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

It has been gradually recognized that computer assistance is essential for the analysis of concurrent systems. There are already 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.

More recently, efforts have been devoted to implementing algebraic proof systems for process calculi. One such system is being developed in Pisa for CCS [NIN 89], and another one is in Amsterdam for ACP [MV 91]. But 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 system 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:

- With pure equational logic one can only reason about finite processes. Infinite processes are defined recursively, and there is no way to handle recursion by rewriting.

- 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 cann't be handled by existing rewriting systems which only allow finite signatures.

PAM (Process Algebra Manipulator) 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 powerful 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.

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 implementation of unique fixpoint induction. Section 5 describes the problem definition format and proof commands available at the present; Finally, future work is outlined 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 postfixing 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.

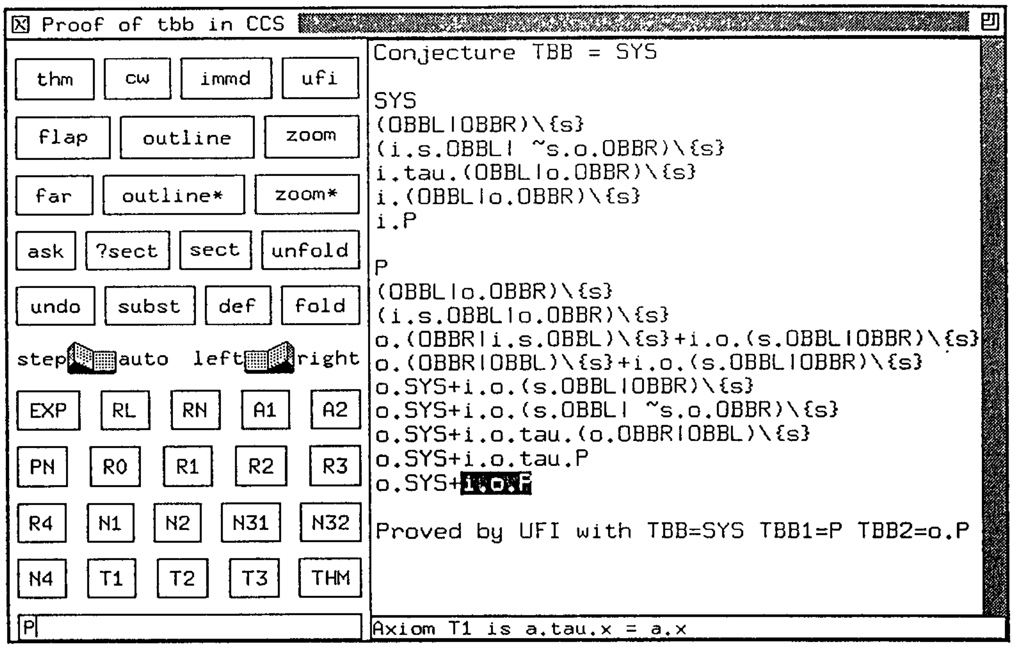
The rules for computing *syntactic sort*[Mil 89] can be placed after the keywords **sort computation**:

```
sort computation

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

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

P
(OBBL|o.OBBR)\{s}
(i.s.OBBL|o.OBBR)\{s}
o.(OBBR|i.s.OBBL)\{s}+i.o.(s.OBBL|OBBR)\{s}
o.(OBBR|OBBL)\{s}+i.o.(s.OBBL|OBBR)\{s}
o.SYS+i.o.(s.OBBL|OBBR)\{s}
o.SYS+i.o.(s.OBBL| ~s.o.OBBR)\{s}
o.SYS+i.o.tau.(o.OBBR|OBBL)\{s}
o.SYS+i.o.tau.P
o.SYS+i.o.P

Proved by UFI with TBB=SYS TBB1=P TBB2=o.P

Axiom T1 is a.tau.x = a.x
```

Figure 1: A Proof Window

```
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
end
```

The system reads in the above problem definition file (assuming the calculus *CCS* has already been compiled), creating a proof window for it. Proofs are constructed by clicking suitable command buttons which correspond to *proof steps*. The available proof steps are explained in Section 5. Figure 1 shows a proof window with 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 optionaly sort computation rule specification.

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 postfixing 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 subtyping 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.

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 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 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 handsides which are parallel compositions, or restricted parallel compositions, of the process terms specified in the **let** clause. The right handsides are the terms into which the left handsides will be expanded. The system provides three primitives that can be used in the right handside 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 some technical reasons, the cases that `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.

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 handside 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

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 match two sets of equations. The algorithm for unique fixpoint induction is outlined in Figure 2 using pseudo-ml code.

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:

```
conjecture

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

need sort computation
end
```

```
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 2: The pseudo-ml code for unique fixpoint induction

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.

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

5.2 Proof Sections

The proof system 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. The proof steps already implemented so far can be classified into three groups:

transformation steps The basic transformation steps are rewriting, 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 window interface. 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 go as far as possible.

6 Conclusions and Future 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.

Only the kernel part of PAM has been implemented. Much more efforts at both design level and implementation level are still needed to make it a practically useful system.

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 indentities 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.

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

[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.

[MV 91] Mauw, S., Veltink, G.J., *A proof Asisteant for PSF.* Programming Research Group, University of Amsterdam, 1991. In this Volume.

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

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

[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.