PAM: A Process Algebra Manipulator

Huimin Lin
Computer Science
School of Cognitive and Computing Sciences
University of Sussex

Abstract

PAM is a general proof tool for process algebras. It allows users to define their own calculi and then perform algebraic style proofs in these calculi by directly manipulating process terms. The logic that PAM implements is equational logic plus recursion, with some features tailored to the particular requirements of process algebras. Equational reasoning is implemented by rewriting, while recursion is dealt with by induction. Proofs are constructed interactively, giving users the freedom to control the proof processes.

1 Introduction

This paper describes a proof tool specially designed to assist equational reasonings in process algebras.

It has been gradually recognized that computer assistance is essential for the analysis of concurrent systems. There already exit a number of proof tools, among them are the Concurrency Workbench [CPS 89], TAV [GLZ 89], and Auto [BRSV 89]. Most of these tools are behaviourally based and perform proofs automatically. They interpret processes as labelled transition systems and proofs are established by automatically searching the resulting spaces. Automatic proof tools suffer from the "state explosion" problem, and can not coup with infinite state problems. Another disadvantage is that users have little control over the proofs. There is no easy way to perform incremental or interactive proofs.

Most process algebras are equipped with equational axiomatisations. Algebraic proofs work directly on syntactical terms without appealing to internal representation, so there is no danger of "state explosion", and even infinite state problem can be dealt with. Recently proof tools based on algebraic manipulations have emerged. One such system has been developed in Pisa for *CCS* [NIN 89], and another one, *PSF* proof assistant, is in Amsterdam for *ACP* [MV 91]. A major problem with these tools is that each of them is designed only for a specific process algebra. So, for instance, one can not use *PSF* for *CSP* or *LOTOS*. Building a proof system takes considerable efforts as one has to implement the parser, user interface, proof strategy, proof environment management, and so on. Since there are quite a few process calculi around and each is evolving (new operators are emerging, with new axioms characterizing them), it is desirable to have a general tool which allows users to define their own calculi, so that much of the implementation efforts can be expended once and for all.

A well-developed technique for general theorem proving in equational logic is term-rewriting [DJ 89]. But there are some difficulties involved in applying existing term rewriting systems to process algebras:

- Existing term rewriting systems work only on *finite terms*, *i.e.* terms with fixed depths, so using these tools one can only reason about finite processes [Kir 93]. But almost all useful processes are defined recursively. To deal with recursion some forms of induction, such as the *unique fixpoint induction* [Mil 84] or *Scott induction* [Hen 88], are needed. These induction principles are not supported by the existing term rewriting systems.

- Some "equations" in process algebras, such as the *expansion laws*[Mil 89], are not simple equations but rather *equation schemes*, i.e. they are common patterns of infinitely many equations. This is beyond the power of existing rewriting systems which can only handle finite set of equations.

- Some calculi have "indexed operators", such as the parallel operators indexed by a set of actions in LOTOS and CSP. They represent classes of infinitely many operators and as such cannot be handled by existing rewriting systems which only allow finite signatures.

PAM (Process Algebra Manipulator) [Lin 91] is a general proof tool for process algebras using rewriting techniques. The logic it implements is essentially equational logic plus recursion, with some features tailored to the particular requirements of process algebras. At the core of PAM is a rewrite machine which is capable of handling associative and commutative operators. A pattern is provided for defining interleaving (or expansion) laws in various calculi, and applications of these laws are treated as special forms of rewriting. Infinite processes can be defined by mutual recursion, and some forms of induction, such as Scott Induction and Unique Fixpoint Induction, have been built into the system to cope with such recursively defined processes. The syntax for signature definitions is flexible enough to allow "indexed operators".

It is possible in PAM to designate as theorems proved conjectures and then use them in subsequent proofs. This allows users to decompose a big problem into subproblems, prove these separately, and then combine all the small proofs together to establish a proof of the original problem. This is an important feature for any proof tool to be practically useful. Unfinished proofs can be saved into files and then loaded back later to be resumed, so that long proofs can be separated into several proof sections over time. Basic transformation steps can be composed into more powerful commands using a simple tactical language.

One disadvantage of interactive theorem proving is that proofs can get quite tedious, and it is desirable to interface PAM with other automatic proof tools for process algebras so that some parts of proofs can be "submitted" to such tools for automatic verification. An experimental interface from PAM to the Concurrency Workbench has been implemented which allows calls to the Workbench from within PAM for checking various congruence/equivalence relations between CCS agents.

The process algebras which have been successfully defined in PAM include *CCS* [Mil 89], *CSP* [Hoa 85], *ACP* [BK 89] and *EPL* [Hen 88]. We have been experimenting with some small examples such as the Scheduling problem [Mil 89] and Alternating Bit Protocol [BK 89] in different calculi.

The rest of the paper is organized as follows: Section 2 shows how to use the system by giving examples; The meta language for calculus definition is explained in Section 3; Section 4 presents the algorithm for unique fixpoint induction. Section 5 describes the problem definition format and available proof commands. The paper is concluded in Section 6.

2 How to Use The System: An Example

PAM accepts a process calculus definition file, yielding a proof manipulator for the calculus. The meta-language for defining calculi is explained in Section 3 . Here we only take *CCS* as an example to show how to use the system.

2.1 Defining A Calculus

A calculus definition consists of two sections: *signature* and *axiom*. The signature section starts with type declarations:

```
signature
type    Label Action  Process
with    Label < Action
```

Here three types `Label`, `Action` and `Process` are introduced, with `Label` declared as a subtype of `Action`.

After type declarations come operator descriptions:

```
operator
   _ + _ :: Process Process -> Process 120 AC LEFT   -- choice
   _ . _ :: Action Process -> Process 200 RIGHT      -- prefixing
   NIL :: -> Process
   _ \ _ :: Process Action set -> Process   300      -- restriction
   ~ _ :: Label -> Label                             -- inverse
   tau :: -> Action
   _ | _ :: Process Process -> Process 150 AC LEFT   -- parallel
   _ [ _ / _ ] :: Process Label Label -> Process 300 -- renaming
```

Here we have 8 operators defined. As an example, the choice operator `+` is infix, has priority 120, is associative and commutative (AC), and associates to the left. Note that `_` is used as *place holder* to indicate where the actual arguments should go, and `set` is a built-in post-fixing type constructor. Anything after `--` in a line is a comment.

In the axiom section we first list basic axioms of the calculus:

```
axiom
A1   x + x = x
```

```
A2   x + NIL = x
PN   x | NIL = x
R0   (x + y)\A = x\A + y\A
R1   (a.x)\A = a.(x\A)               if not(a in A or ~a in A)
R2   (a.x)\A = NIL                   if a in A or ~a in A
R3   NIL\A = NIL
R4   (x|y)\A = (x\A)|y    if Sort(y) inter (A union (map ~ A)) eq {}
N1   (x + y)[a/b] = x[a/b] + y[a/b]
N2   (c.x)[a/b] = c.(x[a/b])         if not(c eq b) and not(c eq ~b)
N3   (c.x)[a/b] = a.(x[a/b])         if c eq b
N4   (c.x)[a/b] = ~a.(x[a/b])        if c eq ~b
N5   NIL[a/b] = NIL
T1   a.tau.x = a.x
T2   x + tau.x = tau.x
T3   a.(x + tau.y) + a.y = a.(x + tau.y)
```

Each axiom consists of a name, an equation, and possibly a side condition.

A subset of axioms can be designated as an *action algebra* which defines the structural behaviour of actions:

```
action algebra
        ~(~a) = a
```

The expansion law is separated from the ordinary axioms and is written in a particular format since they are treated differently from normal equations:

```
expansion law
let x = a1.x1 + ... + an.xn        y = b1.y1 + ... + bm.ym
then
  (x|y)\A = NIL if sync_move(x,y) eq nil and async_move(x,y) eq nil
  (x|y)\A = Sum(+,async_move(x,y))    if sync_move(x,y) eq nil
  (x|y)\A = Sum(+,sync_move(x,y))     if async_move(x,y) eq nil
  (x|y)\A = Sum(+,async_move(x,y)) + Sum(+,sync_move(x,y))
                                       otherwise
with communication function
                sync(a, b) = tau       if a eq (~b) or b eq (~a)
                async(a) = true        if not(a in A or ~a in A)
```

where `sync`, `async`, `sync_move`, `async_move` and `Sum` are built-in primitives and are explained in Section 3.4.

The rules for computing *syntactic sort* [Mil 89] can be placed after the keywords `sort computation`:

```
sort computation

        Sort(NIL) = {}
        Sort(tau.P) = Sort(P)
        Sort(a.P) = {a} union Sort(P)
```

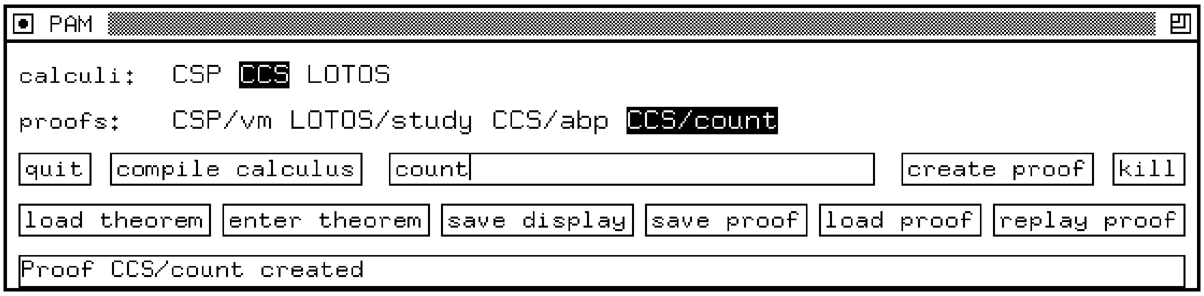

Figure 1: The Top-Level Window

```
        Sort(P + Q) = Sort(P) union Sort(Q)
        Sort(P | Q) = Sort(P) union Sort(Q)
        Sort(P \ A) = Sort(P) diff (A union (map ~ A))
```

2.2 Compiling A Calculus

To compile a calculus, simply enter its name into the small input buffer at the centre of PAM top-level window (Figure 2.2), then click the `compile calculus` button. The compilation usually takes just a few seconds, and, if it succeeds, the calculus' name will appear in the `calculi:` panel. In Figure 2.2 two calculi *CSP* and *CCS* have been compiled, with the *current* one, namely *CCS*, highlighted.

2.3 Proving A Conjecture

Having defined a calculus, one can prove theorems in it. The problems one wants to prove are given in problem definition files. The following is a definition file of the two bit buffer problem in *CCS*:

```
conjecture

        TBB = SYS
where
        TBB = i.TBB1
        TBB1 = i.TBB2 + o.TBB
        TBB2 = o.TBB1

        SYS = (OBBL|OBBR)\{s}
        OBBL = i.s.OBBL
        OBBR = ~s.o.OBBR

rule    H = topunfold ; *EXP ; fold
end
```

The equation after the keyword `conjecture` is the formula to prove. The recursive definitions of the process constants appearing in the conjecture are given in the `where` part. The keyword `rule` introduces a *tactic* which will be explained in Section 5.4.

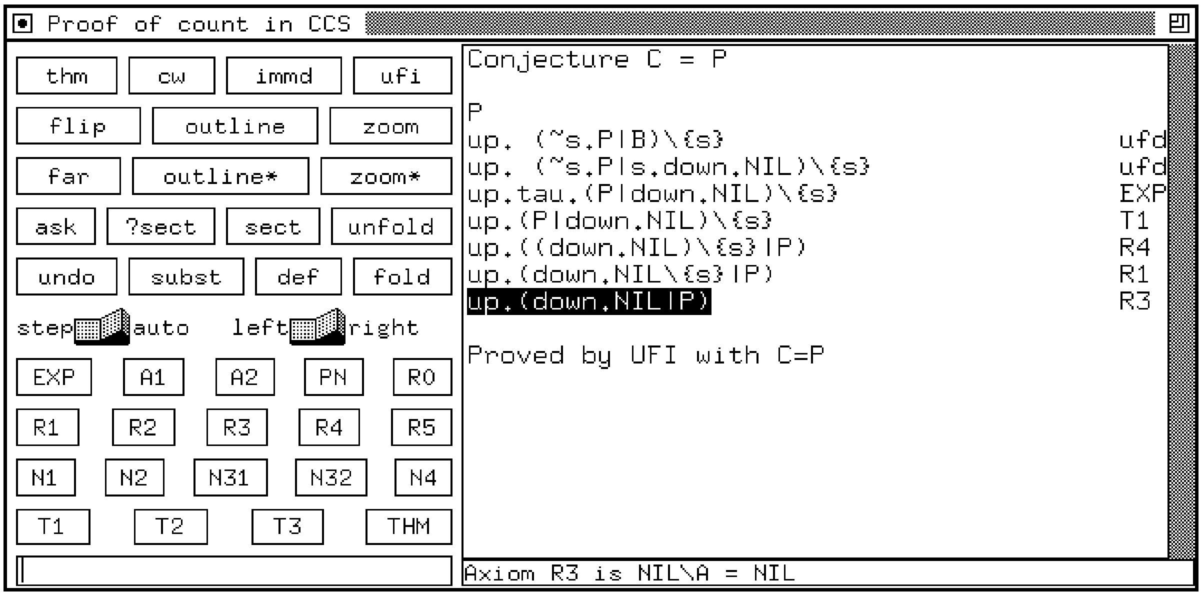

Figure 2: A Proof Window

Here a buffer of capacity two is specified as a *CCS* agent `TBB`. The receiving and sending values are modelled by the input action `i` and output action `o`, respectively. At the beginning it can only perform an `i` action, and if this happens it evolves into `TBB1`, where it can either do an `o` action and then returns to the original state, or do another `i` action and then becomes `TBB2`. In the state `TBB2` only `o` is possible and if it is done the process will go back to `TBB`.

This specification is implemented by a system `SYS` consisting of two one bit buffers `OBBL` and `OBBR`, connected by an internal channel `s`. The individual behavior of `OBBL` or `OBBR` is simply cycling, but as components of `SYS` they have to synchronize on `s` and such synchronization actions are hidden from the outside. The conjecture says `SYS` is a correct implementation of `TBB`.

Proofs are carried out in *proof windows* which are created from problem definition files at the top-level window in a way similar to compiling calculi. Proofs are constructed by clicking suitable command buttons which correspond to *proof steps*. The available proof steps are explained in Section 5. The proof window in Figure 2 contains a complete proof for the two bit buffer problem.

3 The Meta Language

As can be seen in the example of the last section, a calculus definition in this proof system basically consists of *signature* and *axiom* descriptions, with optional sort computation rule specification. The signature describes the syntax of the calculus by listing types and other attributes of the operators, while the semantics of the calculus is characterized by axioms.

3.1 Signature

3.1.1 Types

Type names are listed after the keyword `type`, separated by blanks. The two types `Action` and `Process` must be present in every calculus. There is a pre-defined type `Bool` and a pre-defined post-fixing type constructor `set` (used to form action sets).

It is possible to specify some types as subtypes of others. "T1 < T2" says T1 is a subtype of T2. A blank-separated list of such sub-typing declarations can be placed after the keyword `with` following type declaration.

3.1.2 Operators

A typical operator declaration looks like

```
_ + _ :: Process Process -> Process 120 AC LEFT
```

The symbol `_` is used as a *place-holder* to indicate where the actual arguments should go. So `+` is a binary infix operator, left-associative and commutative, with priority 120.

It is possible to declare *operator schemes*, or *indexed operators*. At the present only associative and commutative operator schemes are allowed, i.e., all parameterized operators must be of attribute `AC`, and arguments other than the first and the last ones are regarded as indexes. For example, the family of parallel operators indexed by sets of actions may be declared as follows:

```
_ |[ _ ]| _ :: Process Action set Process -> Process
                    100 AC LEFT
```

Here a scheme for a family of infix, left-associative and commutative operators is specified. For instance, `|[{a,b}]|` and `|[{c}]|` are two such operators. Each operator in this family will take two processes as arguments and return another process as result.

3.2 Axioms

An axiom is a named equation, or inequation, possibly with a side-condition.

The language for side-conditions is simple, involving equality test `eq` between two actions or two action sets, boolean operations `true`, `false`, `not`, `and` and `or`, and (finite) set operations `in` (membership), `union`, `diff` (difference) and `inter` (intersection). When the rules for sort computation are presented, the operator `Sort` can also be used in side-conditions to compute the syntactic sorts of processes.

A side condition is a boolean expression built up from action names and/or sets of action names using the above operators. It follows the keyword `if` after an equation.

3.3 Action Algebra

A subset of axioms on terms of type `Action` may be grouped under the keywords `action algebra`. These are the laws governing the structural behaviour of actions.

For instance, the action algebra for *CCS* consists of only one equation

```
~~a = a
```

The action algebra for *CSP* is empty, while the action algebra for *SCCS* [Mil 89] is a commutative group.

The equations in the action algebra of a calculus, when it is present, are automatically applied as left-right oriented rewrite rules to evaluate the action expressions in the side-condition every time a conditional equation is used as a rewrite rule. Hence it is crucial for the user to ensure that the action algebra, when left-right oriented, constitutes a confluent and terminating rewrite system (modulo associativity and commutativity).

3.4 Expansion Law

The expansion laws, or interleaving laws, are the laws present in most existing calculi to reduce parallelism to nondeterminism. They are usually formulated in the form of equation schemes which have infinitely many instances. Since standard term rewriting technique work with only finite number of rules, these laws need special treatment.

A process that is a parallel composition of two or more subprocesses can perform two kinds of actions: asynchronized actions performed by individual subprocesses independently from each other, and synchronized actions caused by communications between more than one subprocess. After an asynchronized action the process will behave like the parallel composition of the result of the subprocess performing the action and the other subprocesses (unchanged). After a synchronized action the process will behave like the parallel composition of the result of the subprocesses performing this action and the other subprocesses. Each calculus has its own synchronization mechanism deciding which actions are allowed to occur asynchronously, which have to be synchronized, and the results of synchronizations.

The expansion laws are algebraic formulations of the above ideas. They deal with a process term which is a parallel composition with each component of the composition being a sum of a list of processes prefixed by some actions. Such a composition can then be "expanded" into a process term which comprises sub-terms representing synchronization movement and asynchronization movement. The top level structure of the new term is no longer parallel composition.

The definition of an expansion law in this meta language consists of three clauses: the `let` clause, `then` clause, and `with` clause, as shown in the example of Section 2.

In the `let` clause the form of the components of the parallel composition is specified. It must be the sum of a list of processes prefixed by actions. Ellipsis... is used to make it more comfortable to read. The summation (or *choice*) operator must be associative and commutative.

The `with` clause defines the synchronization mechanism. In this meta language such a mechanism is determined by three parameters: two functions (`sync` and `async`) and one communication style (`handshake` or `broadcast`).

- `async` maps actions to the boolean constant `true`, and may have a side condition attached. The actions satisfying the side condition can occur asynchronously, while the others can not.

- `sync` is the synchronization function. It takes two actions as arguments and, if they satisfy the side condition, gives the action resulting from the communication between them.

- *communication style* decides how the components of a parallel composition participate in communication. There are two communication styles: `handshake` and `broadcast`. The default is `handshake`.

The `then` clause consists of a list of (conditional) equations with identical left hand-sides which are parallel compositions, or restricted parallel compositions, of the process terms specified in the `let` clause. The right hand-sides are the terms into which the left hand-sides will be expanded. The system provides three primitives that can be used in the right hand-side terms.

- `sync_move` takes two processes as specified in the `let` clause, and returns a (possibly empty) list of processes resulting from all possible communications between them. The result of `sync_move` depends on `sync` and the communication style.

- `async_move` takes two processes as specified in the `let` clause, and returns a (possibly empty) list of processes resulting from all possible asynchronous movement of these two processes. The result of `async_move` depends on `async`.

- `Sum` is a term constructor. It takes a summation operator (which must be associative and commutative) and a non-empty list of processes, and returns the sum of these processes.

For technical reasons, the cases when `sync_move` and/or `async_move` are empty must be specified separately. A constant `nil` is provided to test whether they are empty in side conditions.

Usually there would be one expansion law for each parallel operator.

It is possible to characterize parallel operators with finite number of equations if some auxiliary operators are introduced [BK 89], but this would incur exponential rewriting and slow down proofs considerably, because the writing using them lakes the advantage of built-in primitives. But users are, of course, free to design such calculi and use the system to develop proofs in them.

3.5 Sort Computation

The *sort* or *alphabet* information is useful in proofs involving infinite processes [BK 89, Mil 89]. Although the sort of an arbitrary process is uncomputable in general, it is not difficult to calculate the *syntactic sorts* of processes. In PAM the rules for compute syntactic sorts can be listed after the keywords `sort computation`. The top level symbol of the left hand-side of each rule must be the built-in operator `Sort` which has type `Process -> Action set`. When sort computation is enabled (see Section 5.1), an algorithm is invoked to calculate the least sorts determined by these rules.

4 Unique Fixpoint Induction

Infinite processes are defined recursively. Proving conjectures involving recursively defined processes therefore requires induction. Unique fixpoint induction allows one to assert that two process terms are equal if they satisfy the same set of equations. Its practical application usually involves a pair of process terms, one of which is the specification defined by recursive equations, and the other one is the implementation which has been shown to satisfy a set of equations that are structurally the same as (or similar to) the definitional equations of the specification. Applying unique fixpoint induction to prove the equality of two such processes amounts to matching two sets of equations.

For example, in the two bit buffer problem of Section 2, we have the following definition of `TBB`:

```
TBB = i.TBB1
TBB1 = i.TBB2 + o.TBB
TBB2 = o.TBB1
```

It is easy to derive the following equations

```
SYS = i.P
P = o.i.P + i.o.P
```

where `P = (OBBL|o.OBBR)\{s}` is an auxiliary definition. These two sets of equations can be matched with `TBB = SYS`, `TBB1 = P`, `TBB2 = o.p`, so `TBB` and `SYS` can be proved equal by unique fixpoint induction.

The algorithm for unique fixpoint induction is outlined in Figure 2 using pseudo-ML code. It assumes two equation sets are stored in two lists of pairs of terms `spec_defs` and `impl_eqs`, respectively. For instance, for the above example we have

```
spec_defs = [(TBB,i,TBB1),(TBB1,i.TBB2+o.TBB1),(TBB2,o.TBB1)]
impl_eqs = [(SYS,i.P),(P,o.i.P+i.o.P)]
```

Function `lookup` searches such a list for a process term, returning the right component of a pair if the process term is equal (modulo AC) to the left component of the pair, raising exception `Lookup` otherwise. Functions `diff` and `union` compute the difference and union of two sets, represented as lists, modulo AC. Function `match_term` tries to match two terms (modulo AC), returning a list of matchers upon success, raising exception `Match_term` when failed. For instance, `term_match(i.TBB2+o.TBB, o.i.P+i.o.P)` will return `[(TBB,i.P),(TBB2,o.P)]`.

Function `ufi` tries to match the definitional equations for `spec` and the derived equations for `impl`. It returns two lists of pairs of terms, the first of which reports what it has matched and the second one indicates where it failed. The second list will be empty when ufi succeeds, i.e. the two processes can be proved equal by unique fixpoint induction.

With unique fixpoint induction one can prove problems involving infinite state processes. Here is the *counter* problem (in *CCS*) which can be proved in PAM with only a few steps:

```
fun ufi(spec,impl,spec_defs,impl_eqs) =
let fun matching(matched,to_match) =
    let val to_do = diff(to_match,matched)
        fun match_one(sp,im) =
        let val sd = lookup(spec_defs,sp) handle Lookup => sp
            val id = lookup(impl_eqs,im) handle Lookup => im
        in  match_term(id,sd)
        end
    in  if to_do = nil then (matched,[])
        else let val m = map match_one to_do
                 val new_to_match = fold union m []
                 val new_matched = union(matched,to_do)
             in matching(new_matched, new_to_match)
             end  handle Match_term => (matched,to_match)
    end
in  matching([],[(spec,impl)])
end
```

Figure 3: The pseudo-ML code for unique fixpoint induction

```
conjecture

        C = P
where
        C = up.(down.NIL | C)
        P = up.(~s.P | B) \ {s}
        B = s.down.NIL

need sort computation
end
```

As is well-known unique fixpoint induction is unsound. However it is applicable when some condition, called guardedness, is satisfied [Hoa 85, Hen 88, Mil 89, BK 89]. At the present we leave to the users the responsibility of checking these conditions.

5 Proofs

5.1 Problem Definition

To start a proof, one must present the conjecture to the system in a problem definition file. We have already seen an example in Section 2.

The formula (equation or inequation) to prove follows the keyword `conjecture`. The recursive definitions of the process constants, if any, are listed after the keyword `where`. Macros may be defined after the keyword `macro`. They are used to shorten inputs/outputs and have no computational effect. During a proof, any part of the outputs from the system which is identical to the body of a macro definition will be automatically replaced by that macro.

Sort computation can be enabled by the keywords `need sort computation`. It is disabled otherwise.

In some cases extensions to the calculus in question are required for the problem to be proved. At the present two extensions are allowed: extension by defining communication functions and extension by defining fair abstraction rule.

5.1.1 Fair Abstraction Rule

Fair Abstraction Rule [BBK 87] is valid for bisimulation semantics. It can be applied to get rid of τ-loops and is useful in protocol verifications.

There are two ways of looking at fair abstraction rule: it can be regarded as a (scheme of) inference rule, or as a (scheme of) conditional equation. In this proof system we take the second view: fair abstraction rule is used as conditional rewrite rule and applied to transform a term which matches the left hand-side of the conclusion of the rule into the right hand-side when the premise of the rule is satisfied.

The description of fair abstraction rule has two parts: the premise description and the conclusion description. The premise consists of some definitional clauses where a τ-loop occurs, and the conclusion is an equation derived from the premise with the τ-loop removed.

The following is a typical example of the definition of fair abstraction rule (*CCS*):

```
fair abstraction rule

     from x1 = tau.x2 + y1  ...   xn = tau.x1 + yn
     infer x1 = tau.(y1 + ... + yn)
```

where the keyword `from` starts the premise, and the conclusion follows `infer`. In the premise $\mathtt{x1} \xrightarrow{\mathtt{tau}} \mathtt{x2} \xrightarrow{\mathtt{tau}} \cdots \xrightarrow{\mathtt{tau}} \mathtt{xn}$ constitute a τ-loop in the definitions of `x1, ..., xn`, which is removed in the conclusion.

A τ-loop can also occur implicitly because of hiding, as in the following example (*ACP*):

```
fair abstraction rule
    from  x1 = a1.x2+y1 ... xn = an.x1+yn
        with (a1 in A or a1 eq tau) ... (an in A or an eq tau)
    infer H(A,x1) = tau.H(A,y1+ ... +yn)
```

Here we have a `with` clause attached to the premise specifying that each of `a1`, ..., `an` may either be a τ or be hidden in `A`.

At the present these are the only two forms of fair abstraction rule definition allowed in this system.

5.1.2 Action Algebras

In some calculi like *ACP* action algebras, or communication functions, are defined along with problems. This can be done by listing the result action of each pair of actions which can communicate with each other, after the key word *action algebra*, as the following example shows:

```
conjecture

        TBB = SYS

where
        TBB = i.TBB1
        TBB1 = i.TBB2 + o.TBB
        TBB2 = o.TBB1

        SYS = H({c}, E(A, OBBL||OBBR))
        OBBL = i.s.OBBL
        OBBR = r.o.OBBR

macro

        A = {r,s}

action algebra

        s | r = c

end
```

This is *ACP*-version of two-bit buffer problem. In the implementation `SYS`, `OBBL` and `OBBR` are forced to synchronize by defining the communication function `s | r = c`. `s` and `r` are made localized in `OBBL||OBBR` by encapsulation, and finally `c` is made invisible from outside by hiding.

The action algebra given in a problem definition file is regarded as an extension of the action algebra of the calculus (which might be empty).

5.2 Proof Sections

PAM relies heavily on term rewriting techniques. A proof usually consists of several sections, each starts with a process term followed by some terms transformed from it by applying equational reasoning rules or folding/unfolding recursive definitions. Such transformations are invoked by performing the corresponding *proof steps* explained in the following subsection.

5.3 Proof Steps

A proof proceeds when one performs proof steps which can be classified into three groups:

transformation steps The basic transformation steps are rewriting (by directed axioms or theorems), applying expansion law, and folding/unfolding recursive definition.

assertion steps Here we have proof commands such as unique fixpoint induction, proving by transitivity, and proving by the concurrency workbench. The last one is only meaningful for *CCS*. These commands are needed to confirm that a conjecture has been proved so that it can be admitted as theorem by the proof system.

auxiliary steps These include the commands for making auxiliary definitions, commands for opening new sections, and commands for making proved conjectures as theorems so that they can be used in subsequent proofs.

Proof steps are invoked by clicking command buttons in the proof window. At any moment in a proof window there is a unique *current term* which is highlighted. Clicking on the top-level operator of a term selects it as the current term. The transformation steps are applied to current terms.

The behaviour of rewriting commands are controlled by two switches: The left-right switch determines in which direction an axiom is used as a rewrite rule, while the step-auto switch decides if rewriting should be performed for just one-step, or as many times as possible.

5.4 Tactics

The transformation steps as provided are rather primitive. Some combinations of these steps have been found very useful in practical verifications. For instance, unfolding recursive definitions appearing in the top-level of a parallel composition, followed by applying some form of expansion law, then folding recursive definitions, is such a frequently used tactic. Instead of building-in a fixed set of such strategies, PAM provides a tactical language to allow users to define their own tactics targeting at particular problems.

The language is small, basically consisting of sequential composition (`;`) and arbitrary repetition (`*`). For example, the following line

```
rule    TR = *2 T1 ; *{R0 R1 R2 R3 R4}
```

defines a tactic named `TR`. It consists of applying the *CCS* τ-law `T1` twice, followed by applying the set of restriction laws as many times as possible, until none of them is applicable. As another example, the rule `H` defined in the two bit buffer problem 2.3 can easily be recognized as the tactic mentioned in the previous paragraph.

5.5 Help Facilities

Various help facilities are available. The `ask` command button is used to list the names of all axioms and theorems applicable to the current term, in the direction determined by the `left-right` switch.

The other help information can be obtained by clicking the right mouse-button: on a system command button it gives a short description of the functionality of that button; on an axiom button it prints out the axiom; on a rule button it prints out the definition of that tactic; on an identifier (in the proof display sub-window) it will print out its definition body. All help information are output in the small message panel at the right-bottom of the proof window.

5.6 Saving And Loading Proofs

Proofs of large problems may last several days even weeks, and usually can not be finished in one go. A half-way done proof can be saved into a file by the `save proof` button in the top-level window. A saved proof can be loaded back later on by the `load proof` button, in exactly the same status as when it was saved, so the proof can be resumed from that point. In case the calculus definition has been changed since the proof was saved, the `replay proof` command can be invoked to check the validity of the saved proof in the modified calculus. A proof window will be created at the first proof step which is no longer valid. Of cause if all steps used are still valid (for instance when only new operators and axioms have been added to the calculus), the proof window created by `replay proof` will be the same as that created by `load proof`.

6 Conclusions and Related Work

We have described a general process algebra manipulation system which is based on equational axiomatization. It allows the users to define their own process algebras and carry out proofs for problems in the defined calculi. During the proofs terms can be simplified automatically by rewriting, and assertions about recursively defined processes can be verified by induction.

As introduced in this paper PAM is basically a tool for *pure* process algebras in which actions bear no structures, but recently we have been extending it in two directions: *time* and *data*. In the first direction we take the timed language proposed in [Hen 92][Tim 92], generalising the existing PAM expansion law scheme to allow the presence of the *delay-operator*, the main device for describing timed behaviour in that language. The other direction requires more fundamental modification, as adding data to process calculi introduces a binding operator into the calculi, thus making pure equational reasoning (with induction) no longer sufficient. We will report the work on value-passing version of PAM somewhere else.

Attempts have been made to integrate PAM with the Concurrency Workbench. At the moment we have only one direction of such linkage, i.e. to call the Workbench from within PAM, and the experiments gained are encouraging. More efforts are needed to investigate how to cooperate between these two kinds of proof tools so that each of them can take the advantages of the other: the identities proved by algebraic manipulations can be exploited to reduce the state space of behavioural proof systems, while some parts of algebraic proofs can be checked automatically by behavioural tools to reduce the amount of tedious manipulations in algebraic proofs. We believe automatic and interactive tools are complementary to each other. Each has some advantages and disadvantages. Which one to use depends on the nature of the proof to be conducted.

Perhaps the tools that most closely related to *PAM* are the *PSF* proof assistant and the *Pisa* tool. They are also based on axiomatic semantics of process algebras. *PSF* tool also has a graphic interface. But as mentioned in the Introduction, these tools are not parameterised on the underlying process calculi. Each has built into it a specific process algebra, so it is not possible for users to define their own calculi.

Recently some work has been going on using existing general theorem prover based on higher order logic, such as *HOL* and *Coq*, to carry out axiomatisation-based proofs in process algebras ([Nes 91, BG 93]). Exploiting these theorem provers has the advantage that the powerful mechanism of rewriting, mathematical induction and tactics that are built into them are freely available. On the other hand general theorem provers lack facilities specific for process algebras. Encoding process calculi in these provers requires non-trivial efforts (e.g. defining the expansion law for parallel composition and the unfolding law for recursive expressions is far from straightforward [Nes 91]). As these theorem provers do not support reasoning with recursively defined processes, application of the unique fixpoint induction has to be done by manual substitution ([Nes 91]). Also proofs tend to be long and hard to read. For instance the proof file for the alternate bit protocol in *Coq* is as big as 200Kbyte ([BG 93]), while a proof of a similar problem in *PAM* consists of less than 200 lines.

PAM is written in Standard ML. The graphic user interface is implemented using *eXene*, an *X* window system interface for the New Jersey ML compiler [RG 91].

Acknowledgements

Many thanks to Matthew Hennessy for initiating and overseeing this project and detailed comments on an early draft of this paper, to Luca Aceto for help in process algebra and particularly in formulating the expansion law, to Robin Milner for remarks on fair abstraction rule and on the integration with the Concurrency Workbench, to Faron Moller for detailed comments on the paper, to Simon Bainbridge for discussions on the proof system, to Astrid Kiehn and Anna Ingólfsdóttir for their helps during the implementation of PAM.

This project has been carried out with the financial support from the Science and Engineering Research Council of UK, and the ESPRIT II BRA project CONCUR.

References

[BBK 87] Baeten, J.C.M., Bergstra and J.A., Klop, J.W., "On the Consistency of Koomen's Fair Abstrction Rule", *Theoretical Computer Science* 51, pp. 129-176, 1987.

[BG 93] Bezem, M., Groote, J.F., *A Formal Verification of the Alternating Bit Protocol in the Calculus of Constructions*, Report No. 88, Department of Philosophy, Utrecht University, 1993.

[BK 89] Bergstra, J.A., Klop, J.W., "Process Theory Based on Bisimulation Semantics", in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, 1989.

[BRSV 89] Boudol, G., Roy, V., de Simone, R., Vergamini, D., *Process Calculi, From Theory to Practice: Verification Tools*, INRIA Report No 1098, 1989.

[CPS 89] Cleaveland, R., Parrow, J. and Steffen, B., "The Concurrency Workbench", *Proc. of the Workshop on Automated Verification Methods for Finite State Systems*, LNCS 407, 1989.

[DJ 89] Dershowitz, N., Jouannaud, J.-P., "Rewrite Systems", in *Handbook of Theoretical Computer Science* North-Holland, 1989.

[GLZ 89] Godskesen, J.C., Larsen, K.G., Zeeberg, M., *TAV Users Manual*, Internal Report, Aalborg University Centre, Denmark, 1989.

[Hen 88] Hennessy, M., *Algebraic Theory of Processes*, MIT Press, 1988.

[Hen 92] Hennessy, M., *Timed Process Alhebras*, Lecture Notes for the International Summer School on *Program Design Calculi*, Martoberdorf, 1992.

[Hoa 85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Kir 93] Kirkwood, C., Automating (Specification $\equiv$ Implementation) using Equational Reasoning and LOTOS, in *Proc. TOPSOFT'93: Theory and Practice of Software Development*, LNCS 668, 1993.

[Lin 91] Lin, H., *PAM User Manual*, Computer Science Report No. 9/91, Sussex University, 1991.

[Mil 84] Milner, R., A Complete Inference System for a Class of Regular Behaviours, *J. Computer and System Science*, Vol 28, pp. 439-466, 1984.

[Mil 89] Milner, R., *Concurrency and Communication*, Prentice-Hall, 1989.

[MV 91] Mauw, S., Veltink, G.J., "A proof Asisteant for PSF", *Proc. 3rd international Workshop on Computer Aided Verification*, Aalborg, Denmark, LNCS 575, 1991.

[Nes 91] Nesi, M., "Mechanizing a Proof by Induction of Process Algebra Specification in Higher Order Logic", *Proc. 3rd international Workshop on Computer Aided Verification*, Aalborg, Denmark, LNCS 575, 1991.

[NIN 89] De Nicola, R., Inverardi, P., Nesi, M., "Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Specifications", *Proc. Workshop on Automatic Verification Methods for finite State Systems*, LNCS 407, 1989.

[RG 91] Reppy, J. and Gansner, F., *The eXene Library Manual*, AT&T Bell Laboratory, 1991.

[Tim 92] T. Regan, *A Process Algebra for Real-Time Systems.* Ph.D. thesis, University of Sussex, 1992.