Some Experiments using Term Rewriting Techniques for Concurrency

Carron Kirkwood*
University of Glasgow

Kathy Norrie†
Royal Holloway and Bedford New College, University of London

August 13, 1997

Abstract

Theorem proving using term rewriting has been thoroughly explored for equational specifications; we look at the use of term rewriting for proving theorems in the process algebras of concurrency and conduct two experiments in this area. We use the LP theorem prover for proofs about CSP, and the RRL term rewriting system for reasoning about LOTOS. The results of these experiments provide information about possible enrichments to term rewriting theory that will make it more effective in reasoning about concurrent systems.

1 Introduction

The purpose of this paper is to report on preliminary investigations into the use of term rewriting theorem proving to support verification of formal descriptions of concurrent systems. We report on two pieces of work; using the LP theorem prover [14] for equational and inductive proofs about CSP and the RRL system [24, 38], particularly Knuth-Bendix completion and equational proofs, to investigate LOTOS.

Term rewriting theorem proving is designed for reasoning with sets of equations by orienting them into rules, which are then used to simplify terms. If two terms simplify to the same term they have been proved equal. If the set of rules is confluent and terminating (complete — see section 3.3), then any two terms that are equal can be rewritten to the same term: we have a solution to the word problem. Even if the set of rules is not complete we may still have enough rules to prove the results we need. In particular we may be able to do induction proofs, since verifying the base step and induction hypothesis may only involve equational reasoning [13]. Term rewriting systems such as ERIL [10], LP [14] and RRL [38] incorporate these techniques, as well as an algorithm that attempts to find a complete set of rules equivalent to a given set of equations. Additional features include the incorporation of sorts, and

*Funded by SERC grant gr/f 35371/ 4/1/1477, Verification Techniques for LOTOS.
†Funded by SERC grant gr/f 35364/ 4/1/1477, Verification Techniques for LOTOS.

of rewriting and completion modulo a set of equations, such as the associative and commutative laws for some operator. The philosophy of the designers of these systems is that they should be easy to use, and that when they fail they fail quickly, so that the user is able to build a proof by experimenting and debugging. A full introduction to equational reasoning may be found in [22].

Specification languages such as CSP [21], CCS [29, 30], ACP [1] and LOTOS [40, 11] can be defined by equations giving the properties of the various operators involved, which indicates the use of equational reasoning. There are complete axiomatisations for each of these languages except LOTOS. The axiomatisation of ACP, in addition to being complete with respect to the model, is also confluent and terminating [1], while for CCS there are several axiomatisations, only some of which are confluent and terminating.

LOTOS incorporates abstract data types, which makes term rewriting an obvious choice for reasoning about LOTOS specifications, since term rewriting has already been widely used for reasoning about abstract data types, for example in work on the Larch shared language [19], OBJ [17] and Affirm [32].

Of course, our work is only part of a whole body of experiments in reasoning about concurrent systems, either by building specialised tools or by using existing theorem provers. For example, CSP-like systems are supported by the occam transformation system [18], which uses rewriting, and CCS by the Concurrency workbench [6], which implements several model-checking and equivalence-checking algorithms. The manipulation of CCS expressions by rewriting is examined in [7, 23] [1]. A survey of systems for the analysis of communication protocols is given in [2]. Experiments with general purpose theorem provers include using LP to verify synchronised transitions [35], and REVE [27] to investigate process algebras [31]. General purpose theorem provers have also been used for work in LOTOS [3, 12]. This work concentrates on the transition rules on the grounds that the congruence laws of appendix B.2.2 do not yield a canonical set of rewrite rules and are therefore unsuitable for automatic theorem proving by term rewriting. We show that useful results can still be obtained from a set of rewriting rules generated from a subset of the laws in appendix B.2.2.

In this paper we report on two preliminary studies being carried out as part of the SERC/IED ERIL project, which is investigating the use of Equational Reasoning for LOTOS. Section 2 gives a brief overview of the desirable features of a concurrency formalism, and comments on CSP and CCS with repect to these criteria. An introduction to the basics of term rewriting is given in section 3. In section 4 we show how the CSP laws may be expressed in the LP system. This immediately allows equational reasoning to be done, but an obvious difficulty is how to handle recursion since at first sight rewriting would seem just to unwind a recursive definition for ever. The examples we present show how LP can be used to handle this problem, and illustrate the inductive proof mechanisms with an example involving mutual recursion. Section 5 details our attempts to generate a confluent and terminating set of rules for a subset of the weak bisimulation congruence laws of LOTOS and shows how these rules can be used in equational proofs. We also look at problems caused by some of the laws and discover an unsound law in appendix B.2.2 of the LOTOS standard [40]. Section 6 gives a overview of the work carried out by De Nicola, Inverardi and Nesi [7, 23], and draws comparisons with our work. The conclusions taken from our experiments and suggestions for further work are given in section 7.

The examples we present here are all small ones, but are already at the level where hand verification is tedious and error prone. They execute immediately in LP and RRL, and give hope that the same

[1] This work is similar to the work discussed in section 5 (LOTOS and RRL); comparisons are drawn in section 6 (Related Work). Our work was completed independently.

techniques can be used on much larger examples.

2 Concurrency Formalisms and Semantics

Before we contemplate specific concurrency formalisms, we should first consider what we desire of a specification language. A formal description technique (fdt) should satisfy these basic criteria: no ambiguity, consistency, clarity, completeness, abstraction and structure (all described in [5]). These features can make a specification easier to reason about and verify. In addition to these general characteristics, we must also consider the special needs of a concurrency specification language. We may want to verify any of the following properties: deadlock freedom, livelock freedom, fairness, correct allocation/deallocation of resources, mutual exclusion etc.; our modelling of the concurrent system should facilitate specification and verification of these. The following criteria are taken from [15] and should give an idea of what sort of things we want from a concurrency formalism.

Recursion This allows us to describe repetitive behaviour patterns that may continue indefinately.

Nondeterminism We will almost certainly want nondeterminism, but do we want angelic or demonic nondeterminism? For example, take a choice between two processes, one that deadlocks on its first step, another that behaves normally. Angelic choice will never pick the deadlocking process, always the normal one. This is also known as external choice, meaning the choice is influenced by the environment. Demonic choice, however, may pick either process regardless of whether deadlock is possible. This is also known as internal choice since it depends on the process — the environment has no influence.

Communication between parallel processes This ranges from simple synchronisation between processes to the passing of complex values between processes. We also have the choice of whether these interactions occur between only two processes, or are broadcast to any process which may be able to participate.

Hiding How much of a process's actions do we want to be observed from the outside? Will we provide a mechanism for explicitly hiding certain events from some external observer?

Deadlock We must have some representation for this important concept in concurrency; the deadlocked process is as basic as zero in arithmetic.

Divergence Although we do not want processes to have this property, we must put it into our concurrency theory so that we can identify it and avoid it.

Many different approaches to concurrency have been used: CCS [29, 30], CSP [21], LOTOS [40] and ACP [1]. These can be split into two groups: those which are based on a particular model and are presented with laws which are true in that model, and those that are a calculus of rules and axioms and can be presented independently of any mathematical model — the rules are true anywhere. The importance of the mathematical model cannot be over-emphasised, for it is this firm base that provides characteristics such as clarity and consistency.

We look at two concurrency formalisms: CSP, which is based on the trace model, but which has a systematic axiomatisation of all its operators; and CCS, which is presented as a calculus of axioms and rules, but with specific refererence to the model of state transition trees. In each of the following sections we present the semantics of each formalism and the operators that implement the features of concurrency detailed above.

2.1 CSP

In CSP, a process is uniquely represented by its *alphabet*, its *failure set* and its *divergence set*. The alphabet of a process is the set of events in which the process may engage. The definition of the failure set is a little more complicated; first we must define what we mean by *trace* and *refusal*.

A trace is a sequential record of the observable behaviour of a process. Each process may have many possible traces as a result of choices in execution. To define refusal, let X be the set of events offered by the environment of a process, P. If P can deadlock on its first step when placed in this environment, then the set X is known as a refusal of P. We can now say that the failure set of a process, P, contains all pairs (s, X), where s is a trace and X is a refusal, such that P may deadlock after trace s in the environment offering events X.

The divergence set of a process is a set of traces such that, after performing any one of these traces, the process will behave chaotically, i.e. it is impossible to determine what events will occur.

Operators of CSP include: recursion, special demonic nondeterministic choice between processes in addition to deterministic choice, communication (where input/output sets of the same event synchronise i.e. multi-way synchronisation [note that synchronisation is compulsory — if it can happen it must happen]), interleaving (no interaction at all), hiding of events, and special processes to represent deadlock and livelock, STOP and CHAOS. As an example of the unique way of representing CSP processes, consider the two special processes:

$$\mathrm{STOP}_A \;\; = (A, \{<>\} \times \mathcal{P}A, \{\})$$
$$\mathrm{CHAOS}_A = (A, (A^* \times \mathcal{P}A), A^*)$$

where $<>$ denotes the empty trace,
$\mathcal{P}A$ denotes the powerset of A, and
A^* denotes all possible traces using events in A.

These equations say that the alphabet of each process is A, some arbitrary set of events; note the use of a subscript A to make the alphabet explicit in the name of the process, if this is not present the alphabet is assumed to be all events. Secondly, STOP_A can refuse to do any subset of A after the empty trace, while CHAOS_A can refuse to do any subset of A after any trace. Thirdly, STOP_A has an empty divergence set – it never diverges, while CHAOS_A can diverge after every possible trace.

The idea behind the design of CSP was to have a single model in which only observably different processes were distinct; all other processes are identified, i.e. only the essential distinctions are made. For example, processes which diverge on the first step are equated with CHAOS. This model results in a richer set of algebraic laws, allowing more flexible transformation and optimisation of CSP processes. A different approach was taken in the design of CCS.

2.2 CCS

CCS is a calculus designed to investigate different models of concurrency, where each model is defined by different equivalence between processes. The basic model is that of state transition trees. Each process is seen as a set of states, with arcs between states representing actions which move the process from one state to another. A special action is the τ action, which cannot be observed. It moves a process silently from state to state. Loops from a state to itself are allowed.

Unlike CSP, CCS has a very compact set of operators: there is only one choice operator, which is deterministic but can be forced to be nondeterministic if each alternative is prefixed by τ. There is no value passing between processes, only synchronisation. This synchronisation is two-way, but it is not compulsory as in CSP. CCS also has recursion, an operator to hide certain specified actions from an observer and a special deadlocked process.

The most significant difference between CCS and CSP, as far as we are concerned, is the wide range of equivalences defined. This means different sorts of divergent processes, for example, can be distinguished. This means there is no special divergent process as there is in CSP.

The most powerful equivalence used is strong bisimulation. A bisimulation between two processes is a binary relation between the states of those processes. This is most easily described in terms of the state transition diagrams. Two processes, g and h, are identified by the bisimulation R if, and only if,

1. if $(s,t) \in R$ and $s \xrightarrow{a} s'$ is an edge in g then there is an edge $t \xrightarrow{a} t'$ in h, with the same label $a \in \mathrm{A} \cup \{\tau\}$, such that $(s',t') \in R$,

2. if $(s,t) \in R$ and $t \xrightarrow{a} t'$ is an edge in h, then there is an edge $s \xrightarrow{a} s'$ in g with the same label $a \in \mathrm{A} \cup \{\tau\}$, such that $(s',t') \in R$.

CCS identifies fewer processes than CSP because of τ. Strong bisimulation uses the internal action to distinguish between processes, whereas CSP has no notion of such an action. Other equivalences, such as weak bisimulation, testing equivalence, identify more processes.

Weak bisimulation allows us to identify processes which behave in the same way externally, but which may have different internal behaviour. This is important because we are frequently more concerned with how a process appears from the outside; we don't care what else it does.

LOTOS, the language of our second experiment, is in two parts; the abstract data type part and the process part. The process part is based on CCS and retains many of its features. A description of LOTOS is given in section 5.1.1.

It can be seen from the above that CCS and CSP provide two different complementary aspects to our experiments: while one allows us to explore the different equivalences between processes (useful for verification purposes), the other provides us with a set of transformations on processes (useful for building a process). We now go on to present an introduction of term rewriting.

3 Equational Reasoning and Term Rewriting

In a term rewriting system, we manipulate directed rules obtained by orienting equations. This provides a means of reasoning about equational specifications, i.e. specifications that are given by a signature (a set of function names and arities) and a set of equations specifying the properties of the functions in the signature. Much of the reasoning involved in the examples in [21] is equational, making term rewriting an obvious choice for reasoning about CSP. Term rewriting is also particularly suitable for the algebraic data type part of LOTOS. We intend to investigate how suitable equational reasoning is for the process part.

We may also view equations as definitions of functions, as in functional programming. For LOTOS, this relates to the transition laws, which define the semantics of the process part of the language, as opposed to the weak bisimulation congruence laws, which define the properties of the operators of LOTOS under weak bisimulation congruence. These have been investigated extensively using other forms of theorem provers — we intend to concentrate on the process algebra.

The use of term rewriting can also give us a solution to the word problem, i.e. given a set of pairs (X_i, Y_i) [the dictionary], where Y_i may be substituted for X_i with no change of meaning, are two given words of the language, X and Y, equivalent?

If the words in the dictionary were treated as equations, a massive amount of space would be required to solve this problem as we could apply the substitutions either way, effectively doubling the number of possible substitutions to try at any stage in solving the problem. The amount of space required to solve the problem is called the *search space* of the solution. If, however, the words in the dictionary were treated as rewrite rules, with a direction of application, then we have halved the number of possibilities at each stage of the computation, making the computation more tractable.

Term rewriting systems can give us a decision procedure for the equality of terms: given two terms, we can always answer yes or no about their equivalence. This is why we use term rewriting for our examples — we wish to show two processes to be equivalent.

Term rewriting theorem provers can provide a variety of other services such as proof by induction, narrowing, completion modulo a set of equations etc. There are many approaches to proof by induction in term rewriting systems, but in LP, the system used in our experiments with CSP, induction is performed in the traditional way, by proving the base case and then the hypothesis step. Both of these proofs are done by rewriting. The other services of term rewriting theorem provers are not detailed here; they are not relevant to our work.

In the following sections we give some standard definitions from term rewriting. For further details see [22, 20].

3.1 Terminology

Terms are constructed from fixed arity function symbols and variables in the usual way. *Ground terms* are terms that contain no variables. Although many systems can deal with sorted terms, we will refer only to unsorted terms, to simplify matters.

Terms will be denoted by $t, t', t'', t_1, t_2, \ldots$ etc.

A term t is a *subterm* of itself. If t is of the form $f(t_1, \ldots, t_n)$, where f is a function of arity n, then any subterm of any of the t_i, for $i = 1, \ldots, n$, is also a subterm of t.

A variable may be instantiated by replacing it with any term. Substitution is denoted by $t[t'/x]$; i.e. every occurrence of x in t is replaced by t'. Variables will be denoted by $u, v, w, x, y, z, \ldots$ etc.

Given terms t and t', *matching* t to t' is the process of finding a substitution for variables in t that make it identical to t'. *Unifying* t and t' is the process of finding substitutions of terms for variables in both t and t' which makes them identical.

3.2 Term Rewriting Systems

A term rewriting system, R, consists of a set of directed equations, $\{\lambda_i \rightarrow \rho_i \mid i \in I\}$. An example of a term rewriting system is:

$$x + 0 \rightarrow x \qquad \text{R1}$$
$$0 + x \rightarrow x \qquad \text{R2}$$
$$-x + x \rightarrow 0 \qquad \text{R3}$$
$$(x + y) + z \rightarrow x + (y + z) \qquad \text{R4}$$

In the examples in this section, u, v, w, x, y, z denote variables, and a is a constant.

How the equations are given a direction of application is determined by the *ordering*. Orderings specify which side of a rule is more complex than the other, and we usually orient the rules so that we go from more complex to less complex since this ensures that we have no infinite chains of rewrites. Notice that all of the rules in the system given above, with the exception of the associative rule, have smaller terms on the right hand side of the $\rightarrow$ than on the left hand side. The associative rule is ordered so that the brackets group to the right.

The most common orderings used are Knuth-Bendix Ordering (KBO) and Recursive Path Ordering (RPO). We use RPO for our LOTOS experiments, and a simple left to right ordering for our CSP experiments i.e. the rules are ordered in the direction they are entered. For further information on orderings see [8].

To *rewrite* a term $T = f(t_1, \ldots, t_n)$, we find a t_j, for $j \in 1, \ldots, n$, such that one of the λ_i in the rewriting system can be matched to t_j, i.e. $t_j = \sigma\lambda_i$, where σ is a substitution. We then replace t_j in T by $\sigma\rho_i$. For example, the term $-a + a$ rewrites to 0 by the substitution $R3[a/x]$. Iteration of this process is called reducing or rewriting.

The reduction relation $\rightarrow_R$ associated with R, the term rewriting system, is the smallest relation over terms containing R and closed by substitution and replacement. The transitive-reflexive closure of this relation is denoted as usual, $\rightarrow^*$.

The process of reducing a term until none of its subterms can be matched with the left hand side of any of the rewrite rules is called *normalisation*, i.e. given t, there are no rules $\lambda \rightarrow \rho$ such that λ can be matched to a subterm of t.

In the example above, none of the rules has a left hand side which matches *0*. Therefore *0* is in *normal form*.

In a rewriting system, we prove two terms to be equal by reducing them until they are the same term. However, not all terms that are equivalent in the equational theory can be proved equivalent by the rewrite rules. Consider a proof of $- - a \equiv a$, given the rules above. Each side here is in normal form; we cannot reduce either term any further. This is because there are two kinds of proof: a rewrite proof and a non-rewrite proof. If you imagine the sequence of rewrites in a proof as a picture, with the most complex terms at the top, then a rewrite proof has troughs, while a non-rewrite proof has peaks. For example, a rewrite proof of $(-a + a) + a \equiv (a + -b) + b$, and a non-rewrite proof of $- - a \equiv a$:

$$
\begin{tikzcd}
 & & & & (a+-b)+b \arrow[d, "R4"] \\
(-a+a)+a \arrow[dr, "R3"'] & & & & a+(-b+b) \arrow[dl, "R3"] \\
 & 0+a \arrow[dr, "R2"'] & & a+0 \arrow[dl, "R1"] & \\
 & & a & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & (--a+-a)+a \arrow[dl, "R4"'] \arrow[dr, "R3"] & \\
--a+(-a+a) \arrow[d, "R3"'] & & 0+a \arrow[dd, "R2"] \\
--a+0 \arrow[d, "R1"'] & & \\
--a & & a
\end{tikzcd}
$$

If an equivalence has a non-rewrite proof then we are unable to prove it in the term rewriting system, even though we know it to be true in the equational system. In forcing the equations to be uni-directional rewrite rules we have lost the original power of the equational system. To regain this power we use *Knuth-Bendix completion* [25].

3.3 Completion

For a set of rewrite rules to be *complete* we require two properties: the rule set must be *confluent* and *terminating*. These properties ensure that every term has a unique normal form and give us a decision procedure for equality of terms, which means that we can prove everything we could prove in the equational system.

A term rewriting system is *confluent* if, and only if, for all t_1, t_2 and t, such that $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$, there exists a t' such that $t_1 \rightarrow^* t'$ and $t_2 \rightarrow^* t'$. This is also known as the diamond lemma for obvious reasons:

$$
\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 \arrow[dr, "*"'] & & t_2 \arrow[dl, "*"] \\
 & t' &
\end{tikzcd}
$$

Local confluence is like confluence except that we have there exists a t' such that $t_1 \rightarrow t'$ and $t_2 \rightarrow t'$, i.e. t' can be obtained in exactly one rewrite from t_1 and t_2.

Termination means that there are no infinite chains of reductions, i.e. rewrite sequences of the form $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$. A system with this property is also called Noetherian.

Both of these properties are undecidable in themselves. However, Newman's theorem [33] shows that a terminating term rewriting system is confluent if, and only if, it is locally confluent.

Also, a terminating term rewriting system is locally confluent if, and only if, every critical pair is trivial. (Knuth-Bendix theorem [25])

A *critical pair* is a pair of terms generated by applying two different rewrite rules (or two different applications of the same rewrite rule) to a term (the *critical expression*). The critical expression is also the peak of a non-rewrite proof. In the example of $- - a \equiv a$, the critical term is $(- - a + -a) + a$ and the critical pair is $- - a + (-a + a)$, obtained by applying R4 to the critical term, and $0 + a$, obtained by applying R3 to the critical term. We want to eliminate all the peaks of non-rewrite proofs, giving rewrite proofs instead. To do this automatically, critical pairs are generated from rewrite rules by *superposition*. This means unifying subterms of the left hand sides of any of the rewrite rules. Rules may also be superposed onto themselves. Knuth-Bendix completion works by generating all critical pairs and adding non-trivial pairs as rules until no more non-trivial critical pairs are generated.

To illustrate this procedure, consider how we might convert the proof of $- - a \equiv a$ into a rewrite proof. The superpositions to make are obvious because we know what we are aiming towards. First we superpose $R3$ on $R4$ using the substitution $[-v/x, v/y, w/z]$:

$$-x + x = 0 \qquad R3$$
$$(x + y) + z = x + (y + z) \qquad R4$$

The superposition is:

$$\begin{tikzcd}
 & (-v + v) + w \arrow[dl, "R4"'] \arrow[dr, "R3"] & \\
-v + (v + w) & & 0 + w
\end{tikzcd}$$

This gives us the new rule $-v + (v + w) \to 0 + w$. We call the normalised version of this rule, $-v + (v + w) \to w$, $R5$. This simplifies the proof of $- - a \equiv a$:

$$\begin{tikzcd}
 & & - - a + (-a + a) \arrow[dl, "R3"'] \arrow[ddr, "R5"] & \\
 & - - a + 0 \arrow[dl, "R1"'] & & \\
- - a & & & a
\end{tikzcd}$$

From this we can see that the next superposition to make is $R3$ on $R5$, using substitution $[- - v/x, -v/y, v/z]$.

$$
\begin{tikzcd}[column sep=small]
 & --v+(-v+v) \arrow[dl, "R3"'] \arrow[dr, "R5"] & \\
--v+0 & & v
\end{tikzcd}
$$

giving us the rule $--x+0 \rightarrow x$. Normalising this rule we get $--x \rightarrow x$, $R6$. The proof for $--a \equiv a$ is now:

$$
\begin{tikzcd}
--a \arrow[d, "R6"] \\
a
\end{tikzcd}
$$

A rewrite proof!

As stated earlier, the new rules in this example are easy to generate because we can see which superpositions to make by looking at the pictures. Normally, systematic superpositions of each rule with every other rule are made; this ensures that all critical pairs are generated.

Informally, the Knuth-Bendix completion algorithm behaves as follows: given termination ordering $>$, and initial state with axiom set A and rewrite rule set $R = \{\}$,

while $A \neq \{\}$ do
    begin
    select and remove an axiom $a \in A$;
    normalise a; (* this makes the algorithm more efficient *)
    if a is not of the form $x = x$ then
        begin
        orient a into r using $>$;
        add r to R;
        superpose r on all existing rules (including self) and
            for each critical pair (P, Q), add the axiom P = Q to A
        end
    end

Three things can happen as a result of running this procedure:

1. The algorithm may halt, leaving a confluent and terminating set R of rewrite rules.
2. The algorithm may halt because a critical pair has been generated that it cannot orient.
3. The algorithm may diverge, i.e. A will never be empty and we will keep generating new rules forever. This is an area of current research and will be discussed later in section 5.5.

If a complete (i.e. confluent and terminating) set of rules is obtained we can use this in a variety of ways. For example: we have a decision procedure for the word problem specified by the original equational system; we have a check for consistency and completeness of algebraic data types; we can carry out inductionless induction in the system, and we can carry out proofs by refutation in First Order Predicate Calculus.

We now look at how term rewriting can be applied in two different ways to concurrency formalisms. In section 4 we use a set of rewrite rules to perform equational and inductive proofs on concurrent systems described in CSP, and in section 5 we attempt to obtain a complete set of rewrite rules from the axioms of a LOTOS equivalence. We also consider methods of dealing with the problem of a diverging algorithm generating infinite sequences of rewrite rules.

4 CSP and LP

4.1 Aims

The aim in this section is to investigate how term rewriting can be used to reason about systems described in CSP.

4.1.1 CSP

CSP is a formalism defined by processes, which are mathematical abstractions of the interactions between a system and its environment. The behaviour of a process can be recorded as a trace of the sequence of actions in which it engages. The processes are constructed by operators whose behaviour are described by algebraic laws. The familiar technique of recursion describes processes that last a long time or forever. Processes can be assembled together in such a way that they interact with each other, and evolve concurrently. These interactions may be regarded as events which require simultaneous participation of both the processes involved. The formalism provides a secure mathematical foundation for avoidance of errors such as divergence, deadlock and non-termination.

Parts of CSP that are purely equational can be handled in a straightforward way, but clearly processes that are defined recursively will create problems because the obvious method of implementing them will generate infinite rewrite rules. We have looked at ways of overcoming these problems. In implementing CSP we have concentrated on examples which cover a broad range of CSP's capabilities, but for brevity have not used every possible operator. However there seems no reason why the methods employed cannot be applied to the rest of CSP. The operators and examples we have used are described below.

Processes and Operators Let x, y be distinct events and let P, Q and R be processes. Then

$x \rightarrow P$

denotes the process which first engages in the event x and then behaves as P.

In general, if B is any set of events and P(x) defines a process for each x in B, then

$(x{:}B \rightarrow P(x))$

defines a process that first offers a choice of any event y in B, and then behaves like P(y).

Deterministic choice $(x \rightarrow P \mid y \rightarrow Q)$

denotes the process which initially engages in either of the events x or y. Subsequently the process behaves as the process P if the first event was x, and as the process Q if the first event was y.

Non-deterministic choice $P \sqcap Q$

denotes the process which behaves either like P or like Q, where the selection between them is made arbitrarily, without the knowledge or the control of the external environment.

Concurrency $P \parallel Q$

denotes the concurrent process which which behaves like the system composed of P and Q interacting in lock-step synchronization.

Deadlock STOP

denotes the deadlocked process.

The behaviour of these processes is governed by the following laws:

$$
\begin{aligned}
&P \parallel Q = Q \parallel P \\
&P \parallel (Q \parallel R) = (P \parallel Q) \parallel R \\
&P \parallel STOP = STOP \\
&(x \rightarrow P) \parallel (x \rightarrow Q) = (x \rightarrow (P \parallel Q)) \\
&(x \rightarrow P) \parallel (y \rightarrow Q) = STOP
\end{aligned}
$$

The last law generalizes to cases when one or both processes offer a choice of initial event:

$$(x{:}A \rightarrow P(x)) \parallel (y{:}B \rightarrow Q(y)) = (z{:}(A \cap B) \rightarrow (P(z) \parallel Q(z)))$$

$$
\begin{aligned}
&P \sqcap Q = Q \sqcap P \\
&P \sqcap P = P \\
&P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \\
&x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q) \\
&P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)
\end{aligned}
$$

4.1.2 Examples

We consider some standard examples of processes taken from [21] which involve choice, non-determinism and deadlock. So consider the following recursively defined processes which represent various vending machines or their customers.

There are 3 kinds of events:

coin	the insertion of a coin into a vending machine
choc	the dispensing of a chocolate from a vending machine
toffee	the dispensing of a toffee from a vending machine

$$MACND = coin \rightarrow ((choc \rightarrow MACND) \sqcap (toffee \rightarrow MACND))$$

defines a machine which serves either chocolate or toffee in a non-deterministic way, that is to say the choice is not made by the customer.

$$CUSTT = coin \rightarrow toffee \rightarrow CUSTT$$

represents a customer who always chooses toffee.

$$CUSTW = coin \rightarrow ((choc \rightarrow CUSTW) \sqcap (toffee \rightarrow CUSTW))$$

is a customer who chooses chocolate or toffee at whim.

$$CUSTA = coin \rightarrow (choc \rightarrow CUSTA \mid toffee \rightarrow CUSTA)$$

is a customer who is willing to accept whichever the machine gives.
We can now form the concurrent processes

1. $MACND \parallel CUSTT$
2. $MACND \parallel CUSTW$
3. $MACND \parallel CUSTA$

and use the laws to translate them into a form using $\rightarrow$, $\mid$ and $\sqcap$, but not $\parallel$. It is easily (but tediously) verified that after application of the relevant rules the respective processes translate to:

1. $coin \rightarrow (STOP \sqcap (toffee \rightarrow (MACND \parallel CUSTT)))$

Here the customer can take toffees as long as MACND allows; deadlock occurs when MACND "chooses" chocolate.

2. $coin \rightarrow$
$$((choc \rightarrow (MACND \parallel CUSTW)) \sqcap (toffee \rightarrow (MACND \parallel CUSTW)) \sqcap STOP)$$

In this case any number of chocolates and toffees may be obtained until the machine and the customer make different choices. Then deadlock occurs.

3. $coin \rightarrow$
$$((choc \rightarrow (MACND \parallel CUSTA)) \sqcap (toffee \rightarrow (MACND \parallel CUSTA)))$$
$$= MACND$$

Here the non-determinism of MACND is rendered harmless by the willingness of the customer to accept either choice made by MACND. Later we show how LP can prove by rewriting that these are the equivalent processes.

The second example that we consider concerns modified versions of MACND and CUSTT, which will be denoted by ND and TC respectively, in which the vending machines have a finite capacity of sweets. The action 'refill' is added to the possible events, and the processes are defined in terms of mutual recursion with a number of processes which record the number of sweets remaining.

$ND = ND(0)$
$ND(0) = (refill \rightarrow ND(3))$
$ND(n) = ((coin \rightarrow ((choc \rightarrow ND(n-1)) \sqcap (toffee \rightarrow ND(n-1)))) \mid (refill \rightarrow ND(3)))$
$\text{for } n = 1, 2$

The machine ND has a capacity of 3 sweets and can be refilled at any stage. Otherwise it behaves like MACND. Of course the capacity could be made larger without altering the form of proof to be performed; 3 was chosen simply for speed of computation within LP.

$TC = TC(0)$
$TC(0) = (refill \rightarrow TC(3))$
$TC(n) = ((coin \rightarrow ((toffee \rightarrow TC(n-1))) \mid (refill \rightarrow TC(3)))$ $\qquad \text{for } n = 1, 2$

Thus TC is a vending machine which dispenses toffee only and also has a capacity of 3. Again we use LP to reason about the concurrent process ND $\|$ TC, and translate this expression into a form without the concurrency operator. This is of interest to see how LP can cope with mutual recursion; we use induction on the natural numbers.

4.2 Representation

4.2.1 LP

The Larch Prover [14] is an equational reasoning theorem prover developed at MIT by Steve Garland and John Guttag. It supports proof by induction, rewriting and completion, and is used interactively for designing, coding and debugging proofs.

LP is based on equational term-rewriting. That is, given a set of operators and some equations which they satisfy, each equation is oriented into a directed substitution or rewrite rule. The inference mechanism is that of replacing subterms which match the left hand side of a rule by the corresponding right hand side. Thus if LP is given the rule $x + 0 \rightarrow x$ where x is a variable, it can match this twice against $(1 + 0) + 0$ to get 1. LP also supports associative-commutative (a-c) matching, so that if + is declared to be a-c the law $x + 0 \rightarrow x$ will also match $0 + (0 + 1)$ to obtain 1. LP also supports proof by induction. We use both of these proof forms to reason about CSP.

Classical term rewriting, and theorem provers such as Reve [27] and RRL [38] have been much concerned with the completion algorithm. We do not use completion in our CSP work, but in our LOTOS experiments we are interested in completion, and use RRL as our term rewriting tool.

4.2.2 Implementation

The representation we have chosen of the CSP operators within LP is shown in Table 1. This is influenced by the built-in operators of LP. For instance, $\mid$ is a prescribed logical operator, and so we use $\|$ to represent the deterministic choice operator, and $|||$ to represent the concurrency operator. Similarly $=$ and $\rightarrow$ need to be avoided as they have built-in meanings.

The CSP laws that are listed earlier are implemented as follows. The convention is that variables are represented by the letters u, v, w, x, y, z, U, V, W, X, Y and Z. Notice that we need to make use of the built-in operator *if*. This has simplification metarule

<table>
<tr><th>Operator Name</th><th>CSP</th><th>LP</th></tr>
<tr><td>deterministic choice</td><td>|</td><td>||</td></tr>
<tr><td>non-deterministic choice</td><td>⊓</td><td>|–|</td></tr>
<tr><td>concurrency</td><td>||</td><td>|||</td></tr>
<tr><td>deadlock</td><td>STOP</td><td>stop</td></tr>
<tr><td>“arrow”</td><td>→</td><td>*→</td></tr>
</table>

Table 1: CSP Operators and corresponding LP operators

```
if (t1,t2[t1],t3[t1]) → if (t1,t2(true),t3(false))
```

So the LP session is begun by providing the following equations.

```
add

U ||| stop == stop

((x *→ U) ||| ((y *→ V) || (z *→ W))) ==
if (x = y,
    (x *→ (U ||| V)),
    if (x = z,
        (x*→(U ||| W)),
        stop))

% special case of generalized choice
(((x *→ X) || (y *→ Y)) ||| ((z *→ Z) || (u *→ U))) ==
if (x = z,
    if (y = u,
        ((x *→ (X ||| Z)) || (y *→ (Y ||| U))),
        (x *→ (X ||| Z))),
    if (y = u,
        (y *→ (Y ||| U)),
        stop))

(x *→ U) ||| (y *→ V) ==
if (x = y,
    (x*→ (U ||| V)),
    stop)

U |–| U == U

x *→ (U |–| V) == (x *→ U) |–| (x *→ V)

U ||| (V |–| W) == (U ||| V) |–| (U ||| W)
```

```
..
op ac  |||   |−|

set ordering left-to-right
order-automatically
```

The .. indicates end of input. The *op ac* command declares the operators ||| and |−| to be associative and commutative. The choice of letters to represent events and processes conforms to LP's prescribed letters for variables. The last two commands order the equations into rules with the left hand side greater than the right. LP supports proofs of termination but we have not used these here.

The underlying semantics of LP does not assume that constants are distinct unless they have been proved to be so by completion, or declared to be so by the user. Thus we use the device below to generate an alphabet of distinct constants.

```
% Constants are c(0), c(1), c(2) etc
% Need to say that c(1) is not equal to c(2) etc

add
1 == s(0)
2 == s(1)
3==s(2)
4=s(3)
5 = s(4)
6=s(5)
7=s(6)
8=s(7)
9=s(8)
10=s(9)

% Inequalities
(s(x)=s(y))==(x=y)
(0=1)==false
(0=2)==false
(0=3)==false
(0=4)==false
(0=5)==false
(0=6)==false
(0=7)==false
(0=8)==false
(0=9)==false
(0= 10)==false
c(x)=c(y)== x=y
..
```

4.3 Results

We show how the particular examples described above in section 4.1.2 are implemented in LP, and see the results.

4.3.1 Rewriting Proof

Firstly the alphabet and the axiom file are executed:

ex alphabet
ex axioms

The relevant constants are renamed:

add
coin==c(1)
choc==c(2)
toffee==c(3)
MACND == MACND
CUSTT == CUSTT
CUSTW == CUSTW
CUSTA == CUSTA
..

We can now use the *prove* command to check our claims about the concurrent processes; this command attempts a proof by rewriting both sides of the equation to see if they are the same. Thus in order to calculate MACND || CUSTT we enter:

prove
*(coin *→ ((choc *→ MACND) |−| (toffee *→ MACND))) |||*
*(coin *→ toffee *→ CUSTT)==*
*coin *→ (stop |−| (toffee *→ (MACND ||| CUSTT)))*
..

The other two concurrent processes are dealt with in the same fashion. The output is as follows.

```
Conjecture
    (coin *→ (toffee *→ CUSTT)) |||
    (coin *→ ((choc *→ MACND) |−|(toffee *→ MACND)))
    == coin *→ ((toffee *→ (CUSTT |||MACND)) |−|stop)
    [] Proved by rewriting.

Conjecture
    (coin *→ ((choc *→ CUSTW) |−|(toffee *→ CUSTW))) |||
    (coin *→ ((choc *→ MACND) |−|(toffee *→ MACND)))
    == coin *→
```

```
        ((choc *→ (CUSTW ||| MACND))
        |−| (toffee *→ (CUSTW ||| MACND)) |−| stop)
[] Proved by rewriting.

Conjecture
    (coin *→ ((choc *→ MACND) |−| (toffee *→ MACND))) |||
    (coin *→ ((choc *→ CUSTA) || (toffee *→ CUSTA)))
    == coin *→
        ((choc *→ (CUSTA ||| MACND)) |−|
        (toffee *→ (CUSTA ||| MACND)))
[] Proved by rewriting.
```

The reader may wonder why we didn't define recurrent processes by giving rewrite rules like

*MACND → coin *→ ((choc *→ MACND) |−| (toffee *→ MACND))*
*CUSTT → coin *→ toffee *→ CUSTT*

and then asking the machine to translate MACND || CUSTT. Of course this would not work because it would lead to an infinite rewriting loop.

4.3.2 Induction Proof

The input for this proof is similar to that of the previous proof, but it is also necessary to declare generators for natural numbers in order to enable LP to do induction. In this formulation natural numbers are generated by 0 and s.

```
add-generators
0: → nat
s: nat → nat
..
```

The point of this declaration is to establish the structural induction schema:

$$\frac{P(0) \wedge (\forall\ x.P(x) \Rightarrow P(s(x)))}{\forall\ x.P(x)}$$

We next declare the equations we are going to use.

```
add
% nats
x+s(y) == s(x+y)
x+0 == x
s(x)*y == (x*y)+y
x*(y+z) == (x*y) +(x*z)
0*x == 0
1*x == x
```

```
%capacity of vending machine
ok(y) ==   if (y = 0,
                true,
                if (y = s(0)),
                     true,
                     if (y = s(s(0)),
                          true,
                          false)))

% vending machine events
refill == c(1)
coin == c(2)
toffee == c(3)
choc == c(4)

% vending machine dispensing toffee only
TC(0) == (refill *→ T(3))
TC(s(xn)) == if(ok(xn),
                ((coin *→ (toffee *→ TC(xn))) || (refill *→ T(3))), stop)

% vending machine dispensing chocolate or toffee at whim
ND(0) == (refill *→ N(3))
ND(s(xn)) == if(ok(xn),
                ((coin *→((choc *→ ND(xn)) |−| (toffee *→ ND(xn))))
                || (refill *→ N(3))),
                stop)

CC(0) == (refill *→ (T(3) ||| N(3)))
CC(s(xn)) == if(ok(xn),
                ((coin *→ (stop |−| (toffee *→ (TC(xn) ||| ND(xn)))))
                || ( refill *→ (T(3) ||| N(3)))), stop)

..
```

Note that the processes are defined in terms of the natural numbers to allow proof by induction. It is necessary to relabel TC(3) by T(3) etc. to avoid generating an infinite rewrite loop. Finally, the proof by induction command takes the following form.

```
prove
     (TC(xn) ||| ND(xn)) == CC(xn)
by induction xn nat
..
```

The proof is by induction on the variable *xn* of type *nat* and now goes through without any further user input. The typing information selects the structural induction schema for natural numbers. The output from the *prove* command shown below indicates how the proof is performed. First the base case is proved by rewriting, then the induction hypothesis is generated with the variable *xn* being replaced by a new constant *c1* and finally the lemma needed for the induction step is generated and proved. After the proof the normalised form of the proved equation is added as a new rule ordered from left to right.

```
The basis step in an inductive proof of Conjecture 46
    ND(xn) |||TC(xn) == CC(xn)
involves proving the following lemma(s):

46.1:  ND(0) |||TC(0) == CC(0)
    [] Proved by normalization

The induction step in an inductive proof of Conjecture 46
    ND(xn) |||TC(xn) == CC(xn)
uses the following equation(s) for the induction hypothesis:

Induct.1:  ND(c1) |||TC(c1) == CC(c1)

The system now contains 1 equation and 44 rewrite rules.

Ordered equation Induct.1 into the rewrite rule:
    ND(c1) |||TC(c1) → CC(c1)

The system now contains 45 rewrite rules.

The induction step involves proving the following lemma(s):

46.2:  ND(s(c1)) |||TC(s(c1)) == CC(s(c1))
    [] Proved by normalization

Conjecture 46
    ND(xn) |||TC(xn) == CC(xn)
[] Proved by induction over 'xn' of sort 'nat'.

The system now contains 1 equation and 44 rewrite rules.

Ordered equation 46 into the rewrite rule:
    ND(xn) |||TC(xn) → CC(xn)

    Right-hand side reduced:
    CC(s(xn))
    → if(if(s(0) = xn, (0 = xn) | true,
        if(s(s(0)) = xn, (0 = xn) | true, (0 = xn) | false)),
```

```
(c(s(s(0))) *⟶
((c(s(s(s(0)))) *⟶ (ND(xn) |||TC(xn))) |−|stop)) ||
(c(s(0)) *⟶ (N(s(s(s(0)))) |||T(s(s(s(0)))))), stop)
became rewrite rule 45:
CC(s(xn))
⟶ if(if(s(0) = xn, (0 = xn) | true,
    if(s(s(0)) = xn, (0 = xn) | true, (0 = xn) | false)),
    (c(s(s(0))) *⟶ ((c(s(s(s(0)))) *⟶ CC(xn)) |−|stop)) ||
    (c(s(0)) *⟶ (N(s(s(s(0)))) |||T(s(s(s(0)))))), stop)

The system now contains 45 rewrite rules.
```

This completes the induction proof and our work with CSP and LP.

5 LOTOS and RRL

5.1 Aims

5.1.1 LOTOS and Bisimulation

LOTOS (Language of Temporal Ordering Specification) is based on the concept of specifying a system in terms of observable behaviour, i.e. events. Constraints may be expressed on the order of events; however, there is no explicit representation of time. The communication and change of information within a system is expressed by the structure of the events.

LOTOS is made up of two distinct parts. The process part describes the behaviour of processes and is based on CCS, with some modifications drawn from other concurrency formalisms, such as CSP. The abstract data type part describes the observable behaviour of the data being exchanged by the processes.

There are some points of the LOTOS language that are sufficiently different from features of other concurrency languages for us to note them here.

- Communication occurs at *gates*.
- Successful termination, **exit**, also has the ability to pass values, just as a function in an imperative language can return a value. **exit** may produce several values.
- There are two special actions in LOTOS: **i** and δ. **i** represents an internal action by the process and cannot be observed; it is just like the τ action of CCS. δ is the action which moves from **exit** to **stop**. It is used to define the semantics of LOTOS, and is sometimes needed to reason about the laws for weak bisimulation congruence.
- The choice operator in LOTOS is like that of CCS — it is a deterministic choice unless the first event on each side of the choice is the same, in which case it becomes nondeterministic. A common way of ensuring a nondeterministic choice is to precede each option with the special event **i**.
- There are notations for three kinds of parallelism in LOTOS, each of which is based on the amount of synchronisation between the two processes in parallel. *Interleaving* — no synchronisation at

all. *General* — processes synchronise on events involving specified gates. *Full synchronisation* — synchronise on all events.

- Enabling is sequential composition of processes.
- Disabling allows a process to be interrupted at any time in its execution (before it terminates) by a second process, which takes over control.
- Relabelling means relabel gate names.
- Structured interactions come in two varieties: variable declaration and value declaration. These can be considered as roughly equivalent to input and output in CSP. However, as well as matching up input with output i.e. value passing, input can also be matched with input (synchronisation of processes) and output with output (value generation).
- Postfix selection is a means of imposing additional restrictions on the values that may be output (or input).

LOTOS also has a special deadlocked process and an operator to hide certain specified actions from an external observer.

Just as the semantics of CCS processes is given by transition trees, the semantics of LOTOS processes is given by structured labelled transition systems, i.e. each process is seen as a set of states connected by arcs representing actions, which move the process from one state to another. In addition, LOTOS semantics has data values from a many sorted algebra as labels to the transitions.

The most significant inheritance from CCS, as far as we are concerned, is the wide range of equivalences defined. For us, the most important of these is weak bisimulation congruence because it identifies more processes (strong bisimulation identifies fewer processes by use of the internal action), and it ensures that two processes will behave the same way in any environment.

It is useful for us to equate two processes because one may be more intelligible than the other. In particular, some methods of specification (constraint oriented specification) take advantage of the belief that it is easier for people to understand a large specification if it is built up from small processes in parallel with each other, while it can be more difficult to reason about such specifications.

Appendix B.2.2 to the LOTOS standard [40] contains a set of weak bisimulation congruence laws. We know these laws are sound with respect to the model [transition systems], but they are not complete i.e. not everything that can be proved true in the model can be proved true by these laws. This does not concern us as it is always possible to add specific laws to cover a particular case.

The following sections explain how Knuth-Bendix completion [25] was used to obtain a confluent and terminating set of rules from the weak bisimulation congruence laws, and the results obtained.

5.2 Representation

5.2.1 RRL

The term rewriting system used for this work was RRL (Rewrite Rule Laboratory) [24]. RRL can perform completion in the presence of a-c operators and has a limited form of order sortedness.

5.2.2 RRL functions

The sorts used in implementing LOTOS are: *process, event, gate, list* (plus three subtypes of list that have the basic list characteristics but have different kinds of elements), *exp* (arithmetic or boolean expressions) and the universal type, *univ*, which is built into RRL.

For the arities of the functions see figure 1.

stop	:	*process*
exit	:	*explist* $\rightarrow$ *process*
i	:	*event*
out	:	*gate explist* $\rightarrow$ *event*
selpred	:	*event exp* $\rightarrow$ *event*
seq	:	*event process* $\rightarrow$ *process*
ch	:	*process process* $\rightarrow$ *process*
par	:	*process process gatelist* $\rightarrow$ *process*
en	:	*process process* $\rightarrow$ *process*
dis	:	*process process* $\rightarrow$ *process*
hide	:	*gatelist process* $\rightarrow$ *process*
guard	:	*exp process* $\rightarrow$ *process*
relabel	:	*rlist process* $\rightarrow$ *process*
nl	:	*list*
cons	:	*univ list* $\rightarrow$ *list*

Figure 1: Arities of RRL functions

The way these functions are used to implement the corresponding LOTOS operators is given in the next section.

5.2.3 Representation of LOTOS constructs

LOTOS operators represented are given in Table 2. Note that all variables in RRL must start with u, v, w, x, y or z. We try to be consistent in using x, y and z for processes, v for lists and u for events. Some of the operators of LOTOS cause considerable problems, in particular **par**, because it is a ternary operator between two processes and a list of gate names that acts like an associative and commutative binary operator between two processes.

The full LOTOS language has not yet been considered. The most important operators not included in our subset of the language are: generalised choice (i.e. **ch** over a list of processes, rather than just two), **accept** (this is an extension of enable that allows processes to pass values as parameters, i.e. values from one process match up with variable names in another) and variable declarations (similar to input in its behaviour).

The definition of synchronisation involves a function, **L**, which takes a process and returns the label set of that process, i.e. all labels (gate names) used in a process that are not bound by a hiding operator. This function is also used in side conditions for many other rules, see section 5.5.1.

Operator Name	LOTOS	RRL
emptylist	[ ]	nl
inaction	**stop**	stop
termination	**exit** $(E_1, \ldots, E_n)$	exit (v)
action prefix	a; B	seq (u, x)
choice	B_1 [] B_2	ch (x, y)
general parallelism	$B_1 \mid[A]\mid B_2$	par (x, y, v)
interleaving	$B_1 \mid\mid\mid B_2$	par (x, y, nl)
synchronisation	$B_1 \mid\mid B_2$	par (x, y, v)
		where v = L(x) ∪ L(y)
hiding	**hide** A **in** B	hide (v, x)
internal action	**i**	i
enabling	$B_1 \gg B_2$	en (x, y)
disabling	B_1 [> B_2	dis (x, y)
guarded expressions	[E] → B	guard (w, x)
postfix relabelling	B[S]	relabel (v, x)
structured interactions	g!x	out (xg, xl)
(variable declaration)		
postfix selection predicate	g!x[E]	selpred (u, xexp)

Table 2: List of LOTOS expressions and corresponding RRL expressions

5.2.4 Representation of Weak Bisimulation Congruence Laws

A working set of laws was formed by adding rules one at a time, discarding laws that "misbehaved" (see section 5.5 for a discussion of these laws).

The laws successfully implemented and run through the completion algorithm are as follows (referred to as the core set):

(1) seq (selpred (out (xg, xl), u), y)	==	*guard (u, seq (out (xg, xl), y))*
(2) ch (x, y)	==	*ch (y, x)*
(3) ch (x, stop)	==	*x*
(4) ch (x, x)	==	*x*
(5) par (exit(nl), exit(nl), v)	==	*exit(nl)*
(6) par (exit(v), stop, nl)	==	*stop*
(7) en (stop, x)	==	*stop*
(8) en (exit(nl), x)	==	*seq (i, x)*
(9) en (x, en (y, z))	==	*en (en (x, y), z)*
(10) dis (x, stop)	==	*x*
(11) dis (stop, x)	==	*x*
(12) ch (dis (x, y), y)	==	*dis (x, y)*
(13) dis (exit(v), x)	==	*ch (exit(v), x)*
(14) hide (v, guard (u, x))	==	*guard (u, hide (v, x))*
(15) relabel (v, stop)	==	*stop*

(16) *relabel (v, exit(v2))*	==	*exit(v2)*
(17) *seq (u, seq (i, x))*	==	*seq (u, x)*
(18) *ch (x, seq (i, x))*	==	*seq (i, x)*
(19) *ch (seq (u, ch (x, seq (i, y))), seq (u, y))*	==	*seq (u, ch (x, seq (i, y)))*

The following information is also given to RRL:
function arities as given in figure 1,
sort ordering:

rlist	<	*list*
explist	<	*list*
gatelist	<	*list*

and function precedences:

guard < **seq** < **en** < **hide**
ch < **dis**
exit < **seq**

The ordering used is RPO (Recursive Path Ordering) [8].

5.3 Result of the Completion Procedure

The completion procedure, when given the above information, terminates and RRL reports that the resulting set of rules is canonical (i.e. confluent and terminating).

These are the rules generated. (Numbers in brackets after the rules denote how they were generated i.e. from which pair of rules the critical pair was obtained. *User* indicates that the rule was input by the user.) The first 19 are as entered, except the commutative choice law. The system keeps a special set of commutative and/or associative operators and therefore has no need of the actual axioms expressing commutativity or associativity.

(20) *ch (y, ch (y, exit(v)))*	→	*ch (y, exit(v))*	(13, 12)
(21) *en (en (x, stop), z)*	→	*en (x, stop)*	(7, 9)
(22) *en (seq (i, y), z)*	→	*seq (i, en (y, z))*	(8, 10)
(22) *en (x, seq (i, z))*	→	*en (en (x, exit(nl)), z)*	(8, 9)
(24) *en (en (en (x, exit(nl)), exit(nl)), x1)*	→	*en (en (x, exit(nl)), x1)*	(17, 23)
(25) *ch (seq (u, y), seq (u, dis (x, seq (i, y))))*	→	*seq (u, dis (x, seq (i, y)))*	(12, 19)

(26) *ch (guard (u, seq (out (xg, xl), y)),*
guard (u, seq (out (xg, xl), ch (x, seq (i, y))))) →
guard(u, seq (out (xg, xl), ch (x, seq (i, y)))) (1, 19)

(27) *ch (guard (u, seq (out (xg, xl), y)),*
guard (u, seq (out (xg, xl), dis (x, seq (i, y))))) →
guard (u, seq (out (xg, xl), dis (x, seq (i, y)))) (1, 25)

These rules can now be used in normalisation exercises and induction proofs.

5.4 Equational Proofs

5.4.1 Simple Examples

We can now prove the following two equations:

en (en (x, stop), y) == en (x, stop)
en (en (en (x, exit(nl)), exit(nl)), y) == en (en (x, exit(nl)), y)

In other words, it doesn't matter what else is in the chain of **en**'s if there is a **stop** in the middle of it — all things after the **stop** are automatically discarded. This is generated as a rule when the completion procedure is run. Similarly, a chain of **exit** functions is reduced to only one occurrence since, by rule (8), this could be rewritten as a sequence of **i**'s and, by rule (17), sequences of **i**'s can be cut down to just one occurrence of **i**.

5.4.2 Buffer Example

See figure 2. This example proves that a two-way buffer is the same as two one-way buffers in parallel. Unfortunately this example is a little forced as recursion is not part of our restricted LOTOS. This means that these buffers only handle one data element and then die!

In this example, x1, x2, y1 and y2 are all variables of sort *event* . Intuitively, they could represent the acceptance of data elements at a gate, or the output of data elements at a gate.

```
  (x1;  (x2;  (y1; y2; exit
              [] (y2; y1; exit))
        [] (y1; x2; y2; exit))
[] (x2;  (x1;  (y1; y2; exit
              [] y2; y1; exit)
        [] (y2; x1; y1; exit)))
```

is weak bisimulation congruent to

(x1; y1; exit) ||| (x2; y2; exit)

Figure 2: Buffer example

The second statement in this example is far easier to read than the first. This demonstrates that simply defined processes in parallel are frequently easier to understand than large process written without parallelism.

The rules already generated are not enough to prove the buffer example, laws relating some of the higher level constructs (such as **par**) to the basic constructs of **seq** and **ch** are also required. These are known as the expansion laws. Any LOTOS process can be rewritten using just **seq** and **ch**. The expansion law required here is the following:

$$
\begin{aligned}
\text{B} \mid[A]\mid \text{C} = \; & [] \; \{b_i;\, (B_i \mid[A]\mid \text{C}) \mid \text{name } (b_i) \notin A, i \in I\} \\
& [][] \; \{c_j;\, (\text{B} \mid[A]\mid C_j) \mid \text{name } (c_j) \notin A, j \in J\} \\
& [][] \; \{\text{a};\, (B_i \mid[A]\mid C_j) \mid \text{a} = b_i = c_j, \text{ name (a)} \in A, i \in I, j \in J\}
\end{aligned}
$$

where B $= [] \; \{b_i; B_i \mid i \in I\}$,
C $= [] \; \{c_j; C_j \mid j \in J\}$,
and *name* extracts the gate name from an event.

This law cannot be entered straight into RRL because generalised choice has not been implemented yet (as mentioned in section 5.2.3), so three useful instantiations of this expansion law are derived.

par (seq (u1, x1), seq (u2, x2), nl) == ch (seq (u1, par (x1, seq (u2, x2), nl)),
seq (u2, par (seq (u1, x1), x2, nl)))
par (seq (u1, x1), exit (nl), nl) == seq (u1, par (x1, exit(nl), nl))
par (exit (nl), seq (u1,x1), nl) == seq (u1, par (exit(nl), x1, nl))

Note that *exit* $\equiv \delta$*; stop*, which gives it the form required by the expansion law.

When these are added to the core set we can prove the equivalence in figure 2 is an equational theorem. This proof does not require rules (20) – (27), which were generated by the completion algorithm, showing that an incomplete set of rules can also be useful.

5.5 Laws causing Infinite Sequences of Rewrite Rules

It was mentioned earlier that some weak bisimulation congruence laws were not included in the core set because they "misbehaved". These are the following:

(28) hide (v, en (x, y))== en (hide (v, x), hide (v, y))
(29) ch (x, ch (y, z)) == ch (ch (x, y), z)
(30) dis (x, dis (y, z)) == dis (dis (x, y), z)

These laws cause an infinite sequence of rewrite rules to be generated. It would be useful if these rules could be generalised by producing a finite set of rules that subsume the infinite sequence and are terminating and confluent. Further details of techniques for generalising infinite sequences of rules may be found in [36, 26, 37].

We will now consider these laws one by one.

5.5.1 Distributivity over en and hide

This is just a representative of a family of similar laws involving **hide** or **relabel** and the other operators.

(20) hide (v, en (x, y)) == en (hide (v, x), hide (v, y))

Infinite sequences following two different patterns can be identified in the rules generated. The generalisations for these are as follows.

en (x, y) → x
    where x is built from applications of **hide** *and* **en**
    and has **stop** *as the innermost right argument.*

and

en (en (x, y), z) → en (x, z)
    where y is built from applications of **hide**, **en** *and* **exit**,
    and y is a subterm of x.

These sequences could almost be eliminated by rules already in the core set. Rule (7) would eliminate the first sequence and rule (8) the second, if it were not for the applications of **hide** which are in the way. We need a law that will allow us to get rid of them. In appendix B.2.2 we have:

hide A in B = B *where* $A \cap L(B) = \emptyset$

This law was passed over originally because it requires conditional rewriting. We generated two instantiations of this law to give:

(31) hide (v, stop) == stop *since* $L(stop) = \emptyset$
(32) hide (v, exit(nl)) == exit(nl) *since* $L(exit(nl)) = \{\delta\}$
and $v \cap \{\delta\} = \emptyset$
as δ *may not appear as a gate name in v*

When the above are added to the core set the completion procedure terminates producing only one more rule in addition to those given in section 5.3.

(33) hide (v, seq (i, y)) → seq (i, hide (v, y)) *(8, 28)*

5.5.2 Associativity for ch

The rules in the infinite sequence generated by the addition of rule (29) follow this basic pattern:

ch (z, y) → y

This rule is of course undesirable as it makes choice totally unfair (it will always choose the second argument). Also it is not an exact generalisation (i.e. every instantiation is a rule) and so adding it would imply unsound equivalences. We know that z is a subterm of y (by inspection of the rules generated), but there is a more complex relationship between these two terms, which deserves further investigation.

5.5.3 Associativity for dis

When rule (30) is added to the core set we see several different patterns emerging. Some are like this:

dis (x, exit(v)) → x
dis (x, seq (i, exit (v)) → x
    where x is formed from applications of **dis** *and* **exit**

While others follow these patterns:

In all cases y is built from ch, seq, dis and new variables
ch (exit(v), y) $\rightarrow$ *y*
ch (y, z) $\rightarrow$ *z*
where y is a subterm of z
ch (x, y) $\rightarrow$ *y*
where x is a subterm of y and x is composed of applications of **dis**, **seq** *and* **exit**
ch (ch (z, exit(v)), y) $\rightarrow$ *y*
where z is a subterm of y, and exit(v) is a subterm of y

This seems to be the hardest sequence of all to generalise since it is composed of many sequences. We get a different infinite sequence if rule (13) is taken out of the core set.

(13) dis (exit(v), x) == ch (exit(v), x)

This time we have only one pattern.

ch (arg, dis (x, arg)) $\rightarrow$ *dis (x, arg)*
ignoring brackets since **dis** *is associative*

where the form of arg can be described as a grammar.

$arg ::=$ **dis** $(subterm_1, z)$
$subterm_i ::= y \mid$ **dis** $(subterm_{i+1}, v_i)$
where y and z are variables in RRL, and
v_i *is a new variable*

Further investigation is necessary on the generalisation of these infinite sequences.

5.6 Other Results

In section 5.1.1 we stated that the weak bisimulation congruence laws in appendix B.2.2 of the LOTOS standard were sound. We suspected that this was not true and managed to prove that this was the case during our experiments.

When deciding what laws to accept as part of the core set we entered:

en (x, stop) $\rightarrow$ *par (x, stop, nl)*

When taken with rules (6) and (8), this rule causes a contradiction as follows:

seq (i, stop) $\rightarrow$ *stop*

Under weak bisimulation congruence this is false as the two equivalent processes are required to behave in the same way in all contexts. To test this condition it is necessary only to test the way in which the processes behave when part of a choice statement. For our example, let *x = seq (i, exit (nl))* in *ch (x, stop)* $\not\equiv$ *ch (x, seq (i, stop))*. We then have

ch (seq (i, exit (nl)), stop) $\not\equiv$ *ch (seq (i, exit (nl)), seq (i, stop))*

We know this is not an equivalence since choice is forced to be nondeterministic when each alternative is prefixed by the internal action **i**. While the left hand side will always terminate successfully, the right hand side may either terminate successfully or deadlock.

The law in the standard will be sound if a side condition is added.

$$en\ (x,\ stop) \rightarrow par\ (x,\ stop,\ nl) \qquad \text{where } x \neq exit\ (v)$$

6 Related Work

The problem of coping with divergence of Knuth-Bendix completion is also discussed in [7, 23] as a side effect of trying to develop a system to manipulate CCS specifications. De Nicola, Inverardi and Nesi came up against the problem of divergence when trying to complete the axioms for observational congruence (which is another name for weak bisimulation congruence). Instead of looking at term rewriting techniques to reduce the infinite sequences they developed an 'ad hoc' rewriting strategy to normalise finite CCS terms, avoiding the use of completion altogether. This strategy has been proved to be correct and complete with respect to the observational axiomatisation.

On the other hand, the aim of our project is to extend term rewriting theory to deal with the problems we discover in trying to verify aspects of LOTOS processes. Such extensions would hopefully be useful in other situations. Our preliminary investigations have posed questions that are the subject of current work by our colleagues on the ERIL project [37].

In short, while their work is applicable to LOTOS (because of its relationship with CCS), it may not be applicable to other concurrency formalisms, whereas we hope that our solutions will be applicable to a wider area of term rewriting, not just concurrency.

7 Conclusions and Further Work

We conducted two experiments in term rewriting and its application to concurrency formalisms. While the results we obtained were hopeful, we appreciate that there is still much work to be carried out in this area. We identify some of these below.

We have seen that by using LP's facilities for proof by rewriting and proof by induction we are able to reason about examples that are typical of a wide variety of CSP processes. Although not every CSP operator has been considered those that have not been explicitly implemented are similar in structure to ones we have dealt with. For instance, the general choice operator can be implemented in an analogous way to the nondeterministic choice operator.

A possible development for the CSP work would be to concentrate on the trace model of processes, which would then allow consideration of failures and divergences. Some progress has already been made on this approach and we are continuing research in this area.

As mentioned in section 5.2.3, not all LOTOS constructs have been considered. There are laws that use these still to be added to the core set. There are also laws that require conditional rewriting. An implementation of basic list functions is needed if any proofs are to be carried out, since many of the laws still to be considered use side conditions involving lists. We would also like to represent the expansion laws directly.

Previous work on LOTOS and theorem proving [3, 12] has concentrated on using the transition rules of LOTOS to perform process proofs. The transition rules give the semantics of LOTOS processes and could be useful in unravelling processes to ensure, say, that a particular state is reached. This approach suffers from the state explosion problem in the same way as graph reachability analyses of processes. We would like to look at this area in more depth.

The topic of generalising infinite sequences has already been explored by several people [26, 36, 37]. Further research in this area is indicated by some of the results we obtained in this experiment.

This preliminary investigation has been successful in that not only have we have shown that term rewriting is useful in reasoning about concurrency, we have also uncovered several areas for further development.

Acknowledgements

We should like to thank Steve Garland and John Guttag for letting us have a copy of LP, and Deepak Kapur for letting us have RRL. We would especially like to thank Ursula Martin, Muffy Thomas, Phil Watson and Dave Cohen, our colleagues. We should also like to thank British Telecom, our industrial "Uncles" on the ERIL project.

References

[1] J. Bergstra, G. Klop, Algebra of Communicationing Processes with Abstraction, Theoretical Computer Science 37, pp 77-121, 1985.

[2] G. Bochmann, Usage of Protocol Development Tools: the results of a survey *in* Proc of 7th IFIP Symposium on Protocol Specification, Testing and Verification, 1987, North Holland.

[3] R. Booth, An Evaluation of the LCF Theorem Prover using LOTOS, Proc. of 2nd International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, Elsevier Science Publications, North-Holland, 1989.

[4] R.S. Boyer and J.S. Moore, A Computational Logic. Academic Press, New York, 1979.

[5] E. Brinksma, On the design of Extended LOTOS - A Specification Language for Open Distributed Systems, North Holland, 1988.

[6] R. Cleveland, J. Parrow and B. Steffen, The Concurrency Workbench, *in* Automatic Verification Methods for Finite State Systems, ed J Sifakis, LNCS 407.

[7] R. De Nicola, P. Inverardi and M. Nesi, Using the Axiomatic Presentation of Behavioural Equaivalences for Manipulating CCS Specifications, *in* Automatic Verification Methods for Finite State Systems, ed J Sifakis, LNCS 407.

[8] N. Dershowitz, Orderings for Term Rewriting Systems, Theoretical Computer Science 17, 1982.

[9] A. J. J. Dick, "Equational Reasoning and the Knuth-Bendix Algorithm — an Informal Introduction", Imperial College Research Report DOC 84/21, Mar 1984.

[10] A J J Dick and J R Kalmus, "ERIL (Equational Reasoning: an Interactive Laboratory). User's Manual. Version R1.6a", Report RAL-88-055, Rutherford Appleton Laboratory (September 1988)

[11] H. Ehrig, J. Buntrok, P. Boehm, K-P. Hasler, F. Nurnberg, C. Rieckhoff, J.de Meer, Towards an Algebraic Semantics of the ISO-Specification Language LOTOS, SEDOS/C2/N58 1986.

[12] M. Fletcher, The Boyer-Moore Theorem Prover and LOTOS, Research & Technology Memorandum RT62/014/87, British Telecom, 1987.

[13] S.J. Garland, J.V. Guttag: *Inductive Methods for Reasoning about Abstract Data Types*, Proc. 15th ACM Symposium on Principles of Programming Languages (1988), San Diego, 219-228.

[14] S.J. Garland, J.V. Guttag: *An Overview of LP, The Larch Prover*, Proc. Rewriting Techniques and Applications, 3rd Intl. Conference, LNCS 355 (1989), 137-151.

[15] R.J.Van Glabbeek : Notes on the Methodology of CCS and CSP, 1986.

[16] J.C. Godskesen, K.G. Larsen and M. Zeeberg, TAV (Tools for Automatic Verification) : Users Manual, Internal Report, Aalborg University Center, Denmark, 1989.

[17] J.A. Goguen and J.Tardo, An Introduction to OBJ, a language for writing and testing software specifications, in Specification of Reliable Software, 170-189. IEEE, 1979.

[18] M. H. Goldsmith and A. W. Roscoe, Transformation of occam programs, *in* The Design and Application of Parallel Digital Processors, (IEE Conference Publication 298), 1988.

[19] J.V. Guttag, J.J. Horning: *Report on the Larch Shared Language*, Science of Computer Programming 6 (1986), 103-134.

[20] M. Hermann, C. Kirchner and H. Kirchner, Implementations of Term Rewriting Systems.

[21] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[22] G. Huet, D.C. Oppen: *Equations and Rewrite Rules - A Survey*, in: Formal Languages: Perspectives and Open Problems, R. Book (ed.), Academic Press (1982).

[23] P. Inverardi and M. Nesi, A Rewriting Strategy to Verify Observational Congruence, I.E.I. Internal Report B4-38, August 1989.

[24] D. Kapur and H. Zhang, RRL : Rewrite Rule Laboratory User's Manual, 1987, Revised May 1989.

[25] D.E. Knuth and P.B. Bendix, Simple Word Problems in Universal Algebras. In: *Computational Problems in Abstract Algebra*, pp. 193-214, ed. J. Leech, New York: Academic Press, 1970.

[26] St. Lange, Towards a Set of Inference Rules for Solving Divergence in Knuth-Bendix Completion, Proc. Analogical and Inductive Inference '89, GDR, LNCS 367, Springer-Verlag, 1989.

[27] P. Lescanne, Computer experiments with the REVE term rewriting system generator, Proc 10th ACM POPL symposium, Austin,TX 1983, pp99-108.

[28] D. B. MacQueen and D. T. Sannella, Completeness of Proof Systems for Equational Specifications, *in* IEEE Transactions on Software Engineering, Vol SE-11, No. 5, pp. 454–461, May 1985.

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

[30] R. Milner, Communication and Concurrency, Prentice-Hall,1989.

[31] H. Mohamed, Termination and Applications of the Knuth Bendix Algorithm, Ph D, University of Manchester, 1989.

[32] D. R. Musser, Abstract Data Type Specification in the Affirm System, IEEE Trans. Soft. Eng. vol SE-6, no. 1, 1980.

[33] M. H. A. Newman, "On Theoreis with a Combinatorial Definition of 'Equivalence"', *Annals of Mathematics* 43, 2 pp. 223 – 243 , 1942.

[34] L.C. Paulson, Interactive Theorem Proving with Cambridge LCF: a user's manual, University of Cambridge Computer Laboratory technical report no 80, 1985.

[35] J. Staunstrup, S. J. Garland and J. V. Guttag, Localised verification of circuit descriptions, *in* Automatic Verification Methods for Finite State Systems, ed J Sifakis, LNCS 407.

[36] M. Thomas and K.P. Jantke, Inductive Inference for Solving Divergence in Knuth-Bendix Completion, Proc. Analogical and Inductive Inference '89, GDR, LNCS 367, Springer-Verlag, 1989.

[37] M. Thomas and P. Watson, Solving Divergence in Knuth-Bendix Completion by Enriching Signatures, in preparation, 1990.

[38] H. Zhang, D. Kapur, M.S. Krishnamoorthy: *A Mechanizable Induction Principle for Equational Specifications*, Proc. 9th Intl. Conference on Automated Deduction, LNCS 310 (1988).

[39] D. J. Walker, Automated Analysis of Mutual Exclusion Algorithms using CCS, Formal Aspects of Computing Vol 1 No. 3, 1989.

[40] ISO 8807 : LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, 1988.

[41] ISO IS7498 : Information Processing Systems - Open Systems Interconnection - Basic Reference Model, 1984.