published in-depth case studies and guidelines for carrying out verification, must be provided. This document is an initial attempt to fulfill that need.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–76. Elsevier Science Publishers B.V. (North-Holland), 1989.

[2] R. Booth. An Evaluation of the LCF Theorem Prover using LOTOS. In S. Vuong, editor, *Formal Description Techniques, II*, pages 83–100. Elsevier Science Publishers B.V. (North-Holland), 1989.

[3] R. Cleveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 24–37. Springer-Verlag, 1989.

[4] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7), 1993.

[5] R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[6] R. De Nicola, P. Inverardi, and M. Nesi. Equational Reasoning about LOTOS Specifications: A Rewriting Approach. In *Proceedings of 6th International Workshop on Software Specification and Design*, pages 148–155. IEEE Press, 1991.

[7] J-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In L.A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering, ICSE 14*, 1992.

[8] B. Ghribi and L. Logrippo. A Valid Environment for LOTOS. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Specification, Testing, and Verification, XIII*, pages 93–108. Elsevier Science Publishers B.V. (North-Holland), 1993.

[9] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[10] International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[11] J. Jacky. Safety-Critical Computing: Hazards, Practices, Standards and Regulations. In C. Dunlop and R. Kling, editors, *Computerization and Controversy*, pages 612–631. Academic Press, 1991.

[12] D. Kapur and H. Zhang. *RRL : Rewrite Rule Laboratory User's Manual*, 1987. Revised May 1989. Available by anonymous ftp from `herky.cs.uiowa.edu`.

[13] C. Kirkwood. Automating (Specification $\equiv$ Implementation) using Equational Reasoning and LOTOS. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development*, LNCS 668, pages 544–558, 1993.

[14] C. Kirkwood. Using Temporal Logic to Specify the Behaviour of Basic LOTOS Processes. Draft manuscript, 1994.

[15] C. Kirkwood. *Verification of LOTOS Specifications using Term Rewriting Techniques.* PhD thesis, University of Glasgow, 1994.

[16] C. Kirkwood and M. Thomas. A New Semantics and a Modal Logic for Full LOTOS. Draft manuscript, 1994.

[17] H. Lin. PAM : A Process Algebra Manipulator. In K.G. Larsen and A. Skou, editors, *Proceedings of CAV 91*, LNCS 575, pages 136–146, 1992.

[18] H. Lin. A Verification Tool for Value-Passing Processes. In *Protocol Specification, Testing and Verification XIII*, 1993. Also available as a University of Sussex Computer Science Technical report, number 93:08.

[19] LITE User Manual. Technical Report Lo/WP2 /N0034/V08, The LOTOSPHERE Esprit Project, 1992. LOTOSPHERE information disseminated by J. Lagemaat, email `lagemaat@cs.utwente.nl`.

[20] E. Madelaine and D. Vergamini. Auto: A verification tool for distributed systems using reduction of finite automata networks. In S. Vuong, editor, *Formal Description Techniques, II*, pages 61–66. Elsevier Science Publishers B.V. (North-Holland), 1989.

[21] R. Milner. *A Calculus of Communicating Systems.* LNCS 92. Springer-Verlag, 1980.

[22] J. Quemada, S. Pavón, and A. Fernandez. Transforming LOTOS specifications with LOLA - The Parameterised Expansion. In K. Turner, editor, *Formal Description Techniques, I.* Elsevier Science Publishers B.V. (North-Holland), 1988.

[23] M. Thomas. A Proof of Incorrectness using the LP Theorem Prover: The Editing Problem in the Therac-25. *High Integrity Systems Journal*, 1(1):35–48, 1994.

[24] M. Thomas. The Story of the Therac-25 in LOTOS. *High Integrity Systems Journal*, 1(1):3–15, 1994.

[25] M. Thomas and B. Ormsby. On the Design of Side-Stick Controllers in Fly-by-Wire Aircraft. *A.C.M. Applied Computing Review*, 2(1):15–20, Spring 1994.

[26] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.