On Process Equivalence = Equation Solving in CCS

On Process Equivalence = Equation Solving in CCS

Raúl Monroy* (raulm@itesm.mx)
Computer Science Department
Tecnológico de Monterrey—Campus Estado de México
Carretera al lago de Guadalupe Km 3.5, 52926 Atizapán de Zaragoza
México

Alan Bundy† (a.bundy@ed.ac.uk)
Centre for Intelligent Systems and their Applications
School of Informatics University of Edinburgh Appleton Tower
Crichton St, Edinburgh EH8 9LE
Scotland

Ian Green
QSS Ltd, 13 Atholl Crescent, Edinburgh, Scotland, UK

Abstract. Unique Fixpoint Induction (UFI) is the chief inference rule to prove the equivalence of recursive processes in the Calculus of Communicating Systems (CCS) (Milner, 1989). It plays a major role in the equational approach to verification. Equational verification is of special interest as it offers theoretical advantages in the analysis of systems that communicate values, have infinite state space or show parameterised behaviour. We call these kinds of systems *VIPSs*. VIPSs is the acronym of Value-passing, Infinite-State and Parameterised Systems.

Automating the application of UFI in the context of VIPSs has been neglected. This is both because many VIPSs are given in terms of recursive function symbols, making it necessary to carefully apply induction rules other than UFI, and because proving that one VIPS process constitutes a fixpoint of another involves computing a process substitution, mapping states of one process to states of the other, that often is not obvious. Hence, VIPS verification is usually turned into equation solving (Lin, 1995a). Existing tools for this proof task, such as VPAM (Lin, 1993), are highly interactive.

We introduce a method that automates the use of UFI. The method uses middle-out reasoning (Bundy et al., 1990a) and, so, is able to apply the rule even without elaborating the details of the application. The method introduces meta-variables to represent those bits of the processes' state space that, at application time, were not known, hence, changing from equation verification to equation solving.

Adding this method to the equation plan developed by Monroy, Bundy and Green (2000a), we have implemented an automatic verification planner. This planner increases the number of verification problems that can be dealt with fully automatically, thus improving upon the current degree of automation in the field.

* Partially supported by grants CONACyT-33337-A and ITESM CCEM-0302-05.
† Partially supported by EPSRC GR/L/11724.

1. Introduction

Unique Fixpoint Induction (UFI) is an inference rule for reasoning about recursion. It is widely used in the community of process algebras, such as CCS (Milner, 1989) and ACP (Bergstra and Klop, 1985), where it is respectively called the *unique solution of equations* and the *recursion specification principle*. CCS, the *Calculus of Communicating Systems*, is suitable for modelling and analysing processes. It is well-established in both industry and academia, and has strongly influenced the design of LOTOS (ISO, 1989).

In CCS, UFI is used to equationally characterise bisimulation and other notions of process equivalence. It therefore plays a chief role in the so-called *equational approach* to verification. There, both a program and its specification are represented as processes and then logical reasoning is used to prove they are equivalent. The equational approach to verification has captured increasing interest in the last decade. This is because it offers theoretical advantages in the analysis of a large class of practical communicating systems involving value-passing actions, infinite states and parameterised behaviour. We call these kinds of systems *Value-passing, Infinite-state and Parameterised Systems*, or *VIPSs* for short.

Automating the application of UFI in the context of VIPS verification has been neglected, because it is an extremely difficult task. There are two roots to the problem. First, many VIPSs are given in terms of recursive function symbols, making it necessary to carefully apply induction rules other than UFI. Second, proving that one VIPS process constitutes a fixpoint of another usually requires *solving* equations, rather than proving them, as shown by (Lin, 1995a). Existing tools for VIPS verification, such as VPAM (Lin, 1993), are highly interactive. We suggest that techniques based on proof planning (Bundy, 1988), generalisation and middle-out reasoning (Bundy et al., 1990a) can provide significant automation in this context.

Proof planning is a meta-level reasoning technique, developed especially as a search control technique to automate theorem proving. A *proof plan* captures general knowledge about the commonality between the members of a proof family, and is used to guide the search for more proofs in that family. *Generalisation* is the speculation of a new conjecture, which, if successfully proved, is then used to justify the original one, via an application of the cut-rule. Generalisation is used to achieve a more powerful theorem, to simplify the proof task or both. *Middle-out reasoning* is used to postpone choices about key parts of a proof for as long as possible. This is achieved by replacing parts of the

theorem goal with higher-order meta-variables. These meta-variables are gradually refined as further proof steps take place.

Here, middle-out reasoning is used to enable an application of UFI, while leaving certain details of various proof steps to be filled in at a later stage. This is done by introducing meta-level variables that are to be bound when more information about the process term structure becomes available. Hence as Lin in VPAM, we turn the verification task from proving equations to solving them, only that proof plan formation takes place at the meta-level, rather than at the object-level, and is fully automatic.

Middle-out reasoning for unique fixpoint induction theorem proving has been implemented within the *Clam* proof planner (Bundy et al., 1990b). Combining it with two existing proof plans, one for inductive theorem proving (Bundy, 1988) and the other for CCS equational verification (Monroy et al., 1998; Monroy et al., 2000a), we have implemented a proof plan for the equational approach to verification. The verification planner comprehends heuristics to automatically guide proof search. It has been used to verify problems that previously required human interaction, hence improving upon the current degree of automation.

This paper elaborates and extends (Monroy et al., 2000b)—other details can be found in (Monroy, 1998). The rest of this paper is organised as follows: §2 describes CCS: syntax, semantics and process equivalence. It can be safely skipped by readers familiar with process algebras. §3, describes the proof system we use to establish process equivalence. §4 characterises the kinds of proofs we shall automate. §5 introduces the proof planning technique, used to automate program verification. In §6 and §7, we introduce the methods unique, generalisation and equation solving. We also illustrate the strength of our proof plan with a couple of running examples by providing highlights of the proof plans obtained automatically. §8 summarises experimental results and §9 discusses related work. Finally, we draw conclusions in §10.

2. CCS

Terms of the *Calculus of Communicating Systems* (CCS) (Milner, 1989) represent processes. Processes have their own identity, circumscribed by their entire capabilities of interaction. Interactions occur either between two agents, or between an agent and its environment. They are referred to as *actions*. An action is said to be *observable* if it denotes an interaction between an agent and its environment. Otherwise, it is said to

be *unobservable*. This interpretation of observation underlies a precise and amenable theory of behaviour. Whatever is observable is regarded as the behaviour of a system. Two agents are held to be equivalent if their behaviour is indistinguishable to an external observer. This is called bisimilarity.

2.1. Syntax

We assume a set of value variables, *Var*, a set of value constants, *Val*, and a set of value function symbols, F. We use $x, y, \ldots$ to range over *Var* and $v_1, v_2, \ldots$ to range over *Val*. For F, we use the function symbols $\times, \div, \ldots$ following their standard interpretation. A *value expression*, e, is either a value constant, a value variable, or a function symbol of arity n, applied to n value expressions. A *boolean expression*, b, is a quantifier-free formula. So, it is built out of predicate relations, which may take value expressions as parameters, and which are closed under the logical connectives. We assume a set of types for values, ranged over by V.

The set of actions, $\mathcal{A}ct$, contains the set of input actions, the set of output actions and the unobservable action τ, which denotes process interaction. Input actions are of the form $a(x)$, where $a \in \mathcal{A}$, the set of input labels. Respectively, output actions are of the form $\overline{a}(e)$, where $\overline{a} \in \overline{\mathcal{A}}$, the set of output labels. Actions, input or output, may take no parameters at all. In that case, we simply write $a, \overline{a}, \ldots$.

The set of labels, $\mathcal{L}$, is defined to be $\mathcal{A} \cup \overline{\mathcal{A}}$. Thus:[1]

$$\mathcal{A}ct = \{a(x) : a \in \mathcal{A}\} \cup \{\overline{a}(x) : \overline{a} \in \overline{\mathcal{A}}\} \cup \{\tau\}$$

We use $\alpha, \beta, \ldots$ to range over $\mathcal{A}ct$, and use $\ell, \ell', \ldots$ to range over $\mathcal{L}$. Let $a, b, c, \ldots$ range over $\mathcal{A}$ and $\overline{a}, \overline{b}, \overline{c}, \ldots$ over $\overline{\mathcal{A}}$. We use $K, L, \ldots$ to denote subsets of $\mathcal{L}$, and use $\overline{L}$ for the set of complements of labels in L, $\overline{L} = \{\overline{\ell} : \ell \in L\}$. Recall that complementation extends to all labels: a and $\overline{a}$ are said to be *complementary*; i.e. $\overline{\overline{a}} = a$.

$\mathcal{K}$ stands for the set of agent constants, or constants for short. Constants refer to unique behaviour and are assumed to be declared by means of the *definition facility*, $\stackrel{\text{def}}{=}$. We use $A, B, C, \ldots$ to range over $\mathcal{K}$. Constants may take parameters. Each *parameterised constant* A with arity n is assumed to be given by a set of defining equations, each of which is of the form:

$$b_i \rightarrow A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} E_i$$

where $\rightarrow$ denotes logical implication, b_i a boolean expression and where E_i denotes an agent expression. These defining equations should be

[1] $\{x : \phi\}$ means the set of x such that ϕ.

grouped together and be associated with necessary constraints to make up a proper definition.

Let $\{E_i : i \in I\}$ stand for a family of expressions, indexed by I, let $t\{e_1/x_i \ : \ i \in I\}$ for the substitution operation which simultaneously replaces all free occurrences of x_i by e_i, for all $i \in I$, in t, and $fv(\{t_1, \ldots, t_n\})$ stand for the free value variables of the value expressions $t_i\,(i \in \{1, \ldots, n\})$. Then:

DEFINITION 1. *A parameterised constant A is* well-defined *if it is associated with a set of defining equations of the form:*

$$\{b_i \rightarrow A(x_1, \ldots, x_n) \stackrel{def}{=} E_i : i \in I\}$$

such that:

1. $fv(\{b_i : i \in I\}) \subseteq \{x_1, \ldots, x_n\}$ *and* $fv(\{E_i : i \in I\}) \subseteq \{x_1, \ldots, x_n\}$*; and*
2. *for any* $v_1 \in Val, \ldots, v_n \in Val$, $(\bigvee_{i \in I} b_i)\{v_1/x_1, \ldots v_n/x_n\} = true$ *and* $\forall i, j \in I.\ i \neq j \rightarrow (b_i \wedge b_j)\{v_1/x_1, \ldots v_n/x_n\} = false$.

Notice that, in this syntax, to express the conditional process:

$$P \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } Q \textbf{ else } R$$

we use the following two conditional defining equations:

$$b \rightarrow P \stackrel{\text{def}}{=} Q$$
$$\neg b \rightarrow P \stackrel{\text{def}}{=} R$$

The set of *agent expressions*, $\mathcal{E}$, is defined as the smallest set that contains $\mathcal{K}$, the set of constants, $\mathcal{X}$, the set of agent variables, ranged over by $X, Y, \ldots$, and the agent expressions below:

- $a(x).E$, $\overline{a}(e).E$, $\tau.E$, called *Prefixes* ($a \in \mathcal{A}$, $\overline{a} \in \overline{\mathcal{A}}$);
- $\sum_{i \in I} E_i$, called a *Summation* (I an indexing set);
- $E_1 \mid E_2$, called a *Composition*;
- $E \setminus L$, called a *Restriction* ($L \subseteq \mathcal{L}$);
- $E[f]$, called a *Relabelling* (f a relabelling function[2]); and

[2] Relabelling functions map actions into actions; $\ell'_1/\ell_1, \ldots, \ell'_n/\ell_n$ stands for the relabelling function which sends ℓ_i to ℓ'_i and $\overline{\ell_i}$ to $\overline{\ell'_i}$, for $1 \leq i \leq n$, and ℓ to ℓ, otherwise. By convention, $f(\tau) = \tau$. Relabelling functions respect complements; that is, $f(\ell) = \ell'$ implies $f(\overline{\ell}) = \overline{\ell'}$. Also, relabelling functions cannot turn an input action into an output one and vice versa.

– $A(e_1, \ldots, e_n)$, called a *Constant* (constant A is of arity n).

where E, E_i are already in $\mathcal{E}$. A *process* is an agent expression containing no agent variables.

We use a *lambda abstraction* to close up a substituting process term and a *lambda application* to implement value-passing. Thus, the expression $E\{(\lambda\tilde{x}.A)/B\}$, where $\tilde{x}$ stands for the free value variables of A, is the result of substituting $\lambda\tilde{x}.A$ for identifier B in E followed by β-conversion (Lin, 1995a). For example:

$$c(z).P(z)\{(\lambda y.Q(y))/P\} = c(z).(\lambda y.Q(y))(z) = c(z).Q(z)$$

Except for the use of a different representation of the conditional construct, this language is value-passing CCS with parameterised constants. In our implementation, Summation, $\sum$, takes one of the following forms:

1. the *deadlock agent*, $\mathbf{0} \stackrel{\text{def}}{=} \sum_{i\in\emptyset} E_i$, capable of no actions whatever; and
2. *Binary Summation*, $E_1 + E_2 \stackrel{\text{def}}{=} \sum_{i\in\{1,2\}} E_i$, which takes two summands only.

An indexed Summation over a set,[3] is *not* a CCS term, but a function symbol mapping a set into a CCS term, given by:

$$\sum_{i\in\text{nil}} E_i \stackrel{\text{def}}{=} \mathbf{0}$$
$$\sum_{i\in h::t} E_i \stackrel{\text{def}}{=} E(h) + \sum_{i\in t} E_i$$

2.2. Semantics

Processes are given meaning by means of the following labelled transition system:

$$\langle \mathcal{E}, \mathcal{A}ct, \{\stackrel{\alpha}{\rightarrow}: \alpha \in \mathcal{A}ct\}\rangle$$

The transition relation, $\stackrel{\alpha}{\rightarrow} \subseteq \mathcal{E} \times \mathcal{E}$, for each $\alpha \in \mathcal{A}ct$, is the smallest relation satisfying the inference transition rules below (the dual rules for $+$ and $\mid$ have been omitted:)

$$\mathbf{Act}_o \ \frac{}{\alpha.E \stackrel{\alpha}{\rightarrow} E}(\alpha \in \{\tau, \overline{a}(\|e\|)\}) \qquad \mathbf{Act}_i \ \frac{}{a(x).E \stackrel{a(v)}{\rightarrow} E\{v/x\}} \ (v \in V)$$

[3] Sets and set-theoretic operations are implemented using lists and simulated by operations on lists, respectively. $h :: t$ denotes the list of head h and tail t; nil denotes the empty list. :: is the infix constructor function on the list type.

$$\mathbf{Sum}_1 \ \frac{E_1 \xrightarrow{\alpha} E'}{E_1 + E_2 \xrightarrow{\alpha} E'}$$

$$\mathbf{Com}_1 \ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \mathbf{Com}_2 \ \frac{E \xrightarrow{a(v)} E' \quad F \xrightarrow{\overline{a}(v)} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\mathbf{Res} \ \frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L} (\alpha \in \{\tau, \ell(\|e\|])\}; \ell, \overline{\ell} \notin L)$$

$$\mathbf{Rel} \ \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]} \qquad \mathbf{Con} \ \frac{E\rho \xrightarrow{\alpha} E'\rho}{A\rho \xrightarrow{\alpha} E'\rho} (b \to A \stackrel{\text{def}}{=} E, \ b\rho = \text{true})$$

where $\|e\|$ denotes the result of evaluating the (variable-free) expression e, and where ρ is a data evaluation assigning each data variable a data value.

Whenever $E \xrightarrow{\alpha} F$, F is said to be an *α-derivative* (or simply a *derivative*) of E.

The rules **Act**, **Sum**, **Com**, **Res**, **Rel** and **Con** are associated respectively with Prefix, Summation, Composition, Restriction, Relabelling and with constants. Together, these transition inference rules provide the *early interpretation* for modelling communication (Milner et al., 1993). The interpretation of each rule is straightforward; here, we shall look only into $\mathbf{Act}_i$ and $\mathbf{Com}_2$. $\mathbf{Act}_i$ indicates that transition, by the execution of actions, yields an instantiation of bound value variables, very much like the β-conversion rule $(\lambda x.M)N \to M\{N/x\}$.

$\mathbf{Com}_2$ stands for the possibility of two agents interacting with each other by the execution of complementary actions; that is, communication. Communication actions are all indistinguishable from each other; hence, they are all denoted by τ. The execution of τ is assumed to take no time (perfection), and cannot be noticed by an external observer (invisibility). $\mathbf{Com}_2$ handles value-passing accordingly, as suggested by $\mathbf{Act}_i$.

2.3. Process Equivalence and Bisimulation

CCS provides a precise and tractable theory of behaviour based on observations, used as a basis for deciding process equivalence. Different interpretations of what is observable give rise to different equivalence relations. *Observation congruence* is an equivalence relation that supplies two essential ingredients for process analysis: (i) the abstraction of internal actions, and (ii) the property of being a congruence. Without such ingredients the analysis of complex communicating systems would rapidly become unmanageable, due to the copious number of internal communications and system subcomponents.

Observation congruence, like other behavioural equivalences, resorts to the notion of *bisimulation* (Park, 1981), which Rathke (1997) defines as follows:[4]

DEFINITION 2 (Weak Bisimulation). *A binary symmetric relation, $\mathcal{S}$, over closed processes expressions is a* weak bisimulation*, or a* bisimulation *for short, if $(P,Q) \in \mathcal{S}$ implies, for all $v \in$ Val,*

i) whenever $P \stackrel{a(v)}{\rightarrow} (\lambda x.P')(v)$ then, for some Q', $Q \stackrel{a(v)}{\Rightarrow} (\lambda y.Q')(v)$ and $(P'\{v/x\}, Q'\{v/y\}) \in \mathcal{S}$; and

ii) whenever $P \stackrel{\alpha}{\rightarrow} P'$, with $\alpha \neq a(v)$, then for some Q', $Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $(P', Q') \in \mathcal{S}$.

where the *weak* transition relation, $\Rightarrow$, is defined as follows: $\stackrel{\epsilon}{\Rightarrow}$ is the reflexive, transitive closure of $\stackrel{\tau}{\rightarrow}$. $\stackrel{\alpha}{\Rightarrow}$ is $\stackrel{\epsilon}{\Rightarrow}\stackrel{\alpha}{\rightarrow}\stackrel{\epsilon}{\Rightarrow}$, whenever α is τ or $\overline{a}(e)$, and $\stackrel{\epsilon}{\Rightarrow}\stackrel{a(v)}{\rightarrow}$, otherwise. The relation $\stackrel{\hat{\alpha}}{\Rightarrow}$ is $\stackrel{\epsilon}{\Rightarrow}$ if α is τ and is $\stackrel{\alpha}{\Rightarrow}$ otherwise. Whenever $P \stackrel{\alpha}{\Rightarrow} P'$ we say that P' is an α-*descendant* of P.

The union of all bisimulations yields an important equivalence relation, called bisimilarity and closely related to observation congruence. P and Q are *bisimilar*, written $P \approx Q$, if $(P,Q) \in \mathcal{S}$ for some bisimulation $\mathcal{S}$.

Bisimilarity is not a congruence relation: $\sum$ is the only operator which breaks equivalence. The largest congruence relation included in $\approx$ is observation congruence:

DEFINITION 3 (Observation congruence). *Two processes, P and Q, are* observation-congruent*, written $P \approx^c Q$, if any initial τ move of one is matched by at least one τ move of the other and the derivatives are bisimilar thereafter.*

Most properties that relate $\approx$ and $\approx^c$ are captured in the following theorem:

THEOREM 4 (Hennessy's theorem).

$$P \approx Q \text{ if and only if } P \approx^c Q, \text{ or } P \approx^c \tau.Q, \text{ or } \tau.P \approx^c Q$$

Observation congruence is the relation selected to achieve the verification task. The proof system whereby verification proofs are produced is shown in § 3.

[4] Like Milner, we adopt the convention of using *bisimulation* as an abbreviation for *weak bisimulation*. Similarly, we use *bisimilarity* to denote the equivalence relation induced by bisimulation, also called *observation equivalence* or *bisimulation equivalence*.

Table I. The Set of Basic Axioms

A1	$P + Q = Q + P$	**P1**	$P \mid Q = Q \mid P$
A2	$(P + Q) + R = P + (Q + R)$	**P2**	$(P \mid Q) \mid R = P \mid (Q \mid R)$
A3	$P + P = P$	**P3**	$P \mid \mathbf{0} = P$
A4	$P + \mathbf{0} = P$		
T1	$\alpha.\tau.P = \alpha.P$		
T2	$P + \tau.P = \tau.P$		
T3	$\alpha.(P + \tau.Q) + \alpha.Q = \alpha.(P + \tau.Q)$		

3. The Proof System

The proof system we use is the standard one for value-passing CCS plus UFI and structural induction. A judgement is an expression of the form:

$$\Gamma \vdash G$$

where G is a formula and Γ is a set of formulae. Its intended meaning is that G is provable under the assumptions Γ. Often, G takes the form $b \rightarrow T = U$, where b is a boolean expression, and T and U are terms of the same type. If the type relates two CCS terms, then the intended meaning of G is that T and U are observation congruent, $T \approx^c U$, for every interpretation Γ that satisfies b, $\Gamma \vdash b$. In this case, we further require that both T and U are determinate.[5]

3.1. Axioms

The set of basic axioms is shown in Table I. **A1**–**A4** are required to characterise strong equivalence, and **T1**–**T3**, together with **A1**–**A4**, to characterise observation congruence. The axioms for Composition, **P1**–**P3**, deal with basic parallel processes. Notice that we do not include axioms for restriction or relabelling, since these operators are captured by expansion.

Expansion is an axiom scheme which reduces parallelism into non-deterministic choice. Let P be an abbreviation of $(P_1 \mid \ldots \mid P_n) \setminus L$,

[5] Let $\sigma = \alpha_1 \cdots \alpha_n$ be a sequence, possibly empty, of observable actions, and let $P \stackrel{\sigma}{\Longrightarrow} P'$ denote $P \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} P'$. Then, P is *determinate* if $P \stackrel{\sigma}{\Longrightarrow} P'$ and $P \stackrel{\sigma}{\Longrightarrow} P''$ implies that $P' \approx P''$..

with $n \geq 1$ and where each P_i, $1 \leq i \leq n$, might take the form $Q_i[f_i]$ for some relabelling function f_i, then:

$$
\begin{aligned}
P = & \\
& \sum \left\{ \alpha.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_n) \setminus L \,:\, P_i \xrightarrow{\alpha} P_i', \alpha \in \{\tau, \ell(e)\}, \ell, \overline{\ell} \notin L \right\} \\
& + \sum \left\{ \tau.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_j' \mid \ldots \mid P_n) \setminus L \,:\, P_i \xrightarrow{\ell(x)} P_i',\, P_j \xrightarrow{\overline{\ell(e)}} P_j' \right\} \\
& + \sum \left\{ \tau.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_j' \mid \ldots \mid P_n) \setminus L \,:\, P_i \xrightarrow{\overline{\ell(e)}} P_i',\, P_j \xrightarrow{\ell(x)} P_j' \right\}
\end{aligned}
$$

Expansion computes all the actions and derivatives of an agent. The actions can be either asynchronous, when one of the process subcomponents acts independently, or synchronous, when any pair of process subcomponents interact. In expansion, the first summand collects the terms that correspond to the transitions of each individual subcomponent, as long as the associated action does not use a $\ell \in L \cup \overline{L}$. Reciprocally, the second summand and third summand collect the τ-prefixed terms that arise as a result of subcomponent interaction, which by definition takes place upon complementary actions.

3.2. Inference Rules

The inference rules are divided into three classes: rules for manipulating connectives and quantifiers, general purpose rules, and rules for manipulating process terms. The rules for manipulating connectives and quantifiers are basically Gentzen's sequent calculus (Gallier, 1986). General purpose rules include structural induction (on lists and on natural numbers), cut, rewrite, α- and β-conversion. The rules for manipulating process terms are similar to those introduced in (Lin, 1995a) and (Rathke, 1997). They differ only in the treatment of action Prefix: we make use of Hennessy's theorem, while Lin and Rathke do not. Since both the rules for logical formulae and the rules for general purpose are rather standard, attention is given to the rules for manipulating process terms only. Unique Fixpoint Induction forms the basis of both the proof system and the verification plan, and hence it is discussed separately.

3.2.1. *Unique Fixpoint Induction*

Unique fixpoint induction is a rule for reasoning about recursive processes. It states that two processes are equivalent, if they satisfy the same set of equations, so long as such a set of equations has one, and only one, solution. *Uniqueness of solution of equations* is guaranteed by two syntactic properties: guardedness and sequentiality. X is *guarded*

Table II. Rules for Manipulating Process Terms

IN	$\dfrac{\Gamma \vdash P = S \vee P = \tau.S \vee \tau.P = S}{\Gamma \vdash \alpha.P = \alpha.S}\ \alpha \in \{\tau, a(x)\}, x \notin fv(\Gamma)$
OUT	$\dfrac{\Gamma \vdash e = e' \qquad \Gamma \vdash P = S \vee P = \tau.S \vee \tau.P = S}{\Gamma \vdash \overline{a}(e).P = \overline{a}(e').S}$
CONGR	$\dfrac{\Gamma \vdash P_i = S_i \quad i = 1,2}{\Gamma \vdash P_1 \circ P_2 = S_1 \circ S_2}\ \circ \in \{+, \mid\}$
CONS_1	$\dfrac{}{\Gamma \vdash A\phi = E\phi}\ b \to A \stackrel{\text{def}}{=} E, \Gamma \vdash b\phi$
CONS_2	$\dfrac{}{\Gamma \cup \{b_i\} \vdash A_i(\tilde{e}) = E_i\{\tilde{e}/\tilde{x}\}}\ \{b_i \to A_i(\tilde{x}) \stackrel{\text{def}}{=} E_i : i \in I\}$

in E if each occurrence of X is within some subexpression $\ell(x).F$ of E, for $\ell \in \mathcal{L}$. X is *sequential* in E if it occurs in E only within the scope of Prefix or Summation. We write $X \rhd E$ to mean that X is both guarded and sequential in E. Following a standard convention (Milner, 1989), we abbreviate the indexed family $\{E_i : i \in I\}$ of expressions by $\widetilde{E}$, when I is understood.

Let the expressions $E_i\,(i \in I)$ contain at most the variables $X_i\,(i \in I)$ free. Then, the UFI inference rule is as follows:

$$\frac{\Gamma \vdash \widetilde{b} \to \widetilde{P} = \widetilde{E}\{\widetilde{P}/\widetilde{X}\} \qquad \Gamma \vdash \widetilde{b} \to \widetilde{S} = \widetilde{E}\{\widetilde{S}/\widetilde{X}\}}{\Gamma \vdash \widetilde{b} \to \widetilde{P} = \widetilde{S}}\ \widetilde{X} \rhd \widetilde{E} \qquad (1)$$

There is one aspect of the UFI rule that is worth mentioning: it reasons about families of process expressions. We cannot prove that an individual expression satisfies a property without proving that such a property is satisfied by all. In symbols:

$$\text{UFI}_i \qquad \frac{\Gamma \vdash \widetilde{b} \to \widetilde{P} = \widetilde{S}}{\Gamma \vdash b_j \to P_j = S_j}\ j \in I$$

3.2.2. *Other rules for Manipulating Process Terms*
The rest of the rules are displayed in Table II. We omit from display the usual rules for *reflexivity*, *symmetry*, *transitivity*, and some forms of *substitution* of $=$. Notice that we do not require a specific rule for the conditional construct as it is given thorough treatment via the Gentzen rule for $\vee$-elimination.

IN deals with input prefixing. This rule may be a little surprising. There are two aspects to it. First, the prefix: If the prefix is input, care is to be taken to avoid variable capturing. With the side condition,

$x \notin fv(\Gamma)$, the IN rule is sound,[6] for it cannot be used to prove, for example:

$$x = 1 \vdash in(x).\overline{out}(1).\mathbf{0} = in(x).\overline{out}(x).\mathbf{0}$$

This captures that on action $in(x).\overline{out}(x).\mathbf{0}$ becomes $\overline{out}(v).\mathbf{0}$, for some $v \in V$ (c.f. $\mathbf{Act}_i$). Second, the derivative: the alternative rule

$$\frac{P = S}{\alpha.P = \alpha.S}$$

is not sufficiently powerful to prove many true identities, mainly for two reasons:

1. we reason backwards, from the input goal to the axioms of the logic; and
2. bisimilarity is strengthened to observation congruence by Prefix, that is $P \approx S \rightarrow \alpha.P = \alpha.S$, but observation congruence is *not* implied by bisimilarity, that is $P \approx S \rightarrow P = S$ is not faithful to the experimental idea of observation.

Thus, to prove $\alpha.P = \alpha.S$, it suffices to show $P \approx S$. But, by Theorem 4, to prove $P \approx S$, we must prove either $P = S$, $P = \tau.S$, or $\tau.P = S$. This result is used in both IN and OUT.

OUT is as IN except that it makes reference to the semantics of value expressions in order to establish identities of the form $\overline{a}(e).P = \overline{a}(e').S$.

The other rules have similar straightforward interpretations and so shall not be considered further. This completes our presentation of the logic used throughout this paper.

4. Program Verification

4.1. The Verification Problem

We use CCS both as a programming language and as a specification language. Programs and their intended specifications need to be expressible as processes themselves (or as function symbols mapping a structured type into a process term). Specifications are constants containing Prefix and Summation only. They provide an abstract description of some (presumably) useful behaviour without hinting at the internal workings of the program. Specifications are all assumed to be declarations.

[6] IN, however, is not complete, because there are different interpretations as to the relation between data and communicating activity. See (Milner et al., 1993) where a thorough discussion of such interpretations is provided.

DEFINITION 5 (declaration). *A set of conditional, defining equations* $\widetilde{b} \rightarrow \widetilde{S} \stackrel{def}{=} \widetilde{E}\{\widetilde{S}/\widetilde{X}\}$ *is a* declaration *if it satisfies the following conditions:*

1. *the expressions* $E_i\,(i \in I)$ *contain at most the variables* $X_i\,(i \in I)$ *free and these variables are all guarded and sequential in each* E_i*; and*

2. *the parameterised constant* $\widetilde{S}$ *is well-defined (definition 1, page 5.)*

For example, to specify the behaviour of a buffer of size n, we write:

$$\begin{array}{lll} n \neq k \wedge k = 0 & \rightarrow & \mathit{Buf}(n,k) \stackrel{\text{def}}{=} \mathit{in}.\mathit{Buf}(n,s(k)) \\ n > k \wedge k \neq 0 & \rightarrow & \mathit{Buf}(n,k) \stackrel{\text{def}}{=} \mathit{in}.\mathit{Buf}(n,s(k)) + \overline{\mathit{out}}.\mathit{Buf}(n,p(k)) \\ n = k \wedge k \neq 0 & \rightarrow & \mathit{Buf}(n,k) \stackrel{\text{def}}{=} \overline{\mathit{out}}.\mathit{Buf}(n,p(k)) \end{array}$$

where s and p, respectively, stand for the successor and the predecessor functions on natural numbers. *Buf* is an indexed family of defining equations, k is the index variable and $\{0,\ldots,n\}$ the indexing set.

As another simple example, to specify an infinite counter which takes as its state any natural number, we write:

$$\begin{array}{lll} n = 0 & \rightarrow & \mathit{Counter}(n) \stackrel{\text{def}}{=} \mathit{inc}.\mathit{Counter}(s(n)) + \mathit{zero}.\mathit{Counter}(n) \\ n \neq 0 & \rightarrow & \mathit{Counter}(n) \stackrel{\text{def}}{=} \mathit{inc}.\mathit{Counter}(s(n)) + \mathit{dec}.\mathit{Counter}(p(n)) \end{array}$$

where n is the index variable and the set of all natural numbers the indexing set.

The systems under verification are given by arbitrary CCS terms possibly containing recursive functions. We call these kinds of expressions *concurrent forms*. Using concurrent forms, it is possible to capture the structure of an infinite-state system, or the structure of a family of finite-state systems. For example, we can implement our example buffer of size n linking n buffers of size 1 (Milner, 1989):[7]

$$\begin{array}{lll} n = 0 & \rightarrow & B^{s(n)} = B \\ n \neq 0 & \rightarrow & B^{s(n)} = B^{\frown} B^{(n)} \end{array} \quad \text{where} \quad \begin{array}{lll} B & \stackrel{\text{def}}{=} & \mathit{in}.B' \\ B' & \stackrel{\text{def}}{=} & \overline{\mathit{out}}.B \end{array}$$

and where

$$P^{\frown} Q \stackrel{\text{def}}{=} (P[c/\mathit{out}] \mid Q[c/\mathit{in}]) \setminus \{c\}$$

[7] Henceforth, the arguments of recursive functions mapping terms of some type into terms of CCS shall be superscript.

Similarly, we can implement the infinite counter above mentioned using a chain of processes. The counter in state n is given by (Milner, 1989):

$$\begin{array}{rcl} C^{(0)} & = & C \\ C^{s(n)} & = & C \frown C^{(n)} \end{array} \quad \text{where} \quad \begin{array}{rcl} C & \stackrel{\text{def}}{=} & inc.(C \frown C) + dec.D \\ D & \stackrel{\text{def}}{=} & \overline{d}.C + \overline{z}.E \\ E & \stackrel{\text{def}}{=} & inc.(C \frown E) + zero.E \end{array}$$

and where the linking combinator, $\frown$, is *now* given by:

$$P \frown Q \stackrel{\text{def}}{=} (P[i'/i, z'/z, d'/d] \;\big|\; Q[i'/inc, z'/zero, d'/dec]) \setminus \{i', d', z'\}$$

Let P be a concurrent form and let S be a declaration. Then we call $P = S$ an instance of the *verification problem*. The following conjectures are example verification problems:

$$\vdash \forall n{:}\text{nat}.\ (n \neq 0 \wedge k = 0) \ \rightarrow B^{(n)} = \mathit{Buf}(n, k) \tag{2}$$

$$\vdash \forall n{:}\text{nat}.\ C^{(n)} = \mathit{Counter}(n) \tag{3}$$

We shall come back to these verification problems later on in the text.

Since in the verification problem we are interested in the specification is a declaration, we will use the following, stronger version of the UFI rule:

$$\frac{\Gamma \vdash b_k \rightarrow P_k = E_k\{(\lambda\tilde{x}.\ P_j)/S_j : j \in I\}\ (k \in I)}{\Gamma \vdash b_i \rightarrow P_i = S_i} \quad \begin{array}{l} b_k \rightarrow S_k \stackrel{\text{def}}{=} E_k \\ S_k \rhd E_k\ (k \in I) \end{array}$$

We shall call $b_k \rightarrow P_k = E_k\{(\lambda\tilde{x}.\ P_j)/S_j : j \in I\}\ (k \in I)$ the *output equation set* and $\{(\lambda\tilde{x}.P_j)/S_j : j \in I\}$ the *process substitution*.

5. Proof Planning

To automate program verification in this context, we use proof planning. In particular, our verification planner is built on top of two existing proof plans, one for inductive proof, called `induction` (Bundy et al., 1991), and the other for a special kind of CCS process equivalence, called `equation` (Monroy et al., 2000a).

5.1. Proof Planning with Methods

Proof planning works in the context of a tactical style of reasoning and uses AI planning techniques to build large compound tactics from simpler ones. A *method* is a high-level description of a tactic. It is a 6-tuple, consisting of an input formula, preconditions, output formulae,

and effects. A method is *applicable* if the current goal matches the method input formula and the method preconditions hold. Preconditions specify properties of the input formula. Proof planning uses the preconditions to predict whether the tactic associated with the method is applicable, without actually running it. Conversely, effects specify properties of the output formulae. Both preconditions and effects are written in a meta-logic. In our case, we use a subset of Prolog as part of the meta-language.

The ultimate result of method application is the output formulae, a list containing the new subgoals generated, if any. When such a list is empty, the method is said to be *terminating*. Upon success, proof planning returns a *proof plan* (i.e., a tactic), whose execution, in the normal case of success, guarantees correctness of the final proof.

5.2. Proof Planning with Critics

The incorporation of an exception handler to proof planning is called *proof planning with critics* (Ireland, 1992). A proof critic is also a tuple consisting of an input formula, preconditions, and effects. A critic is associated with a proof method and is invoked whenever the applicability preconditions of that method partially hold. The application of a critic might yield a proof step. However, it usually is used to enable progress in plan formation, rendering side effects, such as modifications to the input formulae or refinements to the working theory.

5.3. Proof Plans for Induction

Inductive proof planning is driven by the `induction` method, which aims to select the most promising induction scheme for a given goal via a process called *ripple analysis* (Bundy et al., 1989). The base case(s) of proofs by induction are dealt with by the `elementary` and `sym_eval` methods. `Elementary` is a tautology checker for propositional logic and has limited knowledge of intuitionistic propositional sequents, type structures and properties of equality. `Sym_eval` simplifies the goal at hand by means of exhaustive symbolic evaluation, including the unfolding of definitions, and other routine reasoning.

Similarly, the step case(s) of proofs by induction are dealt with by the `wave`, `casesplit` and `fertilise` methods. `Wave` applies *rippling* (Bundy et al., 1993), a heuristic that guides transformations in the induction conclusion to enable the use of an induction hypothesis. Rippling applies a special kind of rewrite-rule, called a *wave-rule*, usually provided by a recursive definition. Wave-rules manipulate the differences between the conclusion and a hypothesis, while preserving their common structure intact. The use of an induction hypothesis,

called *fertilisation*, is handled by the `fertilise` method. `Casesplit` divides a proof into cases, considering a partition defined by a set of conditional rewrite-rules.

Inductive proof planning is conducted using a depth-first search strategy. `Elementary`, `sym_eval`, `wave`, `fertilise` and `induction` are stored in this order. *Clam* looks at the method data-base and tries one method at a time. If no method is applicable, *Clam* will terminate, reporting failure. Otherwise, it will consider the subgoals returned by the first applicable method. This process is applied recursively to each subgoal until no more methods are applicable, or all the goals have been closed with terminating methods.

More information about rippling and proof planning can be found at http://dream.inf.ed.ac.uk/.

5.4. Proof Plans for CCS Identities

The `equation` proof plan deals with a special identity problem where one term, called the *source*, is a concurrent form, and the other, called the *target*, is a sum of prefixed processes. It is composed of four methods: `expansion`, `absorption`, `goalsplit` and `action`.

`Equation` aims to transform the source into the target, following a two-step approach. First, using `expansion`, it transforms the source into a sum of prefixed processes. Then, using `absorption`, it gets rid of any source process summand that is redundant with respect to the behaviour given by the specification. `Expansion` applies the expansion law and `absorption` the axioms **A1**—**4** and **T1**—**3**, shown in Table I. Second, using `goalsplit`, `equation` relates each process summand on one side of the equation with one on the other side. This returns a collection of subgoals, each of which is of the form $\alpha.P = \alpha.Q$, that are dealt with by `action`. `Goalsplit` applies **A1**—**2** and CONGR, and `action` the rules IN and OUT. These two steps resemble the definition of observation congruence, $\approx$ (definition 3, page 8.)

`Equation` has been added to the inductive proof plan, between `elementary` and `sym_eval`. `Sym_eval` (respectively `wave`) includes rewrite-rules (respectively wave-rules) derived from **T2**—**4**, **A1**—**3** and **P1**—**3**. The combined proof plan does not deal with our verification problem, since it does not have a means of controlling the use of the UFI rule: the subject of §6.

To illustrate proof plan methods, we provide the definition of `action` in Figure 1.

Input:
$\Gamma \vdash \alpha.P = \alpha'.Q$
Preconditions:
$\alpha = \alpha'$,
transitions(P, Ps),transitions(Q, Qs),
$Ps \setminus \{\tau\} \subseteq Qs$
Output:
$\Gamma \vdash G'$, where $G' = \begin{cases} P = Q & \text{if } \Delta_1 = \Delta_2 \\ P = \tau.Q & \text{otherwise} \end{cases}$
where
$\Delta_1 = \{\alpha : \text{for some } P', P \stackrel{\alpha}{\Rightarrow} P'\}$ and where
$\Delta_2 = \{\alpha : \text{for some } Q', Q \stackrel{\alpha}{\Rightarrow} Q'\}$
Effects:
$G' \not\equiv \mu.R = \mu.R'$
Meanings of the meta-logic terms:

- transitions(P, Ps) means Ps is a set containing the (single-step) transitions of agent P, in symbols $Ps = \{\alpha : P \stackrel{\alpha}{\rightarrow} P'\}$.

Figure 1. The `action` method

6. The `Unique` Proof Planning Method

This section introduces the `unique` proof planning method and is concerned with *when* and *how* to apply the UFI rule of inference, possibly generalising the input verification conjecture.

6.1. Controlling the Use of UFI

Automating algebraic style proofs for VIPS verification involves two main problems. The first problem lies in dealing with the interactions between two kinds of induction: UFI, concerned with process terms, and structural induction (called induction for short), concerned with data domains. The second problem lies in the fact that there are cases for which UFI should be applied but it is not obvious how to do so; in particular, it is not obvious what process substitution to use.

6.1.1. *The Unique Method*

Let us focus our attention on the first problem. In VIPS verification, UFI and induction are often simultaneously applicable. Whether one rule should be applied before or after the other depends on the verification problem at hand. Heuristic conditions are thus required to enable the application of one rule, while preventing the other. Our heuristic

to coordinate UFI and induction is as follows: *apply induction only if the goal at hand does not suggest a straightforward application of the UFI rule*. We realise this heuristic in two steps. First, through the applicability preconditions of the `unique` method (defined in Figure 2), we capture when an application of the UFI rule is straightforward: the goal at hand is so that the specification conforms to the notion of declaration and both the program and its specification are process symbols indexed over the same scheme.

Input:
$\Gamma \vdash b_i \rightarrow P_i = S_i$
Preconditions:
declaration(S_i), % S_i is a declaration with
indexed_by$(S_i, i \in I)$, % index variable $i \in I$
indexed_by$(P_i, i \in I)$ % P_i is also indexed by i
Output:
$\Gamma \cup \mathcal{C}_i \vdash b_i \ \rightarrow \ P_i = E_i\{(\lambda\tilde{x}.P_i)/S_i\}\,(i \in I)$
where $\mathcal{C}_i = \{b_j \rightarrow P_j = E_j\{(\lambda\tilde{x}.P_j)/S_j\} : j \in I \setminus \{i\}\}$
Effects:
sumPreProc$(E_i\{(\lambda\tilde{x}.P_i)/S_i\})\,(i \in I)$

Meanings of the meta-logic terms:

- declaration(P) means P is a declaration.
- indexed_by$(P, i \in I)$ means P is a family of expressions indexed by I.
- sumPreProc(P) holds only if P is a sum of prefixed processes.

Figure 2. The `unique` method

Second, we place `unique` prior to `induction`, which remains last in the method database. This way, `induction` is tried only if UFI is not straightforwardly applicable. `Unique` is placed after `equation`, since it should be attempted only when the goal relates two recursive CCS terms and thus equivalence cannot be proven without the use of UFI.

6.1.2. *The Generalisation Strategy*
Let us now turn our attention into the second problem, where UFI is applicable to the goal but it is not clear what process substitution to use. To get around this problem, we adopt the following heuristic: *generalise the current goal by replacing the program under verification with a new, non-recursive function symbol, which maps index terms to process terms.* The generalised conjecture is so that i) it can be

used to establish the original goal, and ii) it enables a straightforward application of the UFI rule.

Our mechanism, called `generalisation`, is defined in Figure 3. The new function symbol (c.f. the effects part) is defined so that: i) it contains as many arguments and uses the same indexing scheme as the specification, $(\lambda\tilde{x}.\mathcal{F}_{\mathcal{P}i})/S_i$; it comprehends the current goal, (†); and iii) it contains as many equational cases as the specification, (‡). These conditions all guarantee that, after `generalisation`, `unique` will be applicable and thus the application of the UFI rule will be straightforward.

Input:
$\Gamma \vdash b_i \rightarrow P_i = S_i$
Preconditions:
declaration(S_i), % S_i is a declaration with
indexed_by$(S_i, i \in I)$, % index variable $i \in I$
$\neg$indexed_by$(S_i, i \in I)$ % P_i though is not indexed by i
Effects:
new$(\mathcal{F}_{\mathcal{P}})$ % $\mathcal{F}_{\mathcal{P}}$ is a fresh symbol
Let $b_i \rightarrow S_i\{(\lambda\tilde{x}.\mathcal{F}_{\mathcal{P}i})/S_i\} \stackrel{\text{def}}{=} P_i$, (†)
For every $j \neq i$, if $(b_j \rightarrow S_j \stackrel{\text{def}}{=} \widetilde{E}\{\widetilde{S}/\widetilde{X}\}) \in \widetilde{S}$
Let $b_j \rightarrow S_j\{(\lambda\tilde{x}.\mathcal{F}_{\mathcal{P}j})/S_j\} \stackrel{\text{def}}{=} \mathcal{M}_j(P^{\rightarrow})$ (‡)
Plan a proof of the following generalised conjecture:
$\Gamma \vdash b_i \rightarrow \mathcal{F}_{\mathcal{P}i} = S_i$
Meanings of the meta-logic terms:

- new(t) means t is a newly generated symbol.
- $P^{\rightarrow}$ means some P-derivative.

Figure 3. The `generalisation` critic

The newly introduced function symbol, $\mathcal{F}_{\mathcal{P}}$, is at first under-specified; this is because we do not know how to set some defining equations and so some definition bodies are left as meta-variables, denoted with $\mathcal{M}_i$. These meta-variables, it is hoped, will be instantiated by later planning steps. So, $\mathcal{F}_{\mathcal{P}}$ resembles the *process unknowns* used in VPAM (Lin, 1995b). The use of meta-variables to represent unknown object-level values is called *middle-out reasoning* (Bundy et al., 1990a). Middle-out reasoning changes the verification task, from proving equations in the object-level to solving them in the meta-level.

On any application of UFI, `unique` adds to each goal in the output equation set its context. Let $\mathcal{O} = \{b_h \rightarrow P_h = E_h\{\widetilde{P}/\widetilde{X}\} : h \in I\}$ be the output equation set associated with an application of UFI. Then, the

context of the i-th equation goal, $b_i \rightarrow P_i = E_i\{\widetilde{P}/\widetilde{X}\} \in \mathcal{O}$, is defined to be $\mathcal{C}_i = \mathcal{O} \setminus \{b_i \rightarrow P_i = E_i\{\widetilde{P}/\widetilde{X}\}\}$. The context of an equation goal specifies a behavioural constraint, helping equation solving (see §7). It is *not* an actual hypothesis: *Clam* does not use the context in the application of a proof procedure or an inference rule. The context is an annotation and so is marked with a dashed box, $\boxed{\mathcal{C}}$, to make it distinguishable (c.f. the output parts of Figure 2.) We use context(Γ) to denote the context embedded within a set of hypotheses Γ.

We have implemented the generalisation strategy as a proof critic and associated it with the `unique` method. This way, whenever `unique` is partially applicable (c.f. the applicability preconditions in Figure 3), the `generalisation` critic will be applied. Moreover, if it is applied, an application of `unique` will follow immediately after.

6.2. Example Applications of Unique and Generalise

Some example applications of `unique` would be illustrative. Consider again the verification problems introduced in §4.1. We see that when (3), $\forall n : \text{nat}.\ C^{(n)} = \mathit{Counter}(n)$, is input, `unique` becomes applicable, returning:

$$\vdash\ n = 0 \rightarrow C^n = \mathit{inc}.C^{s(n)} + \mathit{zero}.C^n$$
$$\vdash\ n \neq 0 \rightarrow C^n = \mathit{inc}.C^{s(n)} + \mathit{dec}.C^{p(n)}$$

By contrast, when (2), $\vdash \forall n : \text{nat}.\ (n \neq 0 \wedge k = 0)\ \rightarrow B^{(n)} = \mathit{Buf}(n, k)$, is input, it is not applied, because $B^{(n)}$ is not defined in terms of the indexing scheme, $k \in \{1, \ldots, n\}$. `Induction`, in that case, is attempted, yielding:

$$\vdash\ 0 \neq 0 \rightarrow B^{(0)} = \mathit{Buf}(0, 0) \quad (4)$$
$$n \neq 0 \rightarrow B^{(n)} = \mathit{Buf}(n, 0)\ \vdash\ s(n) \neq 0 \rightarrow B^{s(n)} = \mathit{Buf}(s(n), 0) \quad (5)$$

This is just as required, since it is not obvious what process substitution should be used in an application of UFI to (2).

6.2.1. *An Example Generalisation*

We now provide an example application of `generalisation` by providing intermediate results obtained by *Clam*, when making an attempt to find a proof of (2). After the application of `induction`, `elementary` was used to trivially establish the base case (4). Then, in the step case, (5), `wave` followed by `fertilise` produced the following subgoal:

$$\ldots \vdash B^{\frown}\mathit{Buf}(n, 0) = \mathit{Buf}(s(n), 0)$$

Since $B^\frown \mathit{Buf}(n,0)$ is not an indexed family of expressions, `unique` was not applicable.[8] Then, `generalise` was applied, returning the following new verification problem:

$$\ldots \vdash \mathcal{F}_\mathcal{P}(s(n),0) = \mathit{Buf}(s(n),0)$$

where $\mathcal{F}_\mathcal{P}$ was at this point given by:[9]

$$\begin{aligned} j = 0 &\rightarrow \mathcal{F}_\mathcal{P}(s(n),j) \stackrel{\text{def}}{=} B^\frown \mathit{Buf}(n,j) \\ s(n) > j \wedge j \neq 0 &\rightarrow \mathcal{F}_\mathcal{P}(s(n),j) \stackrel{\text{def}}{=} \mathcal{M}_1(\mathcal{M}_{11}(B,B'), \mathit{Buf}(n,\mathcal{M}_{12}(j))) \\ s(n) = j \wedge j \neq 0 &\rightarrow \mathcal{F}_\mathcal{P}(s(n),j) \stackrel{\text{def}}{=} \mathcal{M}_2(\mathcal{M}_{21}(B,B'), \mathit{Buf}(n,\mathcal{M}_{22}(j))) \end{aligned}$$

With this generalised conjecture, the `unique` method was applied immediately after, yielding the output equation set below:

$$\begin{aligned} &\vdash j = 0 \rightarrow \mathcal{F}_\mathcal{P}(s(n),j) = in.\mathcal{F}_\mathcal{P}(s(n),s(j)) \\ &\vdash s(n) > j \wedge j \neq 0 \rightarrow \\ &\qquad \mathcal{F}_\mathcal{P}(s(n),j) = in.\mathcal{F}_\mathcal{P}(s(n),s(j)) + \overline{out}.\mathcal{F}_\mathcal{P}(s(n),p(j)) \quad (6) \\ &\vdash s(n) = j \wedge j \neq 0 \rightarrow \mathcal{F}_\mathcal{P}(s(n),j) = \overline{out}.\mathcal{F}_\mathcal{P}(s(n),p(j)) \end{aligned}$$

Then, *Clam*, extended with the equation solving strategy (to be introduced next, §7), was capable of producing the following (second-order) substitutions:

- $B^\frown \mathit{Buf}(n,j)/\mathcal{M}_1(\mathcal{M}_{11}(B,B'), \mathit{Buf}(n,\mathcal{M}_{12}(j)))$
- $B'^\frown \mathit{Buf}(n,j)/\mathcal{M}_2(\mathcal{M}_{21}(B,B'), \mathit{Buf}(n,\mathcal{M}_{22}(j)))$

So, $\mathcal{F}_\mathcal{P}$ was found to be given by:

$$\begin{aligned} j = 0 &\rightarrow \mathcal{F}_\mathcal{P}(s(n),j) \stackrel{\text{def}}{=} B^\frown \mathit{Buf}(n,j) \\ s(n) > j \wedge j \neq 0 &\rightarrow \mathcal{F}_\mathcal{P}(s(n),j) \stackrel{\text{def}}{=} B^\frown \mathit{Buf}(n,j) \\ s(n) = j \wedge j \neq 0 &\rightarrow \mathcal{F}_\mathcal{P}(s(n),j) \stackrel{\text{def}}{=} B'^\frown \mathit{Buf}(n,n) \end{aligned}$$

[8] Notice that the associated output set cannot be the one gained by sending $\mathit{Buf}(s(n),j)$ to $B^\frown \mathit{Buf}(n,j)$, for all $0 \leq j \leq s(n)$, in the defining equations of *Buf*:

$$\begin{aligned} &\vdash (s(n) \neq 0 \wedge k = 0) \rightarrow B^\frown \mathit{Buf}(n,k) = in.(B^\frown \mathit{Buf}(n,s(k))) \\ &\vdash (s(n) > k \wedge k \neq 0) \rightarrow \\ &\qquad B^\frown \mathit{Buf}(n,k) = in.(B^\frown \mathit{Buf}(n,s(k))) + \overline{out}.(B^\frown \mathit{Buf}(n,p(k))) \\ &\vdash (s(n) = k \wedge k \neq 0) \rightarrow B^\frown \mathit{Buf}(n,k) = \overline{out}.(B^\frown \mathit{Buf}(n,p(k))) \end{aligned}$$

This is because the second and third identities are not theorems ($\mathit{Buf}(n,k)$ is not defined for $n < k$.)

[9] Note that $\mathcal{M}_{11}(B,B')$ sufficed to represent the state space of B: $\{B,B'\}$, and similarly for $\mathcal{M}_{21}(B,B')$ and $\mathit{Buf}(n,\mathcal{M}_{22}(j))$.

just as required. With this, we complete our presentation of when and how the UFI rule is applied; now we draw attention to the equation solving strategy.

7. Equation Verification = Equation Solving

In this section, we present a search control strategy to automatically solve the output equation set. Each equation in this set takes the form: $\Gamma \vdash b_i \rightarrow P_i = S_i$, where P_i is a concurrent form and S_i a sum of prefixed terms, some of which contain meta-variables.

Except for this latter fact, each equation to be solved poses the kind of problem dealt with by `equation`. This method, as explained in §5.4, attempts to establish an identity, first by proving that both source, P_i, and target, S_i, offer the same initial capabilities of interaction (using `expansion` and `absorption`), and second by proving that, after action execution, source and target evolve into equal agents (using `goalsplit` and `action`). Our equation solving strategy has been designed in the context of an application of `action`, since this enables us to focus meta-variable instantiation using a look-ahead strategy.

7.1. The Equation Solving Strategy

The equation solving strategy thus attempts to solve goals of the form:

$$\Gamma \vdash \alpha.P = \alpha.\mathcal{M}$$

It uses this heuristic: *solve $P/\mathcal{M}$, only if the initial behaviour of P matches the expected initial behaviour of $\mathcal{M}$*. Equation solving thus relies on a look-ahead strategy. To compute the behaviour of P, the look-ahead strategy simply applies process expansion. But, to speculate the behaviour of $\mathcal{M}$, it has recourse to the context: Take $b_h \rightarrow P_h = E_h\{\widetilde{P}/\widetilde{X}\} \in \text{context}(\Gamma)$; then, *assume* that the behaviour of $\mathcal{M}$ is given by the equation body, $E_h\{\widetilde{P}/\widetilde{X}\}$, under the proviso that $\Gamma \vdash b_h$ holds.

If it is not possible to speculate the behaviour of $\mathcal{M}$, then equation solving will fail (and so will this proof planning attempt). If it is and the behaviour of $\mathcal{M}$ and P match, equation solving will solve $P/\mathcal{M}$; otherwise, the equation solving strategy will attempt to plan a proof of $\vdash P = \tau.\mathcal{M}$.[10] The equation solving strategy is defined in Figure 4; it is given as a proof critic, `equation_solving`, associated with the `action` method.

[10] The alternative goal, $\vdash \tau.P = \mathcal{M}$, is not considered. This is because, in this case, no method applies, thus yielding failure in the proof planning attempt.

Input:
$\Gamma \vdash \alpha.P = \alpha'.S$
Preconditions:
$\alpha = \alpha'$, transitions(P, Ps), % no transitions can be % derived from S,
metavar(S), % since it is a meta-variable
$\exists C_i \in C_s.\, \Gamma \vdash C_i$ % which is currently solvable
where $Cs = \{C_h : C_h \to P_h = E_h\{\widetilde{P}/\widetilde{X}\} \in \text{context}(\Gamma)\}$
Effects:
Let $C = C_i$ and $E = E_i\{\widetilde{P}/\widetilde{X}\}$
Let $\Delta_1 = \{\mu : \exists P'.\, P \stackrel{\mu}{\Rightarrow} P'\}$ and $\Delta_2 = \{\mu : \exists E'.\, E \stackrel{\mu}{\Rightarrow} E'\}$
Let $G' = \begin{cases} (P = S)\{P/S\} & \text{if } \Delta_1 = \Delta_2 \\ P = \tau.S & \text{otherwise} \end{cases}$
If applicable, update the process family: $\mathcal{F}_{\mathcal{P}}\{P/S\}$
Set new goal:
$\Gamma \vdash G'$
Meanings of the meta-logic terms:
- metavar(X) holds if X is a meta-variable.

Figure 4. The `equation_solving` critic

Using the unique method and the equation solving strategy, full automation was achieved in proving (5). A snapshot of the obtained proof plan follows. The snapshot starts where equation solving had been used already to successfully fix $\mathcal{M}_1(\mathcal{M}_{11}(B, B'), \mathit{Buf}(n, \mathcal{M}_{12}(j)))$ to $B^\frown \mathit{Buf}(n, j)$. Then the working subgoal was (6) (see page 21):

$$\begin{array}{rcl} j = 0 & \to & \mathcal{F}_{\mathcal{P}}(s(n), j) = in.\mathcal{F}_{\mathcal{P}}(s(n), s(j)) \\ s(n) > j \wedge j \neq 0 & \to & \mathcal{F}_{\mathcal{P}}(s(n), j) = \\ & & \quad in.\mathcal{F}_{\mathcal{P}}(s(n), s(j)) + \overline{out}.\mathcal{F}_{\mathcal{P}}(s(n), p(j)) \\ s(n) = j \wedge j \neq 0 & \to & \mathcal{F}_{\mathcal{P}}(s(n), j) = \overline{out}.\mathcal{F}_{\mathcal{P}}(s(n), p(j)) \end{array}$$

$n : \text{nat}, n \neq 0$
$\vdash s(n) > k \wedge k \neq 0; \to$
$\quad B^\frown \mathit{Buf}(n, k) = in.\mathcal{F}_{\mathcal{P}}(s(n), s(k)) + \overline{out}.\mathcal{F}_{\mathcal{P}}(s(n), p(k))$

$\mathcal{F}_{\mathcal{P}}$ was at this point given as follows:

$$\begin{array}{rcl} j = 0 & \to & \mathcal{F}_{\mathcal{P}}(s(n), j) \stackrel{\text{def}}{=} B^\frown \mathit{Buf}(n, j) \\ s(n) > j \wedge j \neq 0 & \to & \mathcal{F}_{\mathcal{P}}(s(n), j) \stackrel{\text{def}}{=} B^\frown \mathit{Buf}(n, j) \\ s(n) = j \wedge j \neq 0 & \to & \mathcal{F}_{\mathcal{P}}(s(n), j) \stackrel{\text{def}}{=} \mathcal{M}_2(\mathcal{M}_{21}(B, B'), \mathit{Buf}(n, \mathcal{M}_{22}(j))) \end{array}$$

With this goal, a further application of `equation_solving` was required. It occurred after the application of `casesplit` upon the partition: $[n > k, n = k]$, suggested by the need of unfolding the definition of *Buf*. The interesting case is $n = k$, where *Clam* applied `sym_eval`, followed by `expansion` (see §5.3), leaving:

$$\begin{array}{l} \ldots, s(n) > k, k \neq 0, n = k \\ \vdash in.(B'^{\frown} Buf(n,k)) + \overline{out}.(B^{\frown} Buf(n, p(k))) \\ \quad = in.\mathcal{M}_2(\mathcal{M}_{21}(B, B'), Buf(n, \mathcal{M}_{22}(k))) + \overline{out}.(B^{\frown} Buf(n, p(k))) \end{array}$$

Then, `goalsplit` yielded two trivially provable subgoals, one for each action. The *in* subgoal was as follows:

$$\ldots \vdash in.(B'^{\frown} Buf(n,k)) = in.\mathcal{M}_2(\mathcal{M}_{21}(B, B'), Buf(n, \mathcal{M}_{22}(k)))$$

Then, the `equation_solving` critic was applicable, since:

- for $n = k \wedge k \neq 0$, $\{\mu : \exists P'.\, B'^{\frown} Buf(n,k) \stackrel{\mu}{\Rightarrow} P'\} = \{\overline{out}\}$, according to process expansion, and
- for $s(n) = s(k) \wedge s(k) \neq 0$, $\{\mu : \exists P'.\, \mathcal{F}_{\mathcal{P}}(s(n), s(k)) \stackrel{\mu}{\Rightarrow} P'\} = \{\overline{out}\}$, according to the context.

Second-order matching suggested the following substitution:

$$\mathcal{M}_2 \mapsto {}^{\frown},\ \mathcal{M}_{21} \mapsto \mathrm{inr}(B, B'),\ \mathcal{M}_{22} \mapsto \lambda x.x$$

where inr, a selector function, is given by $\mathrm{inr}(x, y) = y$, yielding the refinement: $s(n) = j \wedge j \neq 0 \rightarrow \mathcal{F}_{\mathcal{P}}(s(n), j) \stackrel{\mathrm{def}}{=} B'^{\frown} Buf(n, n)$, which is just as required.

8. Experimental Results: a Summary

Table III reports some of the example verification conjectures against which we tested our proof plan. The proofs are all totally automated. However, our results are not state-of-the-art for manual, mechanical proof. PT stands for the total elapsed planning time, given in seconds. Table III is connected to tables IV and V: Verification conjecture #i, specification #i and process definition #i all correspond. So, for each verification conjecture, we provide the definition of the program, together with the definition of its subcomponents.

Each verification conjecture can be described in terms of the program under analysis:

Table III. Verification Conjectures and Planning Time

#	Verification Conjecture	PT
1	$\forall n : \text{nat}.\ B(n) = \mathit{Counter}(n)$	300.75
2	$\forall n : \text{nat}.\ U^{(n)} = \mathit{Counter}(n)$	38.95
3	$\forall n : \text{nat}.\ C^{(n)} = \mathit{Counter}(n, 0)$	56.933
4	$\forall l : \text{list}.\ S^{(l)} = \mathit{Stack}(l)$	54.55
5	$\forall l : \text{list}.\ Q^{(l)} = \mathit{Queue}(l)$	66.083
6	$\forall n : \text{nat}.\ C^{(n)} = S(0, n, \text{nil})$	1263.983
7	$\forall l : \text{list}.\ l \neq \text{nil} \rightarrow \prod_{i \in l}(S, l) = \mathit{Sem}(\text{length}(l), 0)$	43.17

Examples #1–3: Systems #1 to #3 are all counters. System #1 is a binary dynamic counter, with its least significant bit first (little endian). System #2 is a Basic Parallel Processes (BPP) counter. System #3 is a static counter, with fixed but arbitrary length; it may count sequences of events from 0 to length-1. System #1 and system #2 are both infinite-state, while system #3 shows parameterised behaviour.

A proof plan of verification conjecture #3 outputs the synthesis of a new process family, given in Table VI.

Examples #4–5: Systems #4 and #5 are respectively a dynamic stack and a dynamic queue. Not only are they infinite-state, but they are also value-passing. The associated verification conjecture specifies the expected behaviour.

Example #6: System #6 is an n-size sorting machine with parameterised behaviour. The sorting machine makes use of insertion sort.

The synthesised process family is given in Table VI.

Example #7: System #7 specifies a network that comprises a fixed but arbitrary number of semaphores. The formalisation of the $\prod$ meta-level operator has been borrowed from (Korver and Springintveld, 1994).

As a side result, the proof plan synthesised a new process family, given in Table VI.

The success rate of the proof plan was 83% tested on 26 verification examples. Failure pertained to the inability of the verification plan to handle the general equivalence problem, as it is geared towards a

Table IV. Process Specifications

#	Specification
1	$n = 0 \rightarrow \ Counter(n) \ = \ inc.Counter(s(n)) + zero.Counter(n)$ $n \neq 0 \rightarrow \ Counter(n) \ = \ inc.Counter(s(n)) + dec.Counter(pred(n))$
2	$n = 0 \ \rightarrow \ Counter(n) \ \stackrel{\text{def}}{=} \ up.Counter(s(n))$ $n \neq 0 \ \rightarrow \ Counter(n) \ \stackrel{\text{def}}{=} \ up.Counter(s(n)) + down.Counter(pred(n))$
3	$n > k \ \rightarrow \ Counter(n,k) \ \stackrel{\text{def}}{=} \ in.Counter(n, s(k))$ $n = k \ \rightarrow \ Counter(n,k) \ \stackrel{\text{def}}{=} \ empty.Counter(n, 0)$
4	$l = \text{nil} \ \rightarrow \ Stack(l) \ = \ push(n).Stack(n :: l) + empty.Stack(l)$ $l \neq \text{nil} \ \rightarrow \ Stack(l) \ = \ push(n).Stack(n :: l) + \overline{pop}(hd(l)).Stack(tl(l))$
5	$l = \text{nil} \ \rightarrow \ Queue(l) \ = \ push(n).Queue(n :: l) + empty.Queue(l)$ $l \neq \text{nil} \ \rightarrow \ Queue(l) \ = \ push(n).Queue(l <> n :: \text{nil}) + \overline{out}(hd(l)).Queue(tl(l))$
6	$n = 0 \ \rightarrow \ S(0,n,l) \ \stackrel{\text{def}}{=} \ \overline{out}(0).S(0,n,l)$ $n \neq 0 \ \rightarrow \ S(0,n,h :: l) \ \stackrel{\text{def}}{=} \ \overline{out}(\max\ h :: l).S(0,n,\text{del}(\max\ h :: l,\ h :: l))$ $n \neq 0 \ \rightarrow \ S(s(m),n,l) \ \stackrel{\text{def}}{=} \ in(x).S(m,n,x :: l) \ \text{ with } x > 0$ $D(0,n,\text{nil}) \ \stackrel{\text{def}}{=} \ \overline{out}(0).S(n,n,\text{nil})$ $D(0,n,h :: l) \ \stackrel{\text{def}}{=} \ \overline{out}(\max\ h :: l).S(0,n,\text{del}(\max\ h :: l,\ h :: l))$
7	$n \neq 0 \ \rightarrow \ Sem(n,0) \ \stackrel{\text{def}}{=} \ get.Sem(n, s(0))$ $n > s(k) \ \rightarrow \ Sem(n,s(k)) \ \stackrel{\text{def}}{=} \ get.Sem(n, s(s(k))) + put.Sem(n,k)$ $n = s(k) \ \rightarrow \ Sem(n,s(k)) \ \stackrel{\text{def}}{=} \ put.Sem(n,k)$

specific class of (equivalence-based) verification problems. In our work, specifications are assumed to constitute a declaration, where the body of each defining equation consists of a sum of prefixed processes; programs may be arbitrary CCS terms or functions yielding a CCS term. During verification search, proof planning may return an equivalence problem involving two concurrent forms and so no method will be applicable and thus proof planning will break down next. A failed example related to this proof plan anomaly is Milner's jobshop (Milner, 1989).

Failure also occurred in system exhibiting process divergence. A process is said to be *divergent*, if it contains a τ-cycle, e.g., $P = \tau.P$. An example system where divergence causes verification failure is the alternate bit communication protocol. Negative results arose also in systems containing a network invariant. As the name suggest, the *invariant* of a network captures that part of the network behaviour which

will not change even if the size of the network does. Milner's cycler is an example system with a network invariant.

The average total elapsed planning time was 750 seconds, with a standard deviation of 345. The test was run on a Solbourne 6/702, dual processor, 50MHz, SuperSPARC machine with 128 Mb of RAM. The operating system, Solbourne OS/MP, is an optimised symmetric multi-processing clone of SunOS 4.1. The full test set is available at `http://homepage.cem.itesm.mx/raulm/pub/plan.pdf`.

The proof plan was successfully tested. Some of the successful example verification problems are shown in Table III. These conjectures are all outside the scope of current *automated* verification tools. While our strategy is automatic, it cannot be turned into a decision procedure. This is because in general the problem is undecidable, our heuristics may fail or because there may be theorems for which there is not a proof expressed as a verification plan. These features are not shared by many of the related approaches.

9. Related and Future Work

Semantic-based tools, such as the Concurrency Workbench (Cleaveland et al., 1989) and the FC2Tools package (Bouali et al., 1996), support automatic, equivalence-based verification of finite-state systems (FSS). These tools are all restricted to a very simple process language. Value-passing, even if confined to finite data domains, as, e.g., in CADP,[11] is not directly expressible. Instead, the use of a language translator has been adopted—see, for example, Bruns's work (Bruns, 1991) and the Concurrency Factory (Cleaveland et al., 1996), a "next-generation Concurrency Workbench". By contrast, not only do we deal with FSS, but we also deal with value-passing, parameterised systems.

Semantic-based tools are exceedingly successful in the analysis of FSS. They incorporate very efficient algorithms, capable of deciding process equivalence in polynomial time. Ongoing work is thus concerned with giving the verification plan algorithms to try to determine which behavioural class an agent belongs to. When necessary, the system would be in the position of invoking a suitable decision procedure.

9.1. Symbolic Bisimulations

Symbolic bisimulations (Hennessy and Lin, 1995) were especially designed to reason about value-passing. They are used to study agents that can only be expressed by means of an infinite transition graph in

[11] `http://www.inrialpes.fr/vasy/cadp/`

Table V. Process Definitions

#	System	Agents
1	$B^{(0)} = B$ $B^{s(2\times n)} = D_0{}^\frown B^{(n)}$ $B^{s(s(2\times n))} = D_1{}^\frown B^{(n)}$ $B^{s(n)} = V^{(n)\frown} B^{\mathrm{half}(n)}$	$B \stackrel{\text{def}}{=} inc.(D_0{}^\frown B) + zero.B$ $Cy \stackrel{\text{def}}{=} \overline{i}.D_0$ $D_1 \stackrel{\text{def}}{=} inc.Cy + dec.D_0$ $D_0 \stackrel{\text{def}}{=} inc.D_1 + dec.Bw$ $Bw \stackrel{\text{def}}{=} \overline{d}.D_1 + \overline{z}.B$
	$P^\frown Q \stackrel{\text{def}}{=} (P[i'/i, z'/z, d'/d] \mid Q[i'/inc, z'/zero, d'/dec]) \setminus \{i', z', d'\}$	
2	$U^0 = U$ $U^{s(n)} = U^n \mid down.\mathbf{0}$	$U \stackrel{\text{def}}{=} up.(U \mid down.\mathbf{0})$
3	$C^{(0)} = B$ $C^{s(n)} = C{}^\frown C^{(n)}$	$B \stackrel{\text{def}}{=} empty.B$ $C \stackrel{\text{def}}{=} in.C'$ $C' \stackrel{\text{def}}{=} \overline{up}.C + \overline{down}.C''$ $C'' \stackrel{\text{def}}{=} empty.C$
	$P^\frown Q \stackrel{\text{def}}{=} (P[i/up, d/down] \mid Q[i/in, d/empty]) \setminus \{i, d\}$	
4	$S^{(\mathrm{nil})} = B$ $S^{(h::t)} = C_{(h)}{}^\frown S^{(t)}$	$B \stackrel{\text{def}}{=} push(n).(C_n{}^\frown B) + empty.B$ $C_{(n)} \stackrel{\text{def}}{=} push(x).(C_x{}^\frown C_n) + \overline{pop}(n).D$ $D \stackrel{\text{def}}{=} o(x).C_x + \overline{e}.B$
	$P^\frown Q \stackrel{\text{def}}{=} (P[i'/i, o'/o, e'/e] \mid Q[i'/push, o'/pop, e'/empty]) \setminus \{i', o', e'\}$	
5	$Q^{(\mathrm{nil})} = B$ $Q^{(h::t)} = C_{(h)}{}^\frown Q^{(t)}$	$B \stackrel{\text{def}}{=} in(n).(C_{(n)}{}^\frown B) + \overline{empty}.B$ $C_{(n)} \stackrel{\text{def}}{=} in(x).\overline{i}(x).C_{(n)} + \overline{out}(n).D$ $D \stackrel{\text{def}}{=} o(n).C_n + \overline{e}.B$
	$P^\frown Q \stackrel{\text{def}}{=} (P[i'/i, o'/o, e'/e] \mid Q[i'/push, o'/pop, e'/empty]) \setminus \{i', o', e'\}$	
6	$C^{(0)} = B$ $C^{s(n)} = C{}^\frown C^{s(n)}$	$B \stackrel{\text{def}}{=} \overline{out}(0).B$ $C \stackrel{\text{def}}{=} in(x).C'_x$ $C'_x \stackrel{\text{def}}{=} \overline{d}(x).C + up(y).I(x,y)$ $I(x,y) \stackrel{\text{def}}{=} \overline{out}(\max\ x, y).C''_{\min\ x,y}$ $x = 0 \rightarrow C''_x \stackrel{\text{def}}{=} \overline{out}(x).C$ $x \neq 0 \rightarrow C''_x \stackrel{\text{def}}{=} C'_x$
	$P^\frown Q \stackrel{\text{def}}{=} (P[u/up, d/down] \mid Q[d/in, u/out]) \setminus \{u, d\}$	
7	$\prod_{i\in l}(S, l)$	$S \stackrel{\text{def}}{=} get.V \qquad V \stackrel{\text{def}}{=} put.S$ $\prod(P, \mathrm{nil}) = \mathbf{0}$ $\prod(P, h :: t) = P(h) \mid \prod(P, t)$

Table VI. Synthesised process families

#	Process Family
3	$s(n) > k \;\rightarrow\; \mathcal{F}_{\mathcal{P}}(s(n),k) \stackrel{\text{def}}{=} C^\frown \mathit{Counter}(n,k)$ $s(n) = k \;\rightarrow\; \mathcal{F}_{\mathcal{P}}(s(n),k) \stackrel{\text{def}}{=} C''^\frown \mathit{Counter}(n,0)$
6	$\mathcal{F}_{\mathcal{P}}(0,s(n),h::l) \stackrel{\text{def}}{=} (\overline{out}(\max\{h,\max\ l\}).C''_{\min\{h,\min\ l\}})^\frown D(0,n,\text{del}(\max\ l,\ l))$ $\mathcal{F}_{\mathcal{P}}(s(m),s(n),l) \stackrel{\text{def}}{=} C^\frown S(m,n,l)$ $\mathcal{F}_{\mathcal{P}}{}'(0,s(n),\text{nil}) \stackrel{\text{def}}{=} (\overline{out}(0).C)^\frown D(0,n,\text{nil})$ $\mathcal{F}_{\mathcal{P}}{}'(0,s(n),h::l) \stackrel{\text{def}}{=} (\overline{out}(\max\{h,\max\ l\}).C''_{\min\{h,\min\ l\}})^\frown D(0,n,\text{del}(\max\ l,\ l))$
7	$s(n) \neq 0 \;\rightarrow\; \mathcal{F}_{\mathcal{P}}(s(n),0) \stackrel{\text{def}}{=} S \mid \mathit{Sem}(n,0)$ $s(n) > k \;\rightarrow\; \mathcal{F}_{\mathcal{P}}(s(n),k) \stackrel{\text{def}}{=} S \mid \mathit{Sem}(n,k)$ $s(n) = k \;\rightarrow\; \mathcal{F}_{\mathcal{P}}(s(n),k) \stackrel{\text{def}}{=} V \mid \mathit{Sem}(n,n)$

the underlying (standard) interpretation. Thus, the symbolic bisimulation deals with problems beyond the scope of standard bisimulation tools.

Symbolic bisimulations are binary relations parameterised by an interpretation or possible world. The interpretation captures the state of the entire system, and contains explicit information about the data that have been exchanged. A relation holds only if the associated interpretation is satisfiable. Proving the satisfiability of an interpretation may become a difficult task requiring the use of powerful theorem provers. By contrast, providing a unified verification framework has been one of the main emphases of our work.

Hennessy and Lin (1995) consider systems that have infinite state space or show parameterised behaviour. Both the syntax and operational semantics allow parameterised recursive process definitions. The value-passing process language is *parameterised on the data language*. Thus, primitive recursive functions can be used to represent either a process family, or the structure of a system at any state during execution, so long as an appropriate data language is provided.

Hennessy and Lin (1995) are able to reason about value-passing agents. By comparison, we include nothing but a modest, less powerful mechanism, centred around a subset of the full language. For example, we do not include the **if-then-else** operator. So again in this respect our work is complementary.

Future work contemplates the use of more expressive languages and more powerful specifications of the UFI rule of inference. (Hennessy and Lin, 1997) have formulated unique fixpoint induction for value-passing process calculi. Their work has been extended by (Rathke, 1997).

9.2. Kurshan and McMillan's Induction Theorem

More closely related to ours is the work of Kurshan and McMillan's. Rather than a method, they propose a methodology for reasoning about networks containing a number of similar FSS (Kurshan and McMillan, 1989). Their methodology is to simplify one such network into another containing a fixed, manageable number of components. Central to the methodology is an induction theorem, which states the conditions governing the simplification precisely. For the induction theorem to be applicable, the invariant of the network must be provided.

Kurshan and McMillan provide no method for generating network invariants, but they do suggest heuristics. Thus, the Kurshan and McMillan method is semi-automatic, relying on the user to provide the invariant details, whereas ours is totally automatic. Moreover, the method applies only if the network contains repeated components, and it cannot handle infinite-state systems. By comparison, we do not impose any restriction on the process structure, or process behaviour.

9.3. μCRL: Reasoning About $n+1$ Processes

μCRL (micro Common Representation Language) (Groote and Ponse, 1995; Groote and Ponse, 1994) extends ACP with value-passing, taking account of data in the analysis of communicating systems. μCRL handles data by means of equational abstract data types. The language features value-passing actions, parameterised agents, an if-then-else construct and quantification over summation of possibly infinitely many data elements of a data type.

In μCRL, the analysis of communicating systems is based only on the proof theory (axioms and proof principles), as opposed to a semantics-based approach. Process verification is thus suitable to be mechanised by proof checkers. The cones and foci method (Groote and Springintveld, 2001) is the crux of the μCRL process verification methodology. It has been applied to verify a number of large protocols. The cones and foci method reduces the verification problem (the proof of an equality) to a proof of properties of data parameters. Discovering these properties, including invariants, however, requires human intervention.

(Groote and Reniers, 2001) have approached the verification of the parallel composition of n processes using μCRL. Instead of collapsing the compound process (like Kurshan and McMillan do), Groote and

Reniers extract a linear process equation from it. The linearisation theorem yields a proof method, which is applicable only to networks of linear processes (containing no parallel composition). The application of the linearisation theorem is driven by the cones and foci methodology and so completing the verification process may require user intervention to provide the necessary properties about data parameters. By contrast, our approach is fully automated and does not impose any restriction on the structure of processes; however, it makes use of a few simpler inductive rules.

9.4. Mechanisation of CCS in Proof Checkers

CCS has been mechanised in several theorem provers. For example, Cleaveland and Panangaden describe an implementation of CCS in Nuprl (Cleaveland and Panangaden, 1988). Cleaveland and Panangaden did not strive for automation; instead, they were interested in showing the suitability of type theory for reasoning about concurrency in general.

Also Nesi implemented CCS in HOL (Nesi, 1992). But Nesi's motivation was somewhat different: to show the suitability of (induction oriented) proof checkers for reasoning about parameterised systems. The tool was the first to accommodate the analysis of parameterised systems, but did not improve upon the degree of automation. Nesi extended the pure subset of CCS to its value-passing version (Nesi, 1999). Following Milner's approach, Nesi gave value-passing agents behavioural semantics by translating them into pure agents. The extended proof environment deals with the verification of value-passing systems, which can be defined over an infinite value domain.

In a vein similar to that of Nesi, Lin reported an implementation of a CCS like value-passing process algebra in VPAM (Lin, 1993), a generic proof checker. VPAM was never intended to provide any level of automation, though users may create and execute their own tactics.

These proof frameworks are highly interactive, requiring at each step the user to select the tactic to be applied. By contrast, our verification planner is fully automatic, accommodating parameterised, infinite-state systems. However, unlike VPAM, we do not accommodate truly value-passing systems.

Future work is concerned with taking advantage of these and other implementations of CCS. The Mathematical Reasoning Group at Edinburgh and the Automated Reasoning Group at Cambridge ran a joint project to link the systems *clam* and HOL (Boulton et al., 1998). This linkage will allow to import into *clam* Nesi's implementation of CCS, as well as using numerous decision procedures.

10. Conclusions

The Verification planner handles the search control problems prompted by the use of the UFI rule of inference more than satisfactorily. We have planned proofs of conjectures that previously required human interaction. Full automation is an unattainable ideal, but we should nevertheless strive towards it. It is worth applying the verification planner to larger, industrial strength examples, and see what further extensions are required.

References

Bergstra, J. A. and J. W. Klop: 1985, 'Algebra of Communicating Processes with Abstraction'. *Theoretical Computer Science* **37**(1), 77–121.

Bouali, A., A. Ressouche, V. Roy, and R. de Simone: 1996, 'The FC2TOOLS Set'. In: R. Alur and T. Henzinger (eds.): *Proceedings of the 8th Conference on Computer-Aided Verification, CAV'96*. pp. 441–45, Springer-Verlag. Lecture Notes in Computer Science, Vol. 1102.

Boulton, R., K. Slind, A. Bundy, and M. Gordon: 1998, 'An Interface between CLAM and HOL'. In: J. Grundy and M. Newey (eds.): *11th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'98)*. Camberra, Australia, pp. 87–104, Springer-Verlag. Lecture Notes in Computer Science, Vol. 1479.

Bruns, G.: 1991, 'A language for value-passing CCS'. LFCS Report Series ECS-LFCS-91-175, Department of Computer Science, University of Edinburgh.

Bundy, A.: 1988, 'The Use of Explicit Plans to Guide Inductive Proofs'. In: R. Lusk and R. Overbeek (eds.): *Proceedings of the 9th Conference on Automated Deduction*. Argonne, Illinois, USA, pp. 111–120, Springer-Verlag. Also available from Edinburgh as DAI Research Paper No. 349.

Bundy, A., A. Smaill, and J. Hesketh: 1990a, 'Turning eureka steps into calculations in automatic program synthesis'. In: S. L. Clarke (ed.): *Proceedings of UK IT 90*. pp. 221–6. Also available from Edinburgh as DAI Research Paper 448.

Bundy, A., A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill: 1993, 'Rippling: A Heuristic for Guiding Inductive Proofs'. *Artificial Intelligence* **62**, 185–253. Also available from Edinburgh as DAI Research Paper No. 567.

Bundy, A., F. van Harmelen, J. Hesketh, and A. Smaill: 1991, 'Experiments with Proof Plans for Induction'. *Journal of Automated Reasoning* **7**, 303–324. Earlier version available from Edinburgh as DAI Research Paper No 413.

Bundy, A., F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens: 1989, 'A Rational Reconstruction and Extension of Recursion Analysis'. In: N. S. Sridharan (ed.): *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. pp. 359–365, Morgan Kaufmann. Also available from Edinburgh as DAI Research Paper No. 419.

Bundy, A., F. van Harmelen, C. Horn, and A. Smaill: 1990b, 'The Oyster-Clam System'. In: M. E. Stickel (ed.): *Proceedings of the 10th International Conference on Automated Deduction*. pp. 647–648, Springer-Verlag. Also available from Edinburgh as DAI Research Paper No. 507.

Cleaveland, R., P. J., and B. Steffen: 1989, 'The Concurrency Workbench: A Semantics-based Verification Tool for Finite-State Systems'. In: *Proceedings*

of the Workshop on Automated Verification Methods for Finite-State Systems. Springer-Verlag.

Cleaveland, R., P. Lewis, S. Smolka, and O. Sokolsky: 1996, 'The Concurrency Factory: A Development Environment for Concurrent Systems'. In: R. Alur and T. Henzinger (eds.): *Proceedings of the 8th Conference on Computer-Aided Verification, CAV'96*. pp. 398–401, Springer-Verlag. Lecture Notes in Computer Science, Vol. 1102.

Cleaveland, R. and P. Panangaden: 1988, 'Type Theory and Concurrency'. *International Journal of Parallel Programming* **17**(2), 153–206.

Gallier, J.: 1986, *Logic for Computer Science*. Harper & Row, New York.

Groote, J. and A. Ponse: 1994, 'Proof theory for μCRL'. In: A. et al (ed.): *Proceedings of the International Workshop on Semantics of Specification Languages*. pp. 231–250, Springer-Verlag.

Groote, J. and A. Ponse: 1995, 'The syntax and semantics of μCRL'. In: A. Ponse, C. Verhoef, and S. van Vlijmen (eds.): *Algebra of Communicating Processes 1994*. pp. 26–62, Springer-Verlag.

Groote, J. F. and M. A. Reniers: 2001, 'Algebraic Process Verification'. In: J. A. Bergstra, A. Ponse, and S. Smolka (eds.): *Handbook of Process Algebra*. Elsevier, pp. 1–66.

Groote, J. F. and J. Springintveld: 2001, 'Focus Points and Convergent Processes Operators: a Proof Strategy for Protocol Verification'. *Journal of Logic and Algebraic Programming* **49**, 31–60.

Hennessy, M. and H. Lin: 1995, 'Symbolic Bisimulations'. *Theoretical Computer Science* **138**, 353–389. Also available from Sussex as Computing Science Technical Report 1/92.

Hennessy, M. and H. Lin: 1997, 'Unique Fixpoint Induction for Message-Passing Process Calculi'. In: *Computing: The Australasian Theory Symposium (CATS'97)*. Also available from Sussex as Computing Science Technical Report 6/95.

Ireland, A.: 1992, 'The Use of Planning Critics in Mechanizing Inductive Proofs'. In: A. Voronkov (ed.): *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*. pp. 178–89, Springer-Verlag. Lecture Notes in Artificial Intelligence Vol. 624. Also available from Edinburgh as DAI Research Paper No. 592.

ISO: 1989, *Information processing systems - Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. ISO 8807.

Korver, H. and J. Springintveld: 1994, 'A Computer-Checked Verification of Milner's Scheduler'. In: *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'94)*. Sendai, Japan, pp. 161–78, Springer-Verlag. Lecture Notes in Computer Science, Vol. 789.

Kurshan, R. P. and K. McMillan: 1989, 'A structural induction theorem for processes'. In: *8th ACM Symposium on Principles Of Distributed Computing (PODC)*. New York, pp. 239–47, ACM Press.

Lin, H.: 1993, 'A Verification Tool for Value-Passing Processes'. In: *Proceedings of* 13^{th} *International Symposium on Protocol Specification, Testing and Verification*. North-Holland. Also available from Sussex as Computing Science Technical Report 8/93.

Lin, H.: 1995a, 'On Implementing Unique Fixpoint Induction for Value- Passing Processes'. In: *Proceedings of TACAS'95 Workshop on Tools and Algorithms fo r the Construction and Analysis of Systems*. Aarhus.

Lin, H.: 1995b, 'PAM: A Process Algebra Manipulator'. *Formal Methods in System Design* **7**(3), 243–259.

Milner, R.: 1989, *Communication and Concurrency*. London: Prentice Hall.

Milner, R., J. Parrow, and D. Walker: 1993, 'Mobile Logics for Mobile Processes'. *Theoretical Computer Science* **114**, 149–71. Also available from Edinburgh, as Research Report ECS-LFCS-91-136.

Monroy, R.: 1998, 'Planning Proofs of Correctness of CCS Systems'. Ph.D. thesis, Edinburgh University.

Monroy, R., A. Bundy, and I. Green: 1998, 'Planning Equational Verification in CCS'. In: D. Redmiles and B. Nuseibeh (eds.): *13th Conference on Automated Software Engineering, ASE'98*. Hawaii, USA, pp. 43–52. candidate to best paper award.

Monroy, R., A. Bundy, and I. Green: 2000a, 'Planning Proofs of Equations in CCS'. *Automated Software Engineering* **7**(3), 263–304.

Monroy, R., A. Bundy, and I. Green: 2000b, 'Searching for a solution to program verification = equation solving in CCS'. In: O. Cairó, L. Sucar, and F. Cantú (eds.): *Mexican Internation Conference on Artificial Intelligence, MICAI'00*. Acapulco, Mexico, pp. 1–12, Springer-Verlag. Lecture Notes in Artificial Intelligence, vol. 1793.

Nesi, M.: 1992, 'Mechanizing a Proof by Induction of Process Algebra Specifications in Higher-Order Logic'. In: K. G. Larsen and S. A. (eds.): *Proceedings of the 3rd International Workshop in Computer Aided Verification (CAV'91)*. Lecture Notes in Computer Science No. 575.

Nesi, M.: 1999, 'Formalising a Value-Passing Calculus in HOL'. *Formal Aspects of Computing* **11**, 160–199.

Park, D.: 1981, 'Concurrency and Automata on Infinite Sequences'. In: P. Deussen (ed.): *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. pp. 167–183. LNCS 104.

Rathke, J.: 1997, 'Unique Fixpoint Induction for Value-Passing Processes (Extended Abstract)'. In: *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*. Warsaw, Poland, pp. 140–8, IEEE Computer Society Press.