Automating (Specification $\equiv$ Implementation) using Equational Reasoning and LOTOS

Carron Kirkwood*

Department of Computing Science, University of Glasgow
email: carron@dcs.glasgow.ac.uk

Abstract. We explore some of the problems of verification by trying to prove that some sort of relationship holds between a given specification and implementation. We are particularly interested in the decisions taken in the process of establishing and formalising the verification requirements and of automating the proof. Despite the apparent simplicity of the original problem, the verification is non-trivial.
The example chosen is an abstraction of a real communications problem. We use the formal description technique LOTOS [8] for specification and implementation, and equational reasoning, automated by the RRL term rewriting system [9], for the proof.

1 Introduction

The last few years has seen an increase in the use of formal methods in the design and analysis of computer systems. This has many benefits; one of which is being able to verify that certain properties hold of a system (or not, as the case may be). However, although formal methods are popular for specification, formal verification has not been taken up to the same extent, resulting in the situation where formal methods are used for specification but the implemented systems are tested in the conventional way. Three possible explanations for this are: 1) verification techniques are not as well understood as testing techniques, 2) there is little tool support for verification (making it less appealing), and 3) it is not always straightforward to express the properties to be verified. Our long term aim is to contribute to each of these areas by developing verification methods specifically for systems described using the formal description technique LOTOS [8]. In order to gain a better understanding of the problems of verification we undertook the study of the verification of the small communications problem presented in this paper.

A common problem in system development is showing that some sort of relationship holds between a given specification and implementation, i.e. the implementation *satisfies* the specification. The problem is compounded if, as here, the implementation is not formally derived from the specification. In the course of the verification we explore various ways of expressing the property to

* Funded by SERC grant gr/f 35371/ 4/1/1477, Verification Techniques for LOTOS.

be proved and consider several approaches to the proof. We also try to automate the proofs required (by tailoring a general purpose theorem prover).

The example is presented in section 2: an informal overview of the whole system is given, followed by formal and informal descriptions of the specification and implementation of the system. The formal descriptions given here are written in Basic LOTOS [8]. LOTOS was chosen because of its status as an international standard. Section 3 is concerned with a preliminary discussion of the interpretation of the verification requirements, and possible approaches to the proof that these are satisfied. The details are formalised in section 3.1. The process of automating these proofs, including the system used (the term rewriting system RRL [9]), is described in section 4.

Section 5 tells how initially we failed to meet the verifications requirements. In fact, we could show that the implementation did not satisfy the specification. Close examination of the proofs resulted in a deeper understanding of the requirements and the development of a different approach to the proof. The new approach hinges on adding some extra information in a modular way to the specification; we did this by adopting the constraint oriented style of specification [14]. This allowed the proof to be successfully completed. The new approach and the resulting specification are presented in section 5.3.

We recognise that the example as it stands is simple, so possible extensions to the case study are discussed in section 6. In section 7 we review our experience with LOTOS and RRL, making suggestions for improvements. Finally, we give our conclusions and ideas for further work arising from this study.

2 The Example

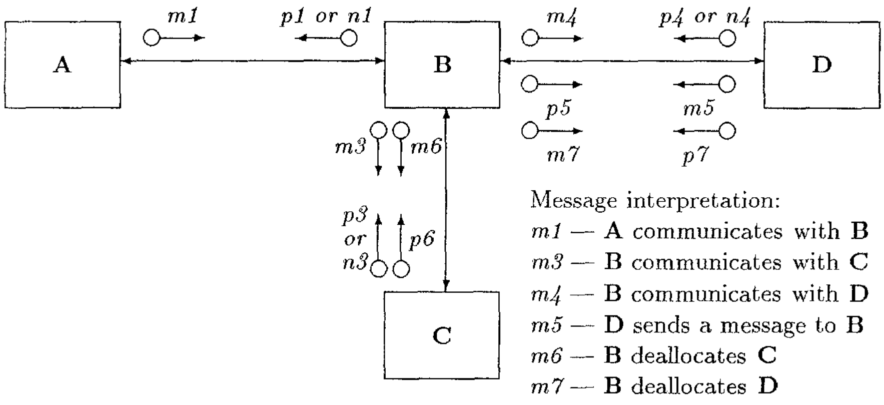
2.1 Informal Overview of the System

The example is an abstraction of a real communications problem involving four communicating processes at OSI Session level. It was first investigated as a case study for the "Verification Techniques for LOTOS" project.

There are four communicating entities: **A**, **B**, **C** and **D**, shown in figure 1. In the diagram, a box represents an entity, and a ○→ represents a message *mx* (sent in the direction of the arrow). The meaning of the *mx* are also given in figure 1. Messages of the form *px* or *nx* (where *x* is a number) are positive and negative acknowledgements, respectively, to the corresponding *mx* messages.[2]

A requests a service from **B**; in order to satisfy that service, **B** must communicate with **C** and **D**. **B** has an internal timer which "times out" if **D** does not reply to its communication within a previously set time limit. **B** must send deallocation messages to **C** and **D** when they are no longer required.

[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* "Service terminated".



Message interpretation:
m1 — **A** communicates with **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**

Fig. 1. The Processes and their messages

The original example [5] was supplied by Jeremy Dick, who worked for RACAL at the time. For reasons of security, we were given only the abstract description of the system as above; no indication of the real content or meaning of the messages was given. To help illuminate the system, we invented a possible interpretation of our own. This 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.

Two possible descriptions of the system are given below: firstly, a group of protocols which make up the specification, and secondly a group of processes which make up the implementation. Note that inconsistencies may be found between the way the specification describes something and the way the implementation describes the same thing. This is because the implementation was not formally derived from the specification and one of the problems considered here is that of trying to reconcile any differences between the two.

The informal introductions to the specification and implementation are followed by their formal descriptions, given in Basic LOTOS. Only the process algebra part of LOTOS is used as no data types are required (see section 6 for extensions involving data types). The reader is assumed to be familiar with LOTOS or a related process algebra such as CCS [12]. The language constructs used for the descriptions are: **exit** denoting successful termination, ; denoting action sequencing, and [] denoting choice between two process expressions. Note that in the remainder of this document, the term *processes* will be used to refer to the implementation part of the example. LOTOS processes will be referred to as such, or as process expressions.

In these descriptions the simplifying assumptions that the carrier is faithful and no messages or acknowledgements are lost or corrupted are made.

2.2 Protocols

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. $P1$ describes the interface between **A** and **B**, ignoring **C** and **D**.

$P1$: **A** sends *m1* to **B**, which must be acknowledged by *p1* or *n1*.
$P2$: **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*.
$P3$: **B** sends *m4* to **D** which must be acknowledged by *p4* or *n4*. After *p4*, **D** may or may not send *m5* to **B**. *m5* must be acknowledged by *p5*. Also after *p4*, **B** may or may not send *m7* to **D**. *m7* is acknowledged by *p7*. Receipt of *m7* removes the capability to send *m5*.

```
process P1 := m1; (n1; exit [] p1; exit) endproc
process P2 := m3; (n3; exit [] p3; (exit [] m6; p6; exit)) endproc
process P3 := exit [] m4; ( n4; exit
                          [] p4; (  exit [] m7; p7; exit
                                  [] m5; p5; (exit [] m7; p7; exit))) endproc
```

Note that in a real system the protocols, and also the processes, would probably be described recursively, i.e. cycling over the same behaviour forever. This is ignored at the moment, the simpler finite case being dealt with first. Having finite LOTOS processes instead of recursive ones results in the initial **exit** branch of $P3$; this expresses the notion that the full $P3$ protocol is not always activated.

2.3 Processes

The implementation of the system is achieved by four interacting processes.

A: **A** sends *m1* to **B**. After this message **B** sends either *p1* or *n1* to **A**, indicating success or failure of the transaction respectively.
C: **C** receives *m3* from **B** to which it replies either *p3* or *n3*. If *p3* is sent then **C** expects an *m6* deallocation message, to which it replies *p6*.
D: **D** receives *m4* from **B**, to which it replies *p4*, and the transaction continues, or *n4*, and the transaction terminates. After *p4*, **D** sends *m5* to **B**, expecting *p5* in response, then deallocation by *m7*, to which **D** replies *p7*. The transaction may be terminated if **D** receives *m7* before it sends *m5*, i.e. the timer has expired causing **B** to terminate the transaction.
B: In a successful execution **B** receives *m1* from **A**, allocates **C** with *m3 p3* and **D** with *m4 p4*, then sets a timer as **D** must send *m5* within some time limit. When *m5* arrives the timer is cancelled and **B** replies with *p5*. **C** and **D** are deallocated by *m6 p6* and *m7 p7* respectively. Finally **B** signals 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 **B** replies *n1* to **A**. Deallocation of **C** and **D** occurs if and only if they originally agreed to participate in the transaction, i.e. if they sent *p3* and *p4* respectively.

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

Now we have the formal descriptions of the specification and the implementation we wish to verify that the implementation is correct with respect to the specification. The next section examines how that correctness can be evaluated.

3 Verification of the Example

The statement to be verified can be expressed as: does the implementation (the processes **A**, **B**, **C** and **D**) satisfy the specification (the protocols $P1$, $P2$ and $P3$)? The terms used above are deliberately vague, allowing exploration of different possible interpretations, discussed informally here and formally in section 3.1. Three terms have yet to be defined: "specification", "implementation" and "satisfies". The meaning of the first two ought to be straightforward since the protocols and the processes have been given, but these are only the bones of the description. For example, the protocols form the specification, but how they should be combined, or indeed *if* they should be combined, is not mentioned. The same is true of the processes and the implementation.

Suppose the protocols are to be combined to form the specification and the processes combined to form the implementation. The statement then becomes:

$$(\mathrm{A} \mid \mathrm{B} \mid \mathrm{C} \mid \mathrm{D}) \text{ satisfies } (P1 \mid P2 \mid P3) \tag{1}$$

where the "|" operator denotes "combined with". Note that each instance of "|" may be replaced by a slightly different operator when the problem is made concrete. For example, the combinator used in A | B may be different from that used in C | D, or $P1$ | $P2$. These things will be formalised in section 3.1.

An alternative approach exploits the modular way in which the system has been defined: each facet of the interaction can be examined separately.

$$(\mathrm{A} \mid \mathrm{B}) \text{ satisfies } P1 \tag{2}$$
$$(\mathrm{C} \mid \mathrm{B}) \text{ satisfies } P2 \tag{3}$$
$$(\mathrm{D} \mid \mathrm{B}) \text{ satisfies } P3 \tag{4}$$

As they stand, these equations are not quite correct since the language, i.e. the events, 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 $P1$. Either these events will have to be hidden, or the interpretation of "satisfies" must take account of the extra events.

Since equations (2), (3) and (4) each yield a boolean, the results can be combined using a boolean operator. We choose & since we want all facets of the interaction to be satisfied, but we must also be sure that satisfying all equations is the same as satisfying the system as a whole. In this case, since $P1, P2$ and $P3$ are all concerned with distinct facets of the communication of the system, it seems likely that the verification can safely be split into parts. Note that this really depends on choosing the right methods of splitting up the system, hiding unimportant events making individual proofs, and recombining the results.

3.1 Formalising the Verification Requirements

We should now give the formal interpretation of "|", the hiding of events, and "satisfies", again using LOTOS.

The general parallelism operator of LOTOS is used to combine both processes and protocols. This operator takes two process expressions and a list of events specifying the events on which the process expressions must synchronise. Variation of the events in this list give the subtly different combinations of the components of the system (as mentioned above). The syntax for this operator is $P|[eventlist]|Q$, where P and Q are process expressions.

The **hide** operator is used to restrict the processes to protocol events only. This operator takes a process expression and a list of events to be hidden. Hidden events are treated like the internal event **i**; they are unobservable and occur instantaneously. The syntax is **hide** *eventlist* **in** P.

There are many different possible interpretations for the "satisfies" relation. The particular sort of relation (e.g. equivalence or preorder) will depend on the sort of decisions made in the step between the specification and the implementation. For example, the implementation may resolve some choices left open in the specification, or it may add some information about how to perform a particular task, or it may substitute one method of performing a task for another. Some steps preserve the observable behaviour of the specification while others do not, therefore in some cases an equivalence or a congruence relation is appropriate, i.e. where the processes must implement *all* the alternatives set out in the protocols, while in other cases a preorder relation will suffice. Since in this example the implementation was not derived directly from the specification we cannot say anything about the sort of steps used, so we must examine a variety of LOTOS relations in more detail to determine which are most suitable. Starting with the strongest (i.e. makes fewer identifications):

Strong Bisimulation Equivalence This relation requires all events, including the internal event, to be matched exactly. Given the use of the **hide** operator which converts hidden events into the internal event, an equivalence which ignores these is required.

Weak Bisimulation Equivalence This relation requires all events except the internal event to be matched exactly. The internal event is given its status as a special, unobservable, event and can be matched by zero or more internal events. This relation does not preserve the substitution property in all LOTOS contexts, i.e. two process expressions may be weak bisimulation equivalent, but their internal structure could cause them to behave differently when in combination with other process expressions.

Weak Bisimulation Congruence This is the largest congruent relation contained in weak bisimulation equivalence. Most, but not necessarily all, of the internal events created by the use of **hide** can be removed by weak bisimulation congruence laws.

Testing Relations The basic testing relation for LOTOS is a preorder called **red**. B_1 **red** B_2 says that B_1 is a deterministic reduction of B_2, which may be interpreted as B_1 "implements" B_2. The equivalence generated by this preorder is not a congruence. To obtain congruence the **cred** relation, the largest congruent sub-relation of **red**, must be used.

Trace Equivalence This says that two process expressions are equivalent if their trace sets, i.e. their sequences of actions, are the same. This does not give a satisfactory interpretation of "satisfies" since deadlock properties are not preserved, i.e. two process expressions may be trace equivalent but one may deadlock after a trace s while the other does not.

In summary, trace equivalence is rejected because too many identifications are made, strong bisimulation equivalence because too few are made. Our system will probably have to interact with other systems, so it is important that it behaves in the same way in all contexts. This leads us to reject weak bisimulation equivalence and testing equivalence, leaving weak bisimulation congruence and testing congruence. We also have the testing preorders. Since there are no other criteria to take into account, any of these relations will suffice as an interpretation of "satisfies".

The next section describes the method and software used to automate the proofs of equivalence.

4 Proof: Technique and Application

Several software tools are currently available which can determine the equivalence/ordering of two process expressions, e.g. the Concurrency Workbench [3] and TAV [6]. These systems generate a finite state machine to represent the processes and apply some sophisticated algorithms to decide their relationship. This approach suffers from the state explosion problem and cannot handle infinite systems. Although the current example is unlikely to cause state explosion and is finite, our aim is to develop methods which may be applicable to other examples. For this reason we use *equational reasoning*, i.e. symbolic manipulation of terms, thus avoiding any special representations. This approach is also successfully used in [4, 11, 13].

Using term rewriting to implement equational reasoning, two terms are proved equivalent by reducing them to their normal forms and comparing these syntactically. If the normal forms are the same then the original terms are equivalent, otherwise not. The same technique is used to prove a preorder between two terms. This procedure relies on having a *complete* (i.e. confluent and terminating) set of rules (giving unique normal forms).

A brief description of the rule sets used for this case study follows; a more detailed presentation can be found in [10]. We split the rules according to their function, giving three sets:

1. Rules derived from the weak bisimulation congruence laws, including the **hide** expansion law, which are given in appendix B.2.2 of the LOTOS standard [8]. These rules remove instances of the **hide** operator (by converting events to be hidden into the internal event, **i**) and reduce terms with respect to weak bisimulation congruence.
2. Rules which "implement" the expansion law for parallelism, also found in appendix B.2.2 of the LOTOS standard. These rules remove instances of the parallel operator, converting the terms into equivalent ones which use only sequencing and choice operators.
3. Rules corresponding to the **cred** laws, which can be found in appendix B.3.2 of the LOTOS standard. Set 3, when used, is always an addition to Set 1, since one of the laws for **cred** states that all the laws of weak bisimulation congruence are also true for **cred**. These rules allow terms to be reduced with respect to the **cred** refinement relation.

Sets 1 and 2 also contain rules for basic data types, e.g. lists, sets etc.

The RRL term rewriting system [9] was used to perform the proofs. RRL features include Knuth-Bendix completion and proof by rewriting. RRL also handles rewriting and completion modulo associative-commutative operators. This was the main reason for choosing this system over others currently available.

The rule sets given above are not confluent and terminating, which means we have a semi-decision procedure for equivalence/ordering of LOTOS processes, i.e. normal forms are not unique. This means that if two terms can be shown to be equivalent/ordered by our rules, then they are equivalent/ordered in the LOTOS semantics, but if two terms cannot be shown to be equivalent/ordered by our rules, then they may or may not be equivalent/ordered in the semantics. No special techniques to cope with non-confluent rule sets are adopted; if two terms cannot be shown equivalent by RRL the proof is completed by hand.

The next section contains details of the proofs which were attempted, and some discussion of why some of those proofs failed.

5 Verification Proofs

In section 3 two possible approaches to proving that the implementation of the system satisfies its specification were discussed. One involved splitting the proof into three parts corresponding to the three protocols in the specification, while the other dealt with the system as a whole. The results of these approaches, successes and failures, in trying to prove automatically that the specification is satisfied by the implementation are given below.

5.1 Splitting the Proofs into Three Sections

Since each protocol describes the interface between just two of the processes, the idea of proving each interface is correct and deducing from that the correctness of the whole system is very appealing. Unfortunately, this approach turned out to be unsuccessful. Proofs about the relationship between the specification and the implementation could be completed, but the results were not strong enough to satisfy the correctness requirement. However, examining these proofs (successful or otherwise) helps illuminate the reasons for the failure of this approach. In the following, $\equiv_{wbc}$ denotes weak bisimulation congruence.

Weak Bisimulation Congruence We tried to prove the following equations.

$$P1 \equiv_{wbc} \textbf{hide } \mathit{CDevents} \textbf{ in } (A \;|[m1, p1, n1]|\; B) \tag{5}$$

$$P2 \equiv_{wbc} \textbf{hide } \mathit{ADevents} \textbf{ in } (C \;|[m3, p3, n3, m6, p6]|\; B) \tag{6}$$

$$P3 \equiv_{wbc} \textbf{hide } \mathit{ACevents} \textbf{ in } (D \;|[m4, p4, n4, m5, p5, m7, p7]|\; B) \tag{7}$$

where $P1, P2, P3$ and A, B, C, D are as defined in section 2 and the event lists to be hidden are:

$$\begin{aligned}\mathit{CDevents} &= [m3,\ p3,\ n3,\ m4,\ p4,\ n4,\ m5,\ p5,\ m6,\ p6,\ m7,\ p7\,]\\ \mathit{ADevents} &= [m1,\ p1,\ n1,\ m4,\ p4,\ n4,\ m5,\ p5,\ m7,\ p7\,]\\ \mathit{ACevents} &= [m1,\ p1,\ n1,\ m3,\ p3,\ n3,\ m6,\ p6\,]\end{aligned}$$

Equations (5), (6) and (7) cannot be proved using RRL. The proof was completed by hand to show that the equations do not hold in the LOTOS semantics. The full proof is not presented here, but may be found, together with the other proofs from this study, in appendix A of [10]. The next step was to try to show the equations hold for a weaker relation.

Testing Relations Taking the left and right hand sides of the equations as above, we substituted the **cred** relation for $\equiv_{wbc}$ and tried to show the new equations held either left-to-right or vice-versa (if they hold in both directions we get testing congruence).

$$P1 \textbf{ cred } \textbf{hide } \mathit{CDevents} \textbf{ in } (A \;|[m1, p1, n1]|\; B) \tag{8}$$

$$P2 \textbf{ cred } \textbf{hide } \mathit{ADevents} \textbf{ in } (C \;|[m3, p3, n3, m6, p6]|\; B) \tag{9}$$

$$P3 \textbf{ cred } \textbf{hide } \mathit{ACevents} \textbf{ in } (D \;|[m4, p4, n4, m5, p5, m7, p7]|\; B) \tag{10}$$

The proofs of equations (8) and (9) can be completed by RRL in the left-to-right direction, which means that the protocols are a deterministic reduction of the processes, i.e. the processes may have some nondeterminism not present in the protocols (which is not what may normally be expected). None of the equations holds in the right-to-left direction; equation (10) holds in neither direction.

At this point it appeared that trying to prove the verification requirement was satisfied was hopeless. However, we strongly believed that the processes were a valid implementation of the system and that therefore it was the approach to the proof which was incorrect. The strategy of splitting the proof into three parts did not work, or rather, some proofs could be completed, but they were not sufficient to satisfy the verification requirement. In particular, it seemed that the hiding of events caused the failure of the proofs by spotlighting apparently non-deterministic choices in the process expressions. These choices are not really non-deterministic; the choices are determined by factors in the other processes. For example, the right-hand process expression in equation (5) makes a non-deterministic choice between replying *p1* and replying *n1*. However, we know that this choice really depends on the receipt of *m5* (which is hidden). This problem affects proofs using weak bisimulation congruence or testing congruence. We observe that we are not the only ones to encounter this problem; the same phenomenon also causes problems for other authors, e.g. [1, 2].

We then went on to try the other approach to the proof, where the system is considered as a whole, thus avoiding the use of **hide**.

5.2 Proving the System as a Whole

No relationships between the processes all combined and the protocols all combined could be demonstrated because although the processes can be combined using parallelism, there is no meaningful way in which to combine the protocols.

There are two operators which could be used to combine the protocols. These are sequential composition of process expressions and interleaving (general parallelism synchronising on no events) since the protocols have no events in common. The former cannot be used because, for example, the events of $P1$ do not all precede the events of $P2$, and the latter cannot be used because the protocols contain no information about the relative ordering of events in different protocols. Interleaving results in a process expression which has a large number of traces which are meaningless in our example, given our informal description.

The reason the protocols cannot be combined in a way that reflects our intuition is that there is some information missing from the specification, leaving too large a step between it and the implementation. This may have occurred because the implementation was not derived from the specification, but it is also generally true that verification becomes harder as the distance between the specification and the implementation increases.

The missing information, which is implicit in the implementation, includes details of a timer, deallocation and what constitutes success or failure of the transaction. In the specification there is no information about any of these things.

Our solution was to add the information in the form of *constraints* giving a successful approach to the problem.

5.3 Adding Constraints to the Example

The constraint oriented style of specification [14] relies on the way in which the LOTOS general parallelism operator handles multi-way synchronisation, i.e. synchronisation between two or more actions. For example, if three LOTOS processes, all of which use the action a, are combined using general parallelism, synchronising on a, then all three must perform that event at the same time. This means that different process expressions can specify different aspects of a behaviour, interacting to give the whole specification. The effect is similar to using conjunction in a logical specification; each part must be satisfied for the whole to be satisfied.

Using this method, more LOTOS processes are defined which express other aspects of the specification not included in the protocols. These include a timer in **B** to determine how long it should wait for **D** to send the *m5* message, compulsory deallocation of **C** and **D**, ordering of events as mentioned in the informal overview of the system, and conditions dictating success or failure of the transaction as a whole. The following constraints are added to the specification:

```
Timer Constraints
process timer     := exit [] set; ( tcancel; exit [] timeout; exit) endproc
process timer_on  := exit [] p4; set; exit endproc
process timer_off := exit [] set; ( m5; tcancel; p5; m7; exit
                                  [] timeout; m7; exit) endproc
```

```
Deallocation Constraints
process dealloc_C := p3; m6; p6; exit [] n3; exit endproc
process dealloc_D := exit [] m4; (p4; m7; p7; exit [] n4; exit) endproc
```

```
Success and Failure
process system :=  m5; p1; exit
                [] n3; n1; exit
                [] n4; n1; exit
                [] timeout; n1; exit endproc
```

Ordering Constraints

```
process order13 := m1; m3; ( n3; n1; exit
                             [] p3; (n1; exit [] p1; exit)) endproc

process order34 :=  m3; ( n3; n1; exit
                        [] p3; m4 ( n4; n1; exit
                                  [] p4; (n1; exit [] p1; exit))) endproc

process order457 :=  n3; n1; exit
                   [] m4; ( n4; n1; exit
                          [] p4; ( m5; p5; m7; p7; p1; exit
                                 [] timeout; m7; p7; n1; exit)) endproc

process order56 :=  n3; n1; exit
                  [] n4; m6; p6; n1; exit
                  [] timeout; m6; p6; n1; exit
                  [] m5; p5; m6; p6; p1; exit endproc

process order67 :=  n3; n1; exit
                  [] p3; ( n4; m6; p6; n1; exit
                         [] p4; m6; p6; m7; p7; (n1; exit
                                                [] p1; exit)) endproc
```

As with the descriptions of the protocols and the processes, some **exit** branches are introduced to express the notion that a constraint may not be activated.

Given these constraints, the equation to be proved by RRL becomes:

```
(((P1 |[p1, n1]| system) |[m1, p1, n1, n3]| order13)
    |[p1, n1, m3, p3, n3, n4, m5, timeout]|
((((P2 |[p3, n3, m6, p6]| dealloc_C)
    |[m3, p3, n3, m6, p6]| (order34 |[p3, n3, p4, n4]| order67))
    |[p1, n1, n3, m4, p4, n4, m7, p7]|
(((P3 |[m4, p4, n4, m7, p7]| dealloc_D) |[m4, p4, n4, m5, p5, m7, p7]| order457)
    |[p4, m5, p5, m7, timeout]|
((timer |[set]| timer_on) |[set, timeout, tcancel]| timer_off))
    |[m5, p5, m6, p6, timeout]| order56))
≡wbc
(((A |[m1, p1, n1]| B) |[m3, p3, n3, m6, p6]| C)
    |[m4, p4, n4, m5, p5, m7, p7]| D)
```

Although the order in which the process expressions are combined does not matter in the semantics, we add as much information as possible to each protocol before combining it with the others. This is because in performing the proof, our system can only deal with one parallel statement at a time, which means that the proof has to be built up gradually from small units. Adding as much information as early as possible helps to cut down the size of the intermediate terms in the proof.

This equation can be proved to hold by RRL. This is an adequate proof of correctness since it means not only that the processes have the same observable behaviour as the protocols, but also that they behave in the same way in all contexts. The proof requires only the rules from set 2; no other rule sets are required. As the specification and implementation use no internal actions we may also deduce that the above equation holds for strong equivalence as well as for weak bisimulation congruence.

6 Extensions to the Example

The example as considered in this document is very simple; there are a number of ways in which it can be made more complex.

- A useful extension would be to add an "abort" message, call it *m2*. **A** can abort the service at any time by sending *m2* to **B**, which should clean up by deallocating any resources held and then replying to **A** with *p2*.
  In LOTOS it would be simple to add *m2 p2* as an abort sequence using the operator [> , which allows one process to take control from another. However, in this example the system is more complicated, requiring varying sequences of actions between *m2* and *p2*, depending on the events which occurred before *m2*. The original solution could not be easily extended to include this new behaviour. This could indicate a fault in the solution to the original problem, perhaps it is not modular enough, or it could be that there is no simple, elegant way to extend the solution. Another possibility is that it is the form of this particular modification which is causing the problem, see section 7.
- Data types could be added to the messages, e.g. the login name and password of the informal interpretation of the example.
- The most obvious extension would be to introduce recursion. Work is in progress on the addition of recursion to the example. So far the proofs have not become any more complex, however, we felt that the example was simpler to present without recursion.

7 Review of the Tools Used

Although some degree of success was achieved in the case study, there were also many problems, not all of which arose from the example itself; some were due to either LOTOS or RRL. For example, in RRL we would have liked more control over the application of rules and more feedback on the rules RRL used in a reduction. The suggested improvements to LOTOS are given below.

LOTOS was not always suitable to describe the example. A major problem was revealed when attempting to extend the original problem to include the abort message. The [> operator was unsuitable for this purpose because it does not allow the abort sequence to be dependent on the state of execution before the abort message. One way round this is to write each abort possibility into the LOTOS processes as choice branches, which makes the specification rather cumbersome. What is required is an operator which allows the abort sequence to be flexible, perhaps allowing parameters to be passed from the interrupted to the interrupting process (as with sequential composition of processes).

Another feature which would have been useful was an operator to "wrap-up" several actions and make them behave as a single action, i.e. like a critical section in a mutual exclusion problem. For example, we wanted to be able to combine two process expressions using interleaving, but to have a section in one of the expressions which, once it had begun, had to finish without interleaving with the other process until after the last action was completed. This could have been solved with a mutual exclusion algorithm, but a language construct to do this would be more convenient. This problem is also identified in [7].

8 Conclusions and Further Work

After much experimentation, we succeeded in showing that the verification requirements of a small communications protocol were indeed satisfied. It must be noted that the given specification was not sufficient for our purposes and had to have more information added to enable the proofs to be carried out. The new information was added in a modular way however, and the text of the original specification was unaltered, although it must be admitted that the size of the new specification is greatly increased. Possible extensions to the problem are provided in section 6. It is hoped that these can also be made in a modular way.

In some ways, the initial failure to meet the verification requirements was perhaps more fruitful than the final proof, because we were able to identify problems in the verification process which need to be further researched. For example, the effect of **hide** on our proofs, introducing non-determinism and thus causing failure, and the difficulty in choosing between the different relations. There are many more equivalences defined in the process algebra literature than presented in this document, we merely chose some of the most well-known.

Another possible line of research is that of investigating alternative formulations of the verification requirements. Other situations may require different interpretations of the statement "specification satisfies implementation", e.g. one alternative is "prove the system satisfies certain temporal formulae", giving another way of looking at what constitutes specification and implementation.

The main result of our work on this case study is the demonstration that verification, even of such a small and simple system, is a difficult process; one which is full of opportunities to take the wrong decision and thereby to fail to prove the correctness of the system under investigation. In this study we only arrived at a successful proof because we persevered, having a strong belief that such a proof must exist. In more complex examples it would perhaps be less easy to hold such a belief.

Acknowledgements We would like to thank the members of the ERIL project group, in particular our colleagues at Glasgow, Muffy Thomas and Phil Watson, for many stimulating discussions on the issues and problems raised by this case study. Thanks also to the referees for their helpful comments, and to Deepak Kapur for supplying RRL.

References

1. J. Baillie. A CCS case study: a safety-critical system. *Software Engineering Journal*, pages 159–167, July 1991.
2. G. Bruns and S. Anderson. The Formalization and Analysis of a Communications Protocol. Technical Report ECS-LFCS-91-137, LFCS, University of Edinburgh, 1991.
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, P. Inverardi, and M. Nesi. Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Expressions. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 54–67, 1989.
5. A.J.J. Dick. A Case Study for the ERIL Project. Private communication, 1990.
6. J.C. Godskesen, K.G. Larsen, and M. Zeeberg. TAV (Tools for Automatic Verification): Users Manual. Technical report, Aalborg University, 1989.
7. R. Gotzhein. Specifying Abstract Data Types with LOTOS. In B. Sarikaya and G.V. Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*, pages 15–26. Elsevier Science Publishers B.V. (North-Holland), 1987.
8. International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.
9. D. Kapur and H. Zhang. *RRL : Rewrite Rule Laboratory User's Manual*, 1987. Revised May 1989.
10. C. Kirkwood. A Case Study for the ERIL Project. Technical Report 1992/R4, University of Glasgow, 1992.
11. C. Kirkwood and K. Norrie. Some Experiments using Term Rewriting Techniques for Concurrency. In J. Quemada, J. Mañas, and E. Vásquez, editors, *Formal Description Techniques, III*, pages 527–530. Elsevier Science Publishers B.V. (North-Holland), 1991. Extended Abstract.
12. R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
13. M. Nesi. Mechanizing a Proof by Induction of Process Algebra Specifications in Higher Order Logic. In K.G. Larsen and A. Skou, editors, *Proceedings of CAV 91*, LNCS 575, pages 288–298, 1992.
14. 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.