On Implementing Unique Fixpoint Induction for Value-passing Processes

H. Lin*
Laboratory for Computer Science
Institute of Software, Chinese Academy of Sciences
P.O Box 8718, Beijing 100080
E-mail: lhm@ios.ac.cn

Abstract

We examine the possible pitfalls in formulating the unique fixpoint induction proof rule in the setting of value-passing process calculi and describe how this rule is implemented in the verification tool *VPAM*. An argument is also given to justify the implementation.

1 Introduction

Computer-aided verification of concurrent systems has attracted considerable research efforts in the past decade. Many proof tools based on the theoretical results in process algebra have been developed and widely used: The Concurrency Work Bench [CPS89], TAV [GLZ89], AUTO [SV89], The Pisa tool [DIN89], PAM [Lin91], and LOTOS tools [BBV92], to name just a few. Some of these tools are automatic and work by searching state space generated according to operational semantics. Some are interactive and rely on direct manipulation of syntactic terms. These verification tools are only for *pure* process algebras where data exchange between processes is reduced to synchronization on signals. To apply them to problems involving value-passing one has to first transform the problems into pure calculi. Even for finite value domains such transformation could cause exponential explosion of states. When the value domains are infinite the problems would become infinite and thus beyond the scope of these tools.

The "symbolic" semantic theories for value-passing processes have recently been advocated, with avoiding such infinity in mind [HL92]. "Symbolic" style proof systems have also been formulated, allowing to reason about semantic equivalences at a syntactic level [HL93]. These proof systems are finitary in nature and are therefore

*Supported by the President Fund of Chinese Academy of Sciences, the National Science Foundation of China, and the EU KIT project SYMSEM.

amenable to implementation. In fact we have developed the proof assistant *VPAM* (for Value-passing Process Algebra Manipulator) [Lin93] based on such proof systems. The aim of this paper is to explain the way *VPAM* deals with *recursion*, a process construction needed in any non-trivial problems.

Let us start by considering a simple example (in full-*CCS* syntax):

$$\begin{array}{lcl} P & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ P'(x)\ \mathit{else}\ P \\ P'(x) & = & c!x.P \\ \\ Q & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ Q'(x)\ \mathit{else}\ Q \\ Q'(x) & = & c!|x|.Q \end{array}$$

where the data domain is integer and $|x|$ means the absolute value of x.

Clearly $P = Q$ when $=$ is interpreted as bisimulation equivalence. What kind inference rules do we need in order to infer $P = Q$? Since P, Q are recursively defined, some form of induction is required. The most-widely used induction rule in the process algebra community is *unique fixpoint induction*. In pure process algebras this rule has the form:

$$\frac{S_i = T_i[S_k/R_k|1 \leq k \leq m], \quad 1 \leq i \leq m}{S_i = R_i} \quad \begin{array}{l} R_i = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{1}$$

The premises of this rule verify that $\tilde{S}$ is a fixpoint of $\tilde{T}$. By definition $\tilde{S}$ is also a fixpoint of $\tilde{T}$. Since $\tilde{T}$ is guarded it has only one fixpoint. Hence the conclusion. But this rule is not applicable to our example because it does not deal with data variables and data expressions. To solve our problem we need to generalise it to the setting of value-passing processes.

Rule (1) uses process substitution $[S_k/R_k|1 \leq k \leq m]$ where S_k:s are process terms and R_k:s process identifiers. When applied to a term T it replaces R_k with S_k. This is fine for pure calculi because in such calculi a process identifier is a process term on its own-right, so replacing R_k with S_k in T is guaranteed to result in a syntactically well-formed term. In value-passing calculi, however, process identifiers may take data parameters and process terms may contain free data variables, so we can not simply substitute terms for identifiers. For instance applying substitution $[P'(x)/Q']$ to term $c?y.Q'(y)$ would result in $c?y.P'(x)(y)$ which is syntactically ill-formed. What is needed here is a device to take care of correct parameter-passing for substitutions. For this we can use lambda *abstraction* to close up substituting terms and *application* to realise parameter-passing: $T[\lambda\tilde{x}A/B]$, where $\tilde{x}$ is the set of free data variables of A, is the result of substituting $\lambda\tilde{x}A$ for identifier B in T followed by β-conversion, so that the $\tilde{x}$ in A get replaced by the actual parameters of B-occurences. For example

$$\begin{array}{cl} & (\mathit{if}\ x \geq 0\ \mathit{then}\ c!x.P(x)\ \mathit{else}\ c?z.P(z))[\lambda y Q(y)/P] \\ = & \mathit{if}\ x \geq 0\ \mathit{then}\ c!x.(\lambda y Q(y))(x)\ \mathit{else}\ c?z.(\lambda y Q(y))(z) \\ = & \mathit{if}\ x \geq 0\ \mathit{then}\ c!x.Q(x)\ \mathit{else}\ c?z.Q(z) \end{array}$$

With this notation a naive generalization of (1) to the value-passing setting might look like

$$\frac{S_i = T_i[\lambda\tilde{x}_k S_k/R_k|1 \leq k \leq m], \quad 1 \leq i \leq m}{S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{2}$$

Applying this rule to $P = Q$ in our example requires to prove the following two equations

$$\begin{array}{lcl} P & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ P'(x)\ \mathit{else}\ P \\ P'(x) & = & c!|x|.P \end{array}$$

The first equation can be settled immediately, but the second gives rise to (after unfolding $P'(x)$)

$$c!x.P = c!|x|.P$$

which can not (and should not) be proved.

It is not difficult to see what went wrong: $P'(x)$ is not bisimilar to $Q'(x)$ for *all values* of x; they are bisimilar only when $x \geq 0$ (i.e. $P'(x) \sim^{x\geq 0} Q'(x)$, using symbolic bisimulation notation). This is sufficient to establish the bisimilarity between P and Q, because in their definitions $P'(x)$ and $Q'(x)$ *appear only in contexts satisfying* $x \geq 0$.

To take such context information into account, instead of pure equations we use *conditional* equations of the form

$$b \rhd T = U$$

where b is a boolean expression on the value domain, as judgments of our inference system. Intuitively it means T and U are equivalent under the constraint b on the free data variables appearing in them. When $b \equiv \mathit{true}$ it becomes unconditional and can be abbreviated as $T = U$. The fact that $P'(x)$ and $Q'(x)$ are bisimilar over $x \geq 0$ can be expressed as

$$x \geq 0 \rhd P'(x) = Q'(x).$$

With conditional equation we could reformulate (2) as

$$\frac{b_i \rhd S_i = T_i[\lambda\tilde{x}_k S_k/R_k|1 \leq k \leq m], \quad 1 \leq i \leq m}{b_i \rhd S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{3}$$

Unfortunately this is unsound. Here is a counter-example: Let

$$\begin{array}{lcl} P_1 & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ P_1'(x)\ \mathit{else}\ P_1 \\ P_1'(x) & = & c!x.c?y.P_1'(y) \\ & & \\ Q_1 & = & c?x.\ \mathit{if}\ x \geq 0\ \mathit{then}\ Q_1'(x)\ \mathit{else}\ Q_1 \\ Q_1'(x) & = & c!|x|.c?y.Q_1'(y) \end{array}$$

Using (3) one can derive

$$x \geq 0 \rhd P_1'(x) = Q_1'(x) \tag{4}$$

This is wrong because although the first outputs from $P_1'(x)$ and $Q_1'(x)$, which are x and $|x|$ respectively, are equal due to the condition $x \geq 0$, after the input action $c?y$ further outputs from the two processes, which are y and $|y|$ now, can no longer match under this condition. This time the problem is with the premises of (3): The conclusion only asserts $S_i = R_i(\tilde{x}_i)$ *under the condition* b_i, but in the premise $\lambda\tilde{x}_i S_i$ are substituted for R_i *unconditionally.*

How to express the idea that S_i is substituted for $R_i(\tilde{x}_i)$ only over the value space constrained by b_i? We can use $\lambda\tilde{x}_i(b_i S_i)$ ($b_i S_i$ is a shorthand for *if* b_i *then* S_i *else NIL*) instead of just $\lambda\tilde{x}_i S_i$ as the substituting abstraction. This ensures that S_i will be constrained by b_i after substitution. With this observation one might change rule (3) to

$$\frac{b_i \rhd S_i = T_i[\lambda\tilde{x}_k(b_k S_k)/R_k | 1 \leq k \leq m], \quad 1 \leq i \leq m}{b_i \rhd S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{5}$$

Now (4) is no longer derivable. But the new rule creates a different problem. To see this let us change the definition clause for $P_1'(x)$ in the above example to

$$P_1'(x) = c!x.c?y.\textit{if } y \geq 0 \textit{ then } P_1'(y) \textit{ else } NIL$$

Apparently $P_1'(x)$ and $Q_1'(x)$ are not bisimilar over $x \geq 0$. But applying rule (5) to the goal $x \geq 0 \rhd P_1'(x) = Q_1'(x)$ results in the subgoal

$$\begin{aligned} x \geq 0 \rhd P_1'(x) &= (c!|x|.c?y.Q_1'(y))[\lambda x(x \geq 0 P_1'(x))/Q_1'] \\ &\equiv c!|x|.c?y.\textit{if } y \geq 0 \textit{ then } P_1'(y) \textit{ else } NIL \end{aligned}$$

which can be proved after unfolding $P_1'(x)$ with its new definition. So what is wrong with rule (5)? The problem is that the premises of this rule only checks whether $b_i S_i$:s constitute a fixpoint of T_i:s over b_i, but does not do so for R_i:s. Though by definition $\tilde{R}$ is a fixpoint of $\tilde{T}$, this does not imply that $b_i R_i$:s constitute a fixpoint of T_i:s over b_i:s. In this example one does not have

$$\begin{aligned} x \geq 0 \rhd Q_1'(x) &= (c!|x|.c?y.Q_1'(y))[\lambda x(x \geq 0 Q_1'(x))/Q_1'] \\ &\equiv c!|x|.c?y.\textit{if } y \geq 0 \textit{ then } Q_1'(y) \textit{ else } NIL \end{aligned}$$

To make the rule sound we must check such a "partial fixpoint" property for $\tilde{R}$ as well. This can be achieved by employing more premises:

$$\frac{\begin{array}{l} b_i \rhd S_i = T_i[\lambda\tilde{x}_k(b_k S_k)/R_k | 1 \leq k \leq m], \\ b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda\tilde{x}_k b_k R_k/R_k | 1 \leq k \leq m], \end{array} \quad 1 \leq i \leq m}{b_i \rhd S_i = R_i(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array} \tag{6}$$

This rule is sound, but has the disadvantage of doubling the number of premises. This is particularly unsatisfactory from the implementation point of view. The *VPAM* implementation has avoided such duplication by exploiting "context conditions" implicitly associated with process identifiers. In the rest of this paper we shall explain how this can be achieved.

The next section briefly describes how proofs are constructed in *VPAM*, setting up the necessary background for further discussion. Section 3 explains the implementation of unique fixpoint induction in *VPAM*. It also includes a correctness proof for the implementation. The paper is concluded with Section 4.

Some familiarity with value-passing process calculi such as full-*CCS* ([Mil89]) is assumed.

2 Proofs in *VPAM*

VPAM is an interactive verification tool for value-passing process calculi. It is an extension of *PAM* ([Lin91]) which can only deal with pure process algebras. *VPAM* implements a "symbolic" style proof system for value-passing processes, extending those proposed in [Hen91, HL93] for recursion-free languages with a version of unique fixpoint induction for value-passing calculi. The proof system consists of a set of inference rules together with some standard equational axioms. The inference rules, as listed in the Appendix A, are built into *VPAM* while axioms are provided by the users, allowing different equivalences to be reasoned about. The syntax of the calculus one wants to work with is also definable. Thus *VPAM* is a *parameterisable* tool. Both the syntax and axioms of a calculus are defined in a single file which can be compiled in *VPAM*'s top-level window. We will not get into details of calculus definition and compilation here. The interested readers are referred to [Lin93]. In the following discussion we will use full-*CCS* ([Mil89]) as the example language. So . is action prefixing, `+` external choice, `|` parallel composition, `if _ then _ else _` conditional, `c?x` input action, `c!e` output action, *NIL* the empty process, *etc.*. The full-*CCS* axioms, as defined in *VPAM*, are listed in Appendix B. We shall write $\vdash b \rhd T = U$ if $b \rhd T = U$ can be derived from the proof system. The set of free data variables of T will be denoted by $fv(T)$.

The problems to be submitted to *VPAM* for verification are described in *problem definition files*. An example follows:

```
function

      ABS _ :: Int -> Int

process

      P1 :: Int
      Q1 :: Int
      P2 :: Int Int
      Q2 :: Int Int

channel  r  w

conjecture
```

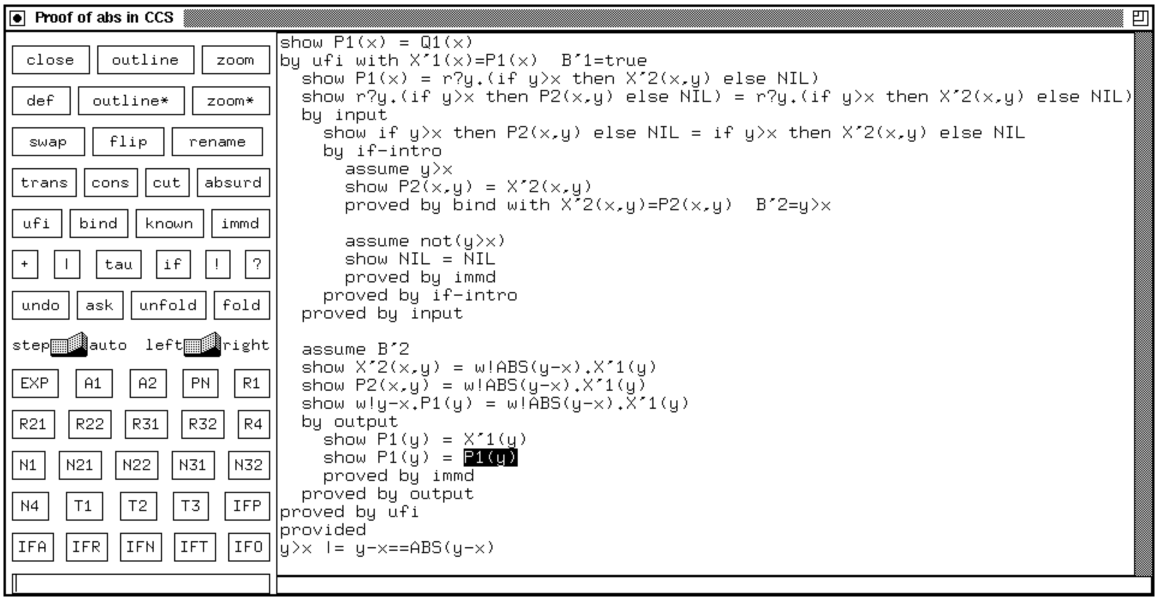

Figure 1: A Proof Window

```
        |- P1(x) = Q1(x)

where
        P1(x) = r?y. if y > x then P2(x,y) else NIL
        P2(x,y) = w!(y-x).P1(y)
        Q1(x) = r?y. if y > x then Q2(x,y) else NIL
        Q2(x,y) = w!ABS(y-x).Q1(y)
end
```

`Int` is a built-in type with the usual operators `>`, `<`, `+`, `-`, *etc.*. `ABS` is the absolute value function. `P2 :: Int Int` declares that `P2` is a process identifier with two integer parameters. Data parameters to a process identifier must be distinct variables. Recursive definitions are listed in the `where` part. The statement to prove is given after the keyword `conjecture`.

Given such a problem definition file *VPAM* creates a *proof window* for it (Figure 1). The left half of a proof window (see Figure 1) is a command button 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 the Appendix A. Two switches (*auto-step* and *left-right*) control the behavior of the rewrite buttons (below them) which are labeled by axiom names appearing in the calculus definition file. Rules and axioms are invoked when the corresponding buttons are pressed. Proofs are displayed on the right subwindow. A conditional equation $b \triangleright T = U$ is displayed as

```
    assum b
```

```
show T = U
```

As the inference system underlining *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 (and does not generate subgoals). Some inference rules of this proof system use the semantics implication relation $\models$ between boolean conditions. These are where we appeal to an "oracle" to answer questions about the data domain. This is is a consequence of the fact that our process languages are parameterised on languages for data and boolean expressions. The inference system treats these expressions "symbolically", i.e. as uninterpreted strings. In fact, one of the main design ideas behind the proof system is to separate data reasoning from process reasoning as much as possible. For example, in order to infer $x = y \rhd c!(x-y).P(x) = c!0.P(x)$ we need to know $x = y \models x - y = 0$. Since we do not want to be involved in reasoning in any particular data domain (integer in this case), this question is left to the oracle. In *VPAM* the proof commands corresponding to these rules, such as `output` and `absurd`, will generate *proof obligations* appended at the end of the proof, after the keyword `provided`. Such proof obligations represent an interface to an auxiliary data domain theorem prover which "implements" the oracle.

In a proof window there is always a unique term, called the *current term*, which is highlighted. The equation and the subgoal containing the current term are called the *current equation* and *current goal* respectively. Each goal has an *assumption* which is a list of booleans (interpreted as connected by conjuncts). The *current global assumption* is the conjunction of the assumption of the current goal together with the assumptions of *all its ancestors*. *Goal equation* is the first equation of a goal.

3 UFI in VPAM

3.1 The Implementation

In *VPAM* the unique fixpoint induction is implemented by three related proof commands: **ufi**, **bind** and **known**. We assume parameters of any process identifiers are distinct data variables.

To apply **ufi**, the right hand side of the current equation must be a recursively defined identifier, possibly with parameters. For easy reference suppose the current conditional equation is

$$b_1 \rhd S_1 = R_1(\tilde{x}_1) \tag{7}$$

and the definition clauses relevant to R_1 are

$$R_i(\tilde{x}_i) = T_i \qquad 1 \leq i \leq m.$$

where the set of identifiers appearing in the T_i:s is $\{\, R_i \mid 1 \leq i \leq m \,\}$. When the **ufi** command is invoked the system automatically generates a pair of "unknowns" for each of these identifiers, one is "process unknown" of the form `X'n` and the other is "boolean unknown" of the form `B'n`, where `n` is a number. `X'1` is bound to the left hand side of the current equation and `B'1` to the global assumption. A subgoal is created for each of the relevant definition clauses in the following manner: The right hand side of the ith subgoal equation is the result of substituting $\lambda\tilde{x}_1 S$ for R_1 and X_k for R_k, $2 \leq k \leq m$, in T_i. The left hand side of the first subgoal equation is S, and the left hand side of the ith subgoal equation for $i \geq 2$ is $X_i'(\tilde{x}_i)$. The goal assumption of the first subgoal is empty and the ith subgoal assumption is B_i'. So for the absolute-value problem in the previous section, applying **ufi** to the top level goal `P1(x) = Q1(x)` will result in the following configuration:

```
show P1(x) = Q1(x)
by ufi with X'1(x)=P1(x) B'1=true
  show P1(x) = r?y.if y>x then X'2(x,y) else NIL

  assume B'2
  show X'2(x,y) = w!ABS(y-x).X'1(y)
```

The "unknowns" generated by the **ufi** command must be bound to some terms for the proof to be completed. In fact the **ufi** command does the initial binding: it binds R_1 to $X'1$ and the current global assumption to $B'1$. This is the "first push" allowing the proof to go ahead. Other "unknowns" must be bound using the **bind** command. To apply **bind**, one side of the current equation must be a "process unknown" which has not been bound before, and the other side must not contain any unknowns. **bind** binds this "process unknown", say $X'n(\widetilde{(x)}_n)$, to the term at the other side of the equation. At the same time it also binds $B'n$, the "boolean unknown" associated with $X'n$, to the current global assumption. This boolean will be referred as the *binding condition* of $X'n$.

An "unknown" that has been bound by **bind** can be "made known" using the command **known**. Similar to the case of **bind**, to apply **known** the current side of the current equation must be a "process unknown". **known** then replaces this "unknown" with the term previously bound to it by **bind**. At the same time **known** also generates a proof obligation which asserts that the current global assumption must imply the binding condition of this unknown. More precisely, suppose the current term is $X'n(\tilde{x}')$, and $X'n(\tilde{x}_n)$ and $B'n$ have previously bound to Sn and b, respectively. Then **known** will unfold $X'n(\widetilde{(x)}')$ to $Sn[\tilde{x}'/\tilde{x}_n]$ and generate $b' \models b[\tilde{x}'/\tilde{x}_n]$ as a proof obligation, where b' is the current global assumption. As said in the previous paragraph the **ufi** command automatically binds $X'1$ to R_1. It also automatically makes $X'1$ known: that is why the left hand side of the first subgoal equation is R_1 instead of $X'1$. The proof obligation of this implicit application of **known** is "the current global assumption implies the current global assumption" which holds trivially (hence is not generated).

3.2 Correctness of the Implementation

The *VPAM* implementation of the unique fixpoint induction, as described above, generates only m subgoals for the goal (7) which involves m mutual-recursive definition clauses while the unique fixpoint induction rule (6) has twice as many premises. In fact *VPAM* does not generates subgoals to check the context conditions of the identifiers, i.e. the m premises

$$b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda \tilde{x}_k b_k R_k / R_k | k] \quad 1 \leq i \leq m$$

in rule (6). They are validated as a by-product. In this subsection we shall justify that the three commands **ufi**, **bind** and **known** correctly implement rule (6).

By α-conversion in the following discussions we assume different input actions in a term use different variables which are also different from the free variables of the term. We also assume the "unknowns" X'_n and B'_n are not used in the problem definitions.

An atom is a process term of the form $P(\tilde{x})$ where P is an identifier or an unknown. An occurrence of an atom A in a process term T is designated by a box surrounding it. When $A \equiv P(\tilde{x})$ we also call $\boxed{\mathrm{A}}$ a P-occurrence. The *context condition* of an occurrence $\boxed{\mathrm{A}}$ in T, denoted $C_T^{\boxed{\mathrm{A}}}$, is defined inductively thus

- $C_{\boxed{\mathrm{A}}}^{\boxed{\mathrm{A}}} = true$

- $C_{if\ b\ then\ T\ else\ U}^{\boxed{\mathrm{A}}} = \begin{cases} b \wedge C_T^{\boxed{\mathrm{A}}} & \text{if } \boxed{\mathrm{A}} \text{ in } T \\ \neg b \wedge C_U^{\boxed{\mathrm{A}}} & \text{if } \boxed{\mathrm{A}} \text{ in } U \end{cases}$

- $C_{\alpha.T}^{\boxed{\mathrm{A}}} = C_T^{\boxed{\mathrm{A}}}$

- $C_{T+U}^{\boxed{\mathrm{A}}} = \begin{cases} C_T^{\boxed{\mathrm{A}}} & \text{if } \boxed{\mathrm{A}} \text{ in } T \\ C_U^{\boxed{\mathrm{A}}} & \text{if } \boxed{\mathrm{A}} \text{ in } U \end{cases}$

- $C_{T|U}^{\boxed{\mathrm{A}}} = \begin{cases} C_T^{\boxed{\mathrm{A}}} & \text{if } \boxed{\mathrm{A}} \text{ in } T \\ C_U^{\boxed{\mathrm{A}}} & \text{if } \boxed{\mathrm{A}} \text{ in } U \end{cases}$

- $C_{T[f]}^{\boxed{\mathrm{A}}} = C_T^{\boxed{\mathrm{A}}}$

- $C_{T\backslash c}^{\boxed{\mathrm{A}}} = C_T^{\boxed{\mathrm{A}}}$

Intuitively, $C_T^{\boxed{\mathrm{A}}}$ represents the boolean condition (implicitly) holds at $\boxed{\mathrm{A}}$ in T. The usefulness of this notion lies in the fact that one can safely insert any boolean expression weaker than it in front of $\boxed{\mathrm{A}}$ without change the behaviour of T, because such an expression will eventually be "absorbed" by $C_T^{\boxed{\mathrm{A}}}$. More generally, a boolean expression can be added at every P-occurrence if it is implied by the context conditions of all such occurrences (module renaming of free data variables). For example, let T denote the term

$$\begin{array}{l} c!x.\mathit{if}\ x \geq 0\ \mathit{then}\ P(x)\ \mathit{else}\ Q\ + \\ d?y.\mathit{if}\ y \geq 1\ \mathit{then}\ P(y)\ \mathit{else}\ Q \end{array}$$

It is straightforward to calculate that $C_T^{\boxed{\mathrm{P(x)}}} = x \geq 0$ and $C_T^{\boxed{\mathrm{P(y)}}} = y \geq 1$. Consider the boolean $z \geq 0$. We have $x \geq 0 \models (z \geq 0)[x/z]$ and $y \geq 1 \models (z \geq 0)[y/z]$. Also

$$\begin{array}{rl} T[\lambda z(z \geq 0)P/P] \equiv & c!x.\mathit{if}\ x \geq 0\ \mathit{then}\ (\mathit{if}\ x \geq 0\ \mathit{then}\ P(x)\ \mathit{else}\ NIL)\ \mathit{else}\ Q\ + \\ & d?y.\mathit{if}\ y \geq 1\ \mathit{then}\ (\mathit{if}\ y \geq 0\ \mathit{then}\ P(y)\ \mathit{else}\ NIL)\ \mathit{else}\ Q \end{array}$$

It is easy to check that $\vdash T[\lambda z(z \geq 0)P/P] = T$.

In general, we have the following lemma:

Lemma 3.1 *If* $C_T^{\boxed{\mathrm{A}}} \models b$ *then* $\vdash T = T[bA/\boxed{A}]$.

Proof: Straightforward induction on the structure of T. □

Corollary 3.2 *Let b be a boolean with $fv(b) \subseteq \{\tilde{x}'\}$, P an identifier or unknown. If* $C_T^{\boxed{\mathrm{A}}} \models b[\tilde{x}/\tilde{x}']$ *for every P-occurrence $A \equiv P(\tilde{x})$ in T, then* $\vdash T = T[\lambda\tilde{x}'bP/P]$.

Applying `ufi` to goal (7) generates the following subgoals:

$$B_i' \rhd X_i'(\tilde{x}_i) = T_i', \qquad 1 \leq i \leq m \tag{8}$$

where $T_i' \equiv T_i[X_k'/R_k | 1 \leq k \leq m]$, $X_1'(\tilde{x}_1)$ and B_1' are bound to S_1 and b_1, respectively. From the description given in the previous subsection, an "unknown" can only be bound or made known when it is the top level term of one side of the current equation. In this case we say this unknown is *exposed*. The current global assumption when an unknown is exposed is called the *exposure condition* of the exposure. Note that although each exposed unknown has a unique occurrence in some T_i', an occurrence of an unknown in a T_i' may correspond to more than one exposures of this unknown due to applications of IF and CUT rules.

During a proof each term T in an equation $B \rhd U = T$ is associated with a global condition, namely the global assumption, say b, of the equation. We call $b \wedge C_T^{\boxed{\mathrm{A}}}$ the *global context condition* of an atom occurrence $\boxed{\mathrm{A}}$ in T. The following lemma says that such global context conditions are invariants of a proof.

Lemma 3.3 *Let* $B \rhd X_j'(\tilde{x}) = T$ *be a goal equation and* $\boxed{A}$ *with* $A \equiv X_k'(\tilde{x}')$ *an unknown occurrence in T. Suppose during the proof* $X_k'(\tilde{x}')$ *is exposed n times with exposure conditions b_i, $1 \leq i \leq n$. Then* $B \wedge C_T^{\boxed{\mathrm{A}}} = \bigvee_{1 \leq i \leq n} b_i$.

Proof: By induction on the number of rule applications during the proof. We need only to check that each rule application preserves the global context condition of an unknown, i.e. $B \wedge C_T^{\boxed{\mathrm{A}}} = \bigvee_{1 \leq i \leq n}(B_i \wedge C_{T_i}^{\boxed{\mathrm{A}}})$, $n \in \{1, 2\}$ where $B \rhd U = T$ is a goal and $B_i \rhd U_i = T_i$, $1 \leq i \leq n$, $n \in \{1, 2\}$ are the subgoal(s) containing $\boxed{\mathrm{A}}$ generated from the goal by a rule application. For this purpose the only interesting rules are IF and CUT.

For the IF rule there are two cases. If T is of the form if b then T' else T'' then the two subgoals are $B \wedge b \rhd U = T'$ and $B \wedge \neg b \rhd U = T''$. If $\boxed{\mathrm{A}}$ is in T' then it is not in T'', and

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge b \wedge C_{T'}^{\boxed{\mathrm{A}}}$$

If $\boxed{\mathrm{A}}$ is in T'' then it is not in T', and

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge \neg b \wedge C_{T''}^{\boxed{\mathrm{A}}}$$

If U is of the form if b then U' else U'' then the two subgoals are $B \wedge b \rhd U' = T$ and $B \wedge \neg b \rhd U'' = T$, and

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge (b \vee \neg b) \wedge C_T^{\boxed{\mathrm{A}}} = (B \wedge b \wedge C_T^{\boxed{\mathrm{A}}}) \vee (B \wedge \neg b \wedge C_T^{\boxed{\mathrm{A}}})$$

For the CUT rule the two subgoals are $B_1 \rhd U = T$ and $B_2 \rhd U = T$ with $B \models B_1 \vee B_2$. Hence

$$B \wedge C_T^{\boxed{\mathrm{A}}} = B \wedge (B_1 \vee B_2) \wedge C_T^{\boxed{\mathrm{A}}} = (B \wedge B_1 \wedge C_T^{\boxed{\mathrm{A}}}) \vee (B \wedge B_2 \wedge C_T^{\boxed{\mathrm{A}}})$$

□

Now suppose during the proofs of the subgoals (8) $X_i'(\tilde{x}_i)$ is bound to S_i, and B_i' to b_i, for $2 \leq i \leq m$. Then we have

Lemma 3.4 *When all subgoals (8) are proved, and the proof obligations generated during the proofs are valid, the following hold:*

1. $\vdash b_i \rhd S_i = T_i'[\lambda \tilde{x}_k S_k / X_k' | 1 \leq k \leq m]$

2. *For each T_i' and each occurrence* $\boxed{A}$ *with $A \equiv X_j'(\tilde{x})$ in T_i',* $b_i \wedge C_{T_i'}^{\boxed{\mathrm{A}}} \models b_j[\tilde{x}/\tilde{x}_j]$

Proof: *1* is a direct consequence of the fact that $X_i'(\tilde{x}_i)$ is bound to S_i for each i. To see *2*, suppose X_j' is exposed n times during the proof with exposure conditions b_k', $1 \leq k \leq n$. By Lemma 3.3

$$b_i \wedge C_{T_i'}^{\boxed{\mathrm{A}}} = \bigvee_{1 \leq k \leq n} b_k'$$

The first time $X_j'(\tilde{x}_j)$ is exposed it is bound to S_j and at the same time B_j' is bound to b_j. For each other exposure $X_j'(\tilde{x})$ is made known and a proof obligation $b_k' \models b_j[\tilde{x}/\tilde{x}_j]$

is generated. Hence $b_i \wedge C^{\boxed{\text{A}}}_{T'_i} \models b_j[\tilde{x}/\tilde{x}_j]$, under the assumption that these proof obligations are valid. $\square$

From this lemma and Corollary 3.2 we can derive the correctness of the implementation:

Theorem 3.5 *The three commands* `ufi`, `bind` *and* `known` *correctly implement the inference rule UFI. That is, when goal (7) is proved with* `ufi`, *and the proof obligations generated during the proofs are valid, the following hold:*

1. $\vdash b_i \rhd S_i = T_i[\lambda\tilde{x}_k(b_k S_k)/R_k|1 \leq k \leq m]$
2. $\vdash b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda\tilde{x}_k b_k R_k/R_k|1 \leq k \leq m]$

Proof: In the following the range of k is omitted.

1.

$$\begin{array}{rcl} \vdash b_i \rhd S_i & \stackrel{3.4.1}{=} & T'_i[\lambda\tilde{x}_k S_k/X'_k|k] \\ & \stackrel{3.4.2,3.2}{=} & T'_i[\lambda\tilde{x}_k b_k X'_k/X'_k|k][\lambda\tilde{x}_k S_k/X'_k|k] \\ & = & T'_i[\lambda\tilde{x}_k(b_k S_k)/X'_k|k] \\ & = & T_i[X'_k/R_k|k][\lambda\tilde{x}_k(b_k S_k)/X'_k|k] \\ & = & T_i[\lambda\tilde{x}_k(b_k S_k)/R_k|k]. \end{array}$$

2.

$$\begin{array}{rcl} \vdash b_i \rhd R_i(\tilde{x}_i) & = & T_i \\ & = & T_i[X'_k/R_k|k][R_k/X'_k|k] \\ & = & T'_i[R_k/X'_k|k] \\ & \stackrel{3.4.2,3.2}{=} & T'_i[\lambda\tilde{x}_k b_k X'_k/X'_k|k][R_k/X'_k|k] \\ & = & T'_i[\lambda\tilde{x}_k b_k R_k/X'_k|k] \\ & = & T_i[X'_k/R_k|k][\lambda\tilde{x}_k b_k R_k/X'_k|k] \\ & = & T_i[\lambda\tilde{x}_k b_k R_k/R_k|k] \end{array}$$

$\square$

4 Conclusions

We have discussed the formulation of unique fixpoint induction in the setting of value-passing process calculi and showed how this inference rule is implemented in the verification tool *VPAM*. By exploiting context conditions the implementation reduces the number of subgoals as required by the UFI rule by half. An argument has also been given to justify the implementation.

Theoretical results concerning the soundness and completeness of the proof system with the UFI rule will be reported in a separate paper.

Acknowledgment Thanks to an anonymous referee for detailed comments.

References

[BBV92] T. Bolognesi, E. Brinksma, and C.A. Vissers. *Proceedings of Third Lotosphere Workshop.* Pisa, CNR-CNUCE, 1992.

[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.

[HL93] M. Hennessy and H. Lin. Proof systems for message-passing process algebras. In *CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 202–216, 1993.

[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.

[Lin93] H. Lin. A verification tool for value-passing processes. In *Proceedings of* 13^{th} *International Symposium on Protocol Specification, Testing and Verification*, IFIP Transactions. North-Holland, 1993.

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

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

Appendices

A Inference Rules

EQUIV $\dfrac{}{true \rhd T = T} \quad \dfrac{b \rhd T = U}{b \rhd U = T} \quad \dfrac{b \rhd T = U, U = V}{b \rhd T = V}$

CONGR $\dfrac{b \rhd T_i = U_i \quad i = 1,2}{b \rhd op(T_1, T_2) = op(U_1, U_2)} \quad op \notin \{., ?, !\}$

EQN $\dfrac{}{true \rhd T\sigma = U\sigma} \quad T = U$ is an axiom

RENAME $\dfrac{}{true \rhd c?x.T = c?y.T[y/x]} \quad y \notin fv(T)$

L-INPUT $\dfrac{b \rhd T = U}{b \rhd c?x.T = c?x.U} \quad x \notin fv(b)$

OUTPUT $\dfrac{b \models e = e',\ b \rhd T = U}{b \rhd c!e.T = c!e'.U}$

TAU $\dfrac{b \rhd T = U}{b \rhd \tau.T = \tau.U}$

IF $\dfrac{b \wedge b' \rhd T = V \quad b \wedge \neg b' \rhd U = V}{b \rhd \text{if } b' \text{ then } T \text{ else } U = V}$

UFI $\dfrac{\begin{array}{c} b_i \rhd S_i = T_i[\lambda\tilde{x}_k(b_k S_k)/R_k | 1 \le k \le m], \\ b_i \rhd R_i(\tilde{x}_i) = T_i[\lambda\tilde{x}_k(b_k R_k(\tilde{x}_i))/R_k | k], \end{array} \quad 1 \le i \le m}{b_1 \rhd S_1 = R_1(\tilde{x}_i)} \quad \begin{array}{l} R_i(\tilde{x}_i) = T_i \\ \tilde{R} \text{ guarded in } \tilde{T} \end{array}$

REC $\dfrac{}{true \rhd P_i(\tilde{x}_i) = T_i[\lambda\tilde{x}_k T_k/P_k | 1 \le k \le m]} \quad P_i(\tilde{x}_i) = T_i,\ 1 \le i \le m$

CUT $\dfrac{b \models b_1 \vee b_2,\ b_1 \rhd T = U \quad b_2 \rhd T = U}{b \rhd T = U}$

ABSURD $\dfrac{}{false \rhd T = U}$

B Value-passing *CCS* Axioms in *VPAM*

```
A1    x + x = x
A2    x + NIL = x
PN    x | NIL = x

R1    (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
R4    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)

IFP   (if b then x else y) | z = if b then (x | z) else (y | z)
IFA   (if b then x else y) + z = if b then (x + z) else (y + z)
IFR   (if b then x else y)\A = if b then x\A else y\A
IFN   (if b then x else y)[a/b] = if b then x[a/b] else y[a/b]
IFT   tau.(if b then x else y) = if b then tau.x else tau.y
IFO   c!e.(if b then x else y) = if b then c!e.x else c!e.y

expansion law

EXP 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
      async(c?x) = true           if  not(c in A)
      async(c!e) = true           if  not(c in A)
```