A Verification Tool for Value-Passing Processes

H. Lin*
Computer Science
University of Sussex
England

Abstract

A proof system is described which is capable of handling processes with value-passing. It implements inference systems that treats data and boolean expressions *symbolically*. Users are allowed to define the syntax and axioms for their own calculi. Proofs are performed by either invoking proof rules or applying equational axioms as rewrite rules. Help facilities are provided to make it easier to conduct proofs.

Keyword Codes: D.1.3; D.2.2; D.2.4
Keywords: Programming Techniques, Concurrent Programming;
Software Engineering, Tools and Techniques; Program Verification

1 Introduction

Both CSP [Hoa85] and CCS [Mil89] were originally proposed as languages for describing systems of processes which can run in parallel and cooperate with each other by transmitting data through communication channels. But in the subsequent years the aspect of data transmition has been somewhat neglected, and "pure" versions of these calculi, so-called process algebras, have been studied extensively, in which data communication has been simplified as synchronizations on signals. The link between the full calculi and their pure, or basic, versions is a translation replacing input actions in the full calculi with summations over all values in the data domain involved.

Accompanying the theoretical development, a number of tools have been built to automate or assist proofs in these calculi. But it is beyond the scope of these tools to deal with full calculi, simply because of the infinity introduced by the translations into basic calculi over non-trivial data domains.

*Current address: Institute of Software, Chinese Academy of Sciences, P.O. Box 8718, Beijing

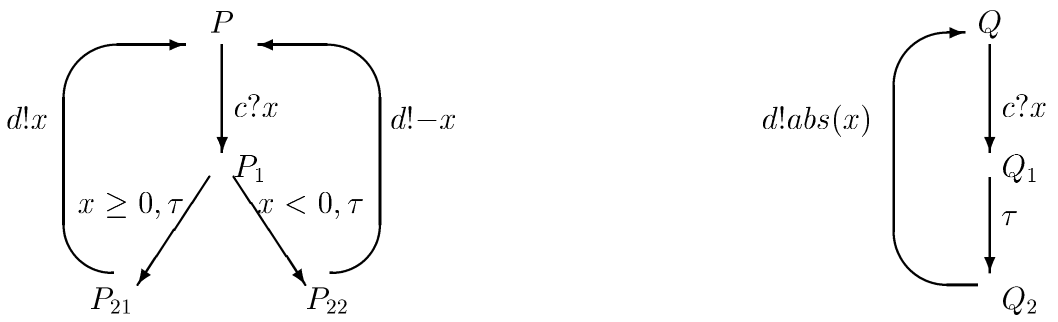

Figure 1: Symbolic Transition Graphs

Recently efforts have been made to develop both semantic and proof theories for process algebras that allow value-passing. Any theory of such calculi will inevitably involve two dimensions – processes and data. But it has been demonstrated that the data aspect can be factored out from processes so that one can concentrate on the concurrent behaviour. The idea is to manipulate data *symbolically*, i.e. do not instantiate input variables or evaluate data/boolean expressions, when analysing value-passing processes. For instance, at the semantic level the notions of *symbolic operational semantics* and *symbolic bisimulations* have been proposed to capture the usual concrete operational semantics and bisimulations so that the infinity caused by instantiating input variables can be avoided. As an example consider the following CCS processes

$$\begin{array}{rcl} P & = & c?x.if\ x \geq 0\ then\ \tau.d!x.P\ else\ \tau.d!{-x}.P \\ Q & = & c?x.\tau.d!abs(x).Q \end{array}$$

where *abs* is the absolute value function on integers. Both P and Q will generate infinite transition graphs according to the standard translation [Mil89]. But the symbolic transition graphs for them are indeed finite, as shown in Figure 1. A symbolic bisimulation between these two processes consists of the following family of boolean indexed relations:

$$\begin{array}{rcl} S^{true} & = & \{(P,Q),(P_1,Q_1)\} \\ S^{x \geq 0} & = & \{(P_{21},Q_2)\} \\ S^{x<0} & = & \{(P_{22},Q_2)\} \end{array}$$

where we have omitted symmetric and identical pairs. As the boolean index for the pair containing P and Q is $true$, we claim that P, Q are bisimulation equivalent. In general, for closed terms $p \sim q$ if and only if $p \sim_s^{true} q$, where $\sim$ is an ordinary equivalence for pure processes, i.e. processes containing no data variables or data expressions, while $\sim_s$ is the symbolic counterpart for value-passing processes [HL92].

A rather surprising fact about the usual axiomatizations of pure CCS is that the only axioms concerning the prefixing operator are those τ-laws, giving that inter-process communication is the main phenomena the calculus was designed to describe. The reason for this is that in pure CCS inputs and outputs have been reduced to atomic actions, so actions do not affect subsequent behaviors of the processes performing them. In other words, actions can be "forgotten" as soon as they have been performed. Proof theoretically this means that the usual congruence rule of equational logic is sufficient for non-τ action prefixing. But this is no longer true if we are to treat value-passing CCS in its own right. Now actions have structures (input and output) which have different meanings; These structures must be characterized by different proof rules.

The work on proof theories for value-passing process algebras was pioneered in [Hen91] where an inference system for testing equivalence for a version of CCS was proposed. It extends pure equational logic by adding rules for different kind prefixing actions (input, output and τ), along with some structural rules. The main philosophy is to concentrate on the process structures by manipulating data and boolean expressions symbolically. Reasoning about data was replaced by appealing to an "oracle". Augmented with a set of rather standard equational laws the inference system was shown to be both sound and complete with respect to testing equivalence over finite processes. More recently this system has been modified so that it works for bisimulation-based equivalences as well [HL93a]. These inference systems do not have infinite rules and is *effective* as far as the process content is concerned, and so provide a sound basis for building theorem provers for value-passing calculi. A modal logic adopting such symbolic approach has been proposed in [HL93b].

In this paper we will describe a verification tool which implements such symbolic inference systems. It is called VPAM, for Value-passing Process Algebra Manipulator. Like its predecessor PAM [Lin91], VPAM is an interactive proof assistant, not an automated tool; Like PAM, VPAM relies on rewriting for equational reasoning and induction for recursively defined processes; Also like PAM, VPAM allows the users to define their own calculi by specifying operators and axioms. But unlike PAM in which proofs are presented as a flattened collection of *sections*, VPAM supports goal-directed proof style. Proofs in VPAM are represented as *goal-trees* which grow in depth when proof rules are applied. In this way overall proof structures are made explicit so that more amenable to user manipulation. The references to the data oracle are factorized in the form of *proof obligations* generated during proof processes. For instance a proof of $\vdash P = Q$ with P and Q defined in the above example will generate the following proof obligation

$$true \models x \geq 0 \vee x < 0$$

The correctness of a proof is relative to the validity of the proof obligations associated with it. These obligations can be submitted to an independent theorem prover for the data domain involved. In this sense this verification tool is "parameterized" on a theorem prover for data, a natural consequence of the facts that our process language is parameterized on a language for data expressions, and the inference rules are parameterized on a data oracle.

The rest of this paper is organized as follows. In the next section we briefly review some related work. The inference systems implemented in VPAM are introduced in Section 3. Section 4 illustrates how to define a calculus in VPAM, while Section 5 is devoted to show how to carry out proofs. The paper is concluded with Section 6 where some future work is also outlined.

2 Related Work

A number of proof tools for process algebras exist, both automatic and interactive. To name a few, in the automatic (or semantics-based) world there are The Concurrency Workbench [CPS89], Auto [SV89] and Tav [GLZ89]; In the interactive (or axiomatization-based) world we have Pisa tools [DIN89], PSF proof assistant [MV91] and PAM [Lin91].

These tools are designed for basic calculi where actions are atomic entities and no data are allowed. They are not capable of dealing with value-passing calculi. Efforts have been made to translate certain version of value-passing CCS into basic CCS so that it can be accepted by existing tools such as the Workbench [Bur91].

Many tools have been developed around the LOTOS language [Lot92]. Among them is a simulator called SMILE which can simulate LOTOS specifications involving data expressions symbolically. But the verification tools can only handle basic LOTOS.

In [JP92] a language for *data-independent* processes is considered which has input and output actions of the form $c?x$, $c!d$ where d is a single data value, but does not allow to test or modify the data received. By using "schematic variables" it was shown that bisimulation equivalence between such processes can be reduced to bisimulation equivalence between processes in basic CCS, so that it can be decided using existing algorithms for basic CCS. Their algorithm has been implemented.

Work on using higher order logic and the general purpose theorem prover HOL to develop an interactive verification tool has been reported in [Nes92]. Their approach is also based on the axiomatisation of behavioural semantics.

We are not aware any existing proof tool other than VPAM that can cope with value-passing process calculi.

3 Inference Systems for Value-Passing Processes

3.1 Symbolic Operational Semantics and Symbolic Bisimulations

The usual operational semantics for value-passing CCS would include the following rule for input prefixing

$$c?x.t \xrightarrow{c?v} t\{v/x\} \qquad \text{for every } v \in Val.$$

$$
\begin{array}{lll}
\alpha.t \xrightarrow{true,\alpha}_L t & & \alpha \in \{\, \tau, c!e \mid c \in Chan, e \in Exp \,\} \\
c?x.t \xrightarrow{true,c?y}_L t\{y/x\} & & y \notin fv(c?x.t) \\
t \xrightarrow{b,\alpha}_L t' & \text{implies} & \text{if } b' \text{ then } t \text{ else } u \xrightarrow{b\wedge b',\alpha}_L t' \\
u \xrightarrow{b,\alpha}_L u' & \text{implies} & \text{if } b' \text{ then } t \text{ else } u \xrightarrow{b\wedge \neg b',\alpha}_L u' \\
t \xrightarrow{b,\alpha}_L t' & \text{implies} & t + u \xrightarrow{b,\alpha}_L t' \\
t \xrightarrow{b,\alpha}_L t' & \text{implies} & t \mid u \xrightarrow{b,\alpha}_L t' \mid u \\
 & & \alpha \in \{\, \tau, c!e \mid c \in Chan, e \in Exp \,\} \\
t \xrightarrow{b,c?x}_L t' & \text{implies} & t \mid u \xrightarrow{b,c?x}_L t' \mid u \\
 & & x \notin fv(u) \\
t \xrightarrow{b,c?x}_L t',\ u \xrightarrow{b',c!e}_L u' & \text{implies} & t \mid u \xrightarrow{b\wedge b',\tau}_L t'\{e/x\} \mid u' \\
t \xrightarrow{b,\alpha}_L t' & \text{implies} & t\backslash c \xrightarrow{b,\alpha}_L t'\backslash c \\
 & & \text{if } \alpha \text{ does not use the channel } c \\
t \xrightarrow{b,c?x}_L t' & \text{implies} & t[d/c] \xrightarrow{b,d?x}_L t'[d/c] \\
t \xrightarrow{b,c!e}_L t' & \text{implies} & t[d/c] \xrightarrow{b,d!e}_L t'[d/c] \\
t \xrightarrow{b,c?x}_L t' & \text{implies} & t[d/a] \xrightarrow{b,c?x}_L t'[d/a] \quad \text{if } a \neq c \\
t \xrightarrow{b,c!e}_L t' & \text{implies} & t[d/a] \xrightarrow{b,c!e}_L t'[d/a] \quad \text{if } a \neq c \\
t\{\underline{e}/\underline{x}\} \xrightarrow{b,\alpha}_L t' & \text{implies} & P(\underline{e}) \xrightarrow{b,\alpha}_L t' \\
 & & \text{if } P(\underline{x}) \Longleftarrow t \text{ is a declaration}
\end{array}
$$

Figure 2: Symbolic Operational Semantics

This rule brings a source of infinity into the semantics when the value domain involved is infinite. To eliminate such infinity we propose *symbolic* operational semantics that does not instantiate input variables and, as a consequence, does not evaluate data/boolean expressions. Figure 2 presents a symbolic operational semantics for full CCS.

Based on such symbolic operational semantics the notion of *late symbolic bisimulation*[1] can then be defined as follows, where $bv(\alpha) is \{x\}$ if $\alpha = c?x$, and is $\emptyset$ otherwise, $fv(b,t,u)$ is the set of free data variables appearing in b, t, and u.

Definition 3.1 A family of symmetric relations over terms $\mathbf{S} = \{\, S^b \mid b \in BExp\}$, indexed by a boolean expression b, is a symbolic bisimulation if

> $(t,u) \in S^b$ implies whenever $t \xrightarrow{b_1,\alpha}_L t'$ with $bv(\alpha) \cap fv(b,t,u) = \emptyset$ then there is a finite collection of booleans B such that $b \wedge b_1 = \vee B$ and for each $b' \in B$ $b' \models b_1$ and there exists a $u \xrightarrow{b_2,\alpha'}_L u'$ such that $b' \models b_2$ and

[1]In the contest of value-passing processes there are two variations of bisimulation equivalences, usually referred to as *early bisimulation* and *late bisimulation* according to [MPW89, HL92]. Here we use the *late* version. For the definition of early bisimulation, see [HL92].

1. if $\alpha = c!e$ then $\alpha' = c!e'$, $b' \models e = e'$ and $(t', u') \in S^{b'}$

2. otherwise $\alpha = \alpha'$ and $(t', u') \in S^{b'}$

Let $\sim_{\mathbf{s}} = \{\sim_s^b\}$ be the largest symbolic bisimulation. $\square$

The following result shows that symbolic bisimulation "captures" the usual ("concrete") bisimulation [HL92]

Theorem 3.2 *For closed terms $t,\ u$, $t \sim_s^{true} u$ if and only if $t \sim u$.*

Since symbolic bisimulation does not suffer from infinite branching caused by instantiating input variables over infinite value domains, it is possible to give it a *finite* proof-theoretical characterization.

3.2 The Inference Rules

The inference system for value-passing process algebras implemented in VPAM is presented in Figure 3, where in the rule EQN σ is any mapping from process variables to process terms. It includes the usual inference rules for equational logic. But there is no congruence rule for the prefixing operator. Instead we have TAU, INPUT and OUTPUT rules for three different forms of prefixing. We also have two structural rules CUT and ABSURD which are needed for proofs to work.

The statements of this inference system are *guarded equations*of the form

$$b \rhd t = u$$

where b is a boolean expression and t and u are two process terms. Its intended interpretation is "t and u are symbolically bisimular over b". In fact let $\mathcal{A}_1,\ \mathcal{A}_2$ be the usual CCS laws for strong bisimulation and observation equivalence, respectively, and $\mathcal{A}_3$ for the equations for testing equivalence as presented in [Hen91], and use $\vdash_i b \rhd t = u$ to mean "$b \rhd t = u$ can be derived from $\mathcal{A}_i$ using the inference rules in Figure 3.2", we then have the following soundness and completeness result for recursion-free processes [HL93a]

Theorem 3.3 *For $i = 1,\ 2$, $\vdash_i b \rhd t = u$ if and only if $t \sim_i^b u$.*

where $\sim_1$ is strong bisimulation and $\sim_2$ is observation equivalence.

Combining with Theorem 3.2 this gives

Theorem 3.4 *Let t, u be closed terms. $\vdash_i$ true $\rhd t = u$ if and only if $t \sim_i u$ for $i = 1,\ 2$.*

$$
\begin{array}{lll}
\text{EQUIVALENCE} & \dfrac{}{true \rhd t = t} \quad \dfrac{b \rhd t = u}{b \rhd u = t} \quad \dfrac{b \rhd t = u, u = v}{b \rhd t = v} & \\[2ex]
\text{CONGRUENCE} & \dfrac{b \rhd t_i = u_i \quad i = 1, 2}{b \rhd op(t_1, t_2) = op(u_1, u_2)} & op \notin \{., ?, !\} \\[2ex]
\text{EQN} & \dfrac{}{true \rhd t\sigma = u\sigma} & t = u \text{ is an axiom} \\[2ex]
\text{RENAMING} & \dfrac{}{true \rhd c?x.t = c?y.t[y/x]} & y \notin fv(t) \\[2ex]
\text{L-INPUT} & \dfrac{b \rhd t = u}{b \rhd c?x.t = c?x.u} & x \notin fv(b) \\[2ex]
\text{OUTPUT} & \dfrac{b \models e = e',\ b \rhd t = u}{b \rhd c!e.t = c!e'.u} & \\[2ex]
\text{TAU} & \dfrac{b \rhd t = u}{b \rhd \tau.t = \tau.u} & \\[2ex]
\text{IF} & \dfrac{b \wedge b' \rhd t = v \quad b \wedge \neg b' \rhd u = v}{b \rhd \text{if } b' \text{ then } t \text{ else } u = v} & \\[2ex]
\text{CUT} & \dfrac{b \models b_1 \vee b_2,\ b_1 \rhd t = u \quad b_2 \rhd t = u}{b \rhd t = u} & \\[2ex]
\text{ABSURD} & \dfrac{}{false \rhd t = u} &
\end{array}
$$

Figure 3: The Inference Rules

If we replace the L-INPUT rule with the following rule

$$
\text{E-INPUT} \qquad \frac{b \rhd \sum_{i \in I} \tau.t_i = \sum_{j \in J} \tau.u_j}{b \rhd \sum_{i \in I} c?x.t_i = \sum_{j \in J} c?x.u_j} \quad x \notin fv(b)
$$

in the inference system, then the above theorem holds for early bisimulation equivalences. See [HL93a] for details.

The corresponding result for testing equivalence over finite processes is proved in [Hen91]:

Theorem 3.5 *Let t, u be closed terms. $\vdash_3 true \rhd t = u$ if and only if $t \sim_3 u$.*

Note that in the premises of some rules appear notations like $b \models b'$ which are not formal statements of the proof system. These are where we call the "oracle" about the data domain involved, and can be "implemented" by calling a subsidiary theorem prover for data. But for simplicity in the current implementation of VPAM they are left as *proof obligations* which may be submitted to a data theorem prover later, or just verified by pencil and paper.

To handle recursive defined processes VPAM also implements recursion induction and unique fixpoint induction. See Section 5 for more details.

4 Calculus Definition

VPAM allows users to define their own operators and axioms. A calculus definition in VPAM consists of *signature* and *axiom* descriptions. Here we use CCS as an example.

The signature consists of type declarations and operator descriptions:

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

Here we introduce three types and 6 operators. 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.

Some operators are pre-defined. These include `_ ? _` which takes a channel and a data-variable yielding an input action, `_ ! _` which takes a channel and a data-expression returning an output action, and the conditional
`if _ then _ else :: Bool Process Process -> Process`.

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
R21  (c?v.x)\A = c?v.(x\A)              if not(c in A)
R22  (c?v.x)\A = NIL                    if c in A
R31  (c!e.x)\A = c!e.(x\A)              if not(c in A)
R32  (c!e.x)\A = NIL                    if c in A
R3   NIL\A = NIL
N1   (x + y)[a/b] = x[a/b] + y[a/b]
N21  (c?v.p)[a/b] = c?v.(p[a/b])        if not(c eq b)
N22  (c?v.p)[a/b] = a?v.(p[a/b])        if c eq b
N31  (c!e.p)[a/b] = c!e.(p[a/b])        if not(c eq b)
```

```
N32  (c!e.p)[a/b] = a!e.(p[a/b])          if c eq b
N4   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.

The expansion laws are equation schemes used to reduce parallelism to nondeterminism. In the setting of value-passing process algebras data transmission is modelled in the expansion laws as textual substitution. So for instance in full CCS the expansion law has the following form

Let X, Y denote $\Sigma_i \alpha_i.X_i$, $\Sigma_j \beta_j.Y_j$, with $FDV(X) \cap BDV(Y) = FDV(Y) \cap BDV(X) = \emptyset$. where $FDV(X)$ and $BDV(X)$ are free data variables and bound data variables in the term X, respectively. Then

$$X \mid Y = \mathit{sync\_move}(X,Y) + \mathit{async\_move}(X,Y)$$

where

$$\begin{aligned} \mathit{sync\_move}(X,Y) = \quad & \Sigma\{\, \tau.(X_i\{e/x\} \mid Y_j) \mid \alpha_i \equiv c?x, \beta_j \equiv c!e \,\} + \\ & \Sigma\{\, \tau.(X_i \mid Y_j\{e/x\}) \mid \alpha_i \equiv c!e, \beta_j \equiv c?x \,\} \\ \mathit{async\_move}(X,Y) = \quad & \Sigma_i \alpha_i.(X_i \mid Y) + \Sigma_j \beta_j.(X \mid Y_j) \end{aligned}$$

In VPAM a format is provided to declare expansion laws as they are treated differently from ordinary equations. The above expansion law can be specified as follows

```
expansion law
let x = a1.x1 + ... + an.xn        y = b1.y1 + ... + bm.ym
then
   (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
        async(c?x) = true               if  not(c in A)
        async(c!e) = true               if  not(c in A)
```

where `sync_move`, `async_move`, `comm` and `async` are built-in primitives and `Sum` is the built-in term constructor for summation. `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. Similarly `async_move` returns a list of processes resulting from all possible asynchronous transitions of these two processes.

VPAM parses a calculus definition file, translating it into internal representation which is used later on when parsing problem definitions and carrying out proofs.

```
type Int Bool
function
   pos _ :: Int -> Bool
process
   Bag ::
   B1 :: Int
   B2 :: Int Int
   C ::
channel
   r w
conjecture
   |- Bag = C | C
where
   Bag = r?x.if pos(x) then B1(x) else Bag
   B1(x) = w!(x+x).Bag + r?y.if pos(y) then B2(x,y) else B1(x)
   B2(x,y) = w!(x+x).B1(y) + w!(y+y).B1(x)
   C = r?x.if pos(x) then w!(x*2).C else C
end
```

Figure 4: Example: Positive-Doubling Bag of Capacity Two

5 Problems and Proofs

Problems submitted to prove are defined in separate files. Most of the syntax for problem definition can be illustrated by the example in Figure 4, where a bag of capacity two is defined. Upon receiving a integer on channel `r`, the bag tests it. Negative numbers will be discarded. The bag can hold up to two positive numbers. When a number is sent out on channel `w` it is doubled. The bag is implemented by two cycling cells running in parallel. Each cell can hold up to one positive number, and the number is multiplied by `2` when it is sent out.

Each problem definition starts with declaring data types and functions. The conjecture to prove follows the keyword `conjecture`. Two standard data types Bool and Int are built-in (so the type declaration Int in this example can be omitted). Not that the integer plus operator are overloaded with the choice process operator which is declared in the process definition file by the user. It is also possible to define macros for abbreviation. Process constants are defined by mutual recursion after the keyword `where`. Note that constants may have data parameters.

The system reads in such a problem definition file, creating a proof window for it, as shown in Figure 5. Initially only the goal `Bag = C|C` was shown. Proofs are proceeded by clicking suitable command buttons.

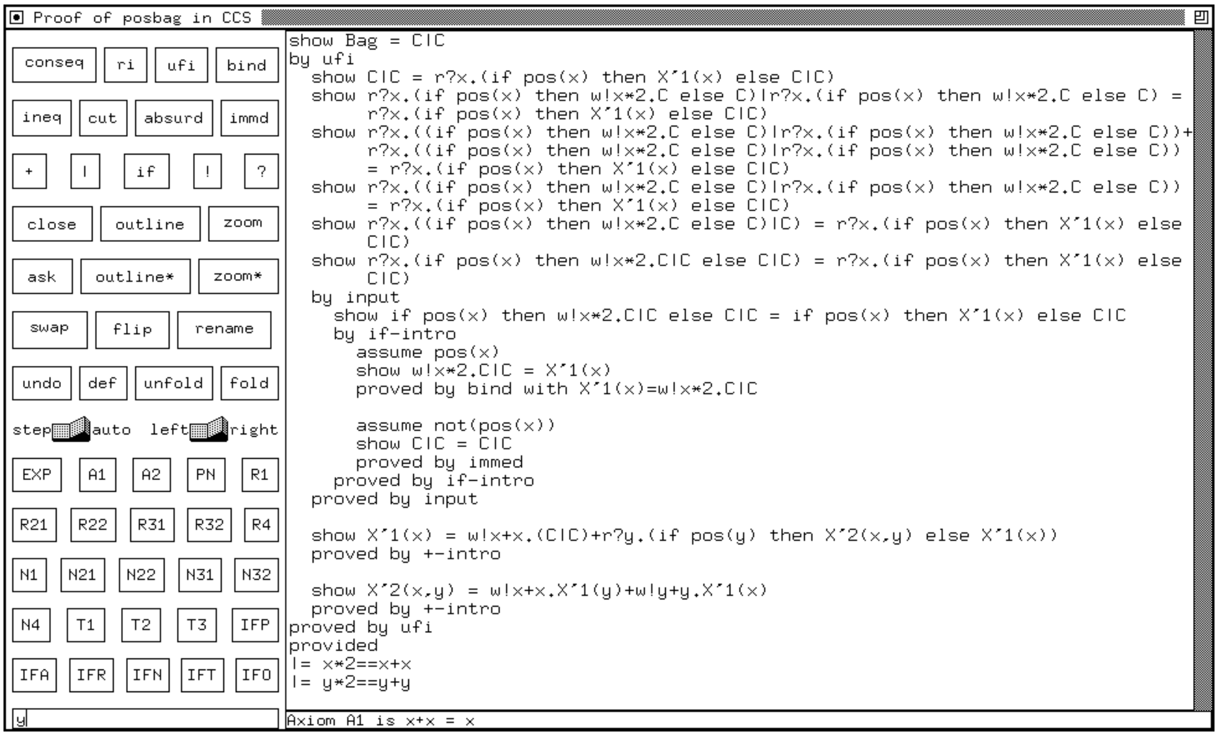

Figure 5: A Proof Window

A well-established technique for presenting proofs in an inference system is *goal-directed* style: the formula (conjecture) to prove is regarded as a goal and one applies to it an inference rule with matching conclusion, generating a subgoal for each premise of the rule (so no subgoal is generated if the rule is an *axiom*, a rule with no premise, in this case the goal is proven); the subgoals are then "hit" in a similar way. A goal is proven if all its subgoals are.

On the other hand, equational reasoning, i.e. replacing equal for equal, is better implemented by *rewriting* which works on *terms* rather than formulae: a subterm of one side of a goal equation gets rewritten to a new term by invoking a directed equation as a *rewrite rule*, resulting in a new goal equation.

As the inference system that underlines VPAM consists of inference rules as well as equations, proofs in VPAM are displayed in a mixed way: invoking inference rules is goal-directed, while applying equations is implemented as rewriting.

The left half of a proof window (see Figure 5) is a command panel where there is a command button for each inference rule. For example the buttons $\boxed{\text{if}}$, $\boxed{?}$ and $\boxed{!}$ correspond, respectively, to the rules IF, INPUT and OUTPUT in Figure 3. Two switches (*auto-step* and *left-right*) control the behavior of the rewrite buttons (below them) which are labelled by axiom names appearing in the calculus definition file. Proofs are displayed on the right subwindow.

In a proof tree there is always a unique term, called *current term*, which is highlighted. The equation containing the current term is called the *current equation*. Click on a rewrite button will cause VPAM to apply the named axiom as a rewrite rule, in the direction controlled by the *left-right* switch, to reduce the current term, generating a new equation. Click on a inference rule button will cause VPAM to apply the rule to the current equation, generating new subgoals. Invoking proof commands for the inference rules involving data domain oracle, such as `output` and `absurd`, will generate *proof obligations* which will be appended at the end of the proof, after the keyword `provided`.

Among the command buttons it is worth mentioning the `ufi`, for unique fixpoint induction. To apply it, the left-handside of the current goal must be a recursively defined identifier. A subgoal will be created for each of the relevant definition clauses. In case there are more than one definition clauses, the system will automatically generate some "unknown" identifiers, start with `X'` and followed by numbers. These unknowns must be bound to process terms later by the `bind` button. After being bound to terms these `X'`s become known and can be unfold/fold just like ordinary identifiers. The `ri` (for recursion induction) command works in a similar way.

Our experiments with PAM shows that "help" facilities are very useful in practice. VPAM provides various ways to get helping information. For instance, click on the `ask` button will list all the axiom names which are applicable to rewrite the current term, in the direction determined by the *left-right* switch. Click the right mouse-button on a command button will give a short description of the command, while click on an axiom button will print out the named equation.

As proofs may get very long and the space for displaying them is limited, we provide two ways to abstract away some proof details, making it easier to grasp the overall proof structures. `Close` a selected goal (a goal is selected by clicking the middle mouse-button on it) will cause the whole sub- proof tree corresponding to this goal, except for the goal-equation, not to be displayed. `Outline` a goal will make the intermediate equations generated by rewriting disappear, without affecting its subgoals. The main difference between `close` and `outline` is: proof can still go on for an outlined goal, but not for a closed goal.

The proof window in Figure 5 contains a complete proof for the positive bag problem, with the second and third sub-goals of the top level goal closed to make it fit into a page.

6 Conclusion

We have presented a proof tool for value-passing processes. It is based on a "symbolic" inference system which, when augmented with well-known equational axioms, is sound and complete with respect to the corresponding semantic equivalences. Allowing user-defined operators and axioms makes it is very easy to change or extend a calculus. With the help of a window interface the tool is easy to use.

The current implementation of VPAM is preliminary. Although the syntax is flexible, it is still limited to CCS-like languages, since we have made two action constructors ? and ! built into the system. As a future research topic we would like to investigate the possibility to extend VPAM for other languages such as CSP and LOTOS.

One of our design goals is to separate reasoning about data from reasoning about processes as much as possible. During a proof the knowledge needed about data domain is extracted as proof obligations. We are interested in cooperating with some suitable data theorem prover to submitted such proof obligations to it for verification.

Acknowledgements Thanks to Matthew Hennessy for initiating and overseeing this project and Alan Jeffrey for reading a draft version and correcting many errors in English.

This work is financially supported by Esprit BRA project CONCUR2.

References

[Bur91] G. Burns. A language for value-passing CCS. Technical Report ECS-LFCS-91-175, LFCS, University of Edinburgh, August 1991.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *Proceedings of the* 9^{th} *International Symposium on Protocol Specification, Testing and Verification*, North Holland, 1989.

[DIN89] R. DeNicola, P. Inverardi, and M. Nesi. Using the axiomatic presentation of behavioural equivalences for manipulating CCS specifications. In *Proc. Workshop on Automatic Verification Methods for finite State Systems*, number 407 in Lecture Notes in Computer Science, 1989.

[GLZ89] J. Godskesen, K. Larsen, and M. Zeeberg. Tav user manual. Report R89-19, Aalborg University, 1989.

[Hen91] M. Hennessy. A proof system for communicating processes with value-passing. *Formal Aspects of Computing*, 3:346–366, 1991.

[HL92] M. Hennessy and H. Lin. Symbolic bisimulations. Technical Report 1/92, Computer Science, University of Sussex, 1992.

[HL93a] M. Hennessy and H. Lin. Proof systems for message-passing process algebras. Technical Report 5/93, Computer Science, University of Sussex, 1993.

[HL93b] M. Hennessy and X. Liu. A modal logic for message-passing processes. Technical Report 3/93, Computer Science, University of Sussex, 1993.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[JP92] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 1992. to appear. Also available as SICS research Report R-89/8908.

[Lin91] H. Lin. PAM: A process algebra manipulator. In *Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 136–146. Springer–Verlag, 1991.

[Lot92] Third lotosphere workshop, Pisa. 1992.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile proceses, part i. Report ECS-LFCS-89-86, University of Edinburgh, 1989.

[MV91] S. Mauw and G. Veltink. A proof assistant for PSF. In *Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 158–168. Springer–Verlag, 1991.

[Nes92] M. Nesi. A formalization of the process algebra CCS in higher order logic. Report Report No. 278, Computer Laboratory, University of Cambridge, 1992.

[SV89] R. De Simone and D. Vergamimi. Aboard auto. Report RT111, INRIA, 1989.