Planning Equational Verification in CCS*

Raúl Monroy
ITESM, Estado de México
Computer Science Department
Atizapán, México, 59296
raulm@campus.cem.itesm.mx

Alan Bundy Ian Green
Department of Artificial Intelligence
University of Edinburgh
Edinburgh EH1 1HN, UK
{A.Bundy,I.Green}@ed.ac.uk

Abstract

Most efforts to automate formal verification of communicating systems have centred around finite-state systems (FSSs). However, FSSs are incapable of modelling many practical communicating systems and hence there is interest in a novel class of problems, which we call VIPS, involving value-passing, infinite-state, parameterised systems. Existing approaches using model checking over FSSs are insufficient for VIPSs, due to their inability both to reason with and about domain-specific theories, and to cope with systems having an unbounded or arbitrary state space.

We use a Calculus of Communicating Systems (CCS) [13] with parameterised constants to express and specify VIPSs. We use the laws of CCS to conduct the verification task. This approach allows us to study communicating systems, regardless of their state space, and the data such systems communicate. Automating theorem proving in this system is an extremely difficult task.

We provide automated methods for CCS analysis; they are applicable to both FSSs and VIPSs. Adding these methods to the Clam proof-planner, we have implemented an automated theorem prover capable of dealing with problems outside the scope of current methods. This paper describes these methods, gives an account as to why they work, and provides a short summary of experimental results.

1. Introduction

Most efforts to automate formal verification of communicating systems have centred around finite-state systems (FSSs). A FSS is one in which system behaviour can be represented by means of a transition graph with finitely many nodes, and where value passing during interaction is disallowed; moreover, the number of system subcomponents must be known and fixed. However, FSSs are incapable of modelling many practical communicating systems and hence there is interest in a novel class of problems, which we call VIPS, involving value-passing, infinite-state, parameterised systems.

Existing approaches using model checking over FSSs are successful. However, they are insufficient for VIPSs. Ironically, the key problem with these approaches has nothing to do with state space explosion, but instead has to do with their inability to reason about systems with an unbounded or arbitrary state space; furthermore, they are unable to express and reason within domain-specific theories. Thus, there is a need for new automation paradigms, not based on an exploration of state space.

In this paper, we use a Calculus of Communicating Systems (CCS) [13] with parameterised constants to express and specify VIPSs. Moreover, we use the laws of CCS to conduct the verification task. This so-called *equational* approach is modular and extensible: we can add or remove laws at will; furthermore, we can include domain-specific theories. Thus the equational approach offers a unified framework in which we can study a wide variety of systems and the data such systems communicate.

Unfortunately, the equational approach poses significant search control issues, giving rise to an extremely difficult automation problem. In particular, the use of exhaustive rewriting without subtle heuristic control is not going to work. This is because the CCS laws often need to be used so that simple terms are replaced with more complex ones: combinatorial explosion is inevitable. To compound the problem, VIPSs often contain recursive functions and hence, a way to automate inductive theorem proving is required.

This paper makes two key contributions: First, it provides general knowledge heuristics to drive the search for a verification proof in CCS; these heuristics have the desirable quality of being applicable in the study of both FSSs and VIPSs. Second, the paper shows that by adding

*The research reported in this paper was supported by EPSRC grant GR/L/11724. First author was funded by grant SEP-REDII.

this heuristic control to an existing proof plan for inductive theorem proving (described in [2] and implemented in the *Clam* [5] proof planning system), we can build a automatic verification planner capable of dealing with problems outside the scope of existing verification tools. In this way we move to the automatic verification of CCS systems that previously required human attention.

The rest of this paper is organised as follows: §2 describes CCS: syntax, semantics, process equivalence, and the proof system used to conduct the verification task. §3 illustrates the kind of proof we shall automate, while highlighting significant search control issues. §4 gives a brief introduction to proof planning, the search engine and the framework where we capture general knowledge to automate search guidance. §5 is the core of this paper; it introduces '`Equation`' a heuristic for guiding search in this context, and shows a simple uniting example. §6 summarises results and compares `Equation` to rival techniques. Finally we draw conclusions in §7.

2. Basic CCS plus Parameterised Constants

Terms of CCS 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 communicating activities and 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 considered equivalent if their behaviour is indistinguishable to an external observer.

2.1. Syntax

The set of Actions, $\mathcal{A}ct = \{\alpha, \beta, \ldots\}$, contains the set of *names*, $\mathcal{A}$, the set of *co-names*, $\overline{\mathcal{A}}$, and the unobservable action τ, which denotes process intercommunication. $\mathcal{A}$ and $\overline{\mathcal{A}}$ are both assumed to be countable and infinite. Let $a, b, c, \ldots$ range over $\mathcal{A}$, and $\overline{a}, \overline{b}, \overline{c}, \ldots$ over $\overline{\mathcal{A}}$. The set of labels, $\mathcal{L}$, is defined to be $\mathcal{A} \cup \overline{\mathcal{A}}$; hence, $\mathcal{A}ct = \mathcal{L} \cup \{\tau\}$. Let $\ell, \ell', \ldots$ range over $\mathcal{L}$. Let K, $\overline{K}$, L, $\overline{L}$, ... denote subsets of $\mathcal{L}$.

$\mathcal{K}$ is the set of agent constants, which refer to unique behaviour and are assumed to be declared by means of the definition facility, $\stackrel{\text{def}}{=}$. Let $A, B, C, \ldots$ 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: $b \rightarrow A_{\langle x_1,\ldots,x_n\rangle} \stackrel{\text{def}}{=} E$, where E may contain no process variables, and no free value variables other than $x_1, \ldots, x_n$.

Values, in general, might be of any specified type. Value expressions e and boolean expressions b may contain value-variables $x, y, \ldots$, value-constants $v_1, v_2, \ldots$, and any operators we may require, e.g., $\times, \div, \mathit{even}, \ldots$. These extensions (which are formally no more expressive that standard CCS) allow a more succinct and natural expression of process behaviour.

The set of *agent expressions*, $\mathcal{E}$, is defined by the following abstract syntax:

$$E ::= A \mid \alpha.E \mid \sum_{i \in I} E_i \mid E \mid E \mid E \setminus L \mid E[f]$$

where f stands for a relabelling function.[1] Informally, the meaning of the *combinators* is as follows: *Prefix*, $(.)$, is used to convey discrete actions. *Summation*, $(\sum)$, disjoins the capabilities of the agents E_i, $(i \in I)$; as soon as one performs any action, the others are dismissed. Summation takes an arbitrary, possibly infinite, number of process summands. Here, Summation takes one of the following forms: i) The *deadlock agent*, **0**, capable of no actions whatever; **0** is defined to be $\sum_{i \in \emptyset} E_i$; ii) *Binary Summation*, which takes two summands only, $E_1 + E_2$ is $\sum_{i \in \{1,2\}} E_i$; iii) *Indexed Summation over sets*,[2] $\sum_{i \in \{e\} \cup I} E_i = E(e) + \sum_{i \in I} E_i$; and iv) *Infinite Summation over natural numbers*. The last two forms of Summation are introduced in order to specify parameterised behaviour and data communication.

Parallel Composition, $(|)$, is used to express concurrency: $E \mid F$ denotes an agent in which E and F may proceed independently, but may also interact with each other. The *Relabelling* combinator, $([])$, is used to rename port labels: $E[f]$ behaves as E, except that its actions are renamed as indicated by f. *Restriction*, $(\setminus)$, is used for internalising ports: $E \setminus L$ behaves like E, except that it cannot execute any action $\ell \in L \cup \overline{L}$.

2.2. Semantics

Processes are given a meaning via a labelled transition system[3] $(\mathcal{E}, \mathcal{A}ct, \{\stackrel{\alpha}{\rightarrow} : \alpha \in \mathcal{A}ct\})$, where $\stackrel{\alpha}{\rightarrow}$ is the smallest transition relation given by the following inference transition rules. Whenever $E \stackrel{\alpha}{\rightarrow} F$, F is said to be an α-

[1] These 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, $f(\tau) = \tau$.

[2] Sets and set-theoretic operations are implemented using lists and simulated by operations on lists, respectively.

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

derivative (or a *derivative*) of E.

$$\textbf{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \textbf{Com}_1\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$$

$$\textbf{Com}_2\ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad \textbf{Com}_3\ \frac{E \xrightarrow{\ell} E' \quad F \xrightarrow{\overline{\ell}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\textbf{Con}\ \frac{E \xrightarrow{\alpha} E'}{A \xrightarrow{\alpha} E'}\ A \stackrel{\text{def}}{=} E \qquad \textbf{Sum}_j\ \frac{E_j \xrightarrow{\alpha} E'}{\sum_{i \in I} E_i \xrightarrow{\alpha} E'}\ j \in I$$

$$\textbf{Rel}\ \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]} \qquad \textbf{Res}\ \frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L}\ \alpha, \overline{\alpha} \notin L$$

The interpretation of these rules is straightforward so we shall only describe **Com**$_3$. It stands for the possibility of two agents interacting with each other by the execution of complementary actions, that is, synchronisation. Notice that on synchronisation the agents in question change state simultaneously and Parallel Composition is preserved. Synchronisation actions are all indistinguishable from each other and so they are all denoted by the single action τ.

2.3. Process Equivalence and Bisimulation

Different interpretations of what is observable give rise to different equivalence relations. *Observation congruence* is the only relation that supplies two essential ingredients for process analysis: the abstraction of internal actions, and the property of being a congruence. Observation congruence resorts (as do all other behavioural equivalences), to the notion of *bisimulation*, which [13] defines as follows:

Definition 1 (Bisimulation) *A binary relation, $\mathcal{S}$, over processes is a* bisimulation *if $(P,Q) \in \mathcal{S}$ implies, for all $\alpha \in Act$,*

i) Whenever $P \xrightarrow{\alpha} P'$ and $\alpha \neq \tau$ then, for some Q', $Q \stackrel{\alpha}{\Rightarrow} Q'$ and $(P',Q') \in \mathcal{S}$, and similarly for P and Q interchanged;

ii) Whenever $P \xrightarrow{\tau} P'$ then either, for some Q', $Q \stackrel{\tau}{\Rightarrow} Q'$ and $(P',Q') \in \mathcal{S}$, or $(P',Q) \in \mathcal{S}$, and similarly for P and Q interchanged.

where $P \stackrel{\alpha}{\Rightarrow} P'$ means $P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* P'$. P' is said to be an α-descendant *(or a* descendant*) of P, whenever $P \stackrel{\alpha}{\Rightarrow} P'$.*

The union of all bisimulations yields *bisimilarity*: 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 known to be the only operator which breaks equivalence. The largest congruence relation included in $\approx$ is observation congruence [13]:

$P+Q = Q+P$	$P+(Q+R) = (P+Q)+R$
$P \mid Q = Q \mid P$	$(P \mid Q) \mid R = P \mid (Q \mid R)$
$P+P = P$	$\alpha.\tau.P = \alpha.P$
$P+\mathbf{0} = P$	$\tau.P + P = \tau.P$
$P \mid \mathbf{0} = P$	$\alpha.(P+\tau.Q)+\alpha.Q = \alpha.(P+\tau.Q)$
(2) Expansion	$\sum(P, nil) = \mathbf{0}$
	$\sum(P, h :: t)) = P(h)) + \sum(P,t)$

Table 1. The axioms $\mathcal{A}$

Definition 2 (Observation congruence) *P and Q are* equal *or* observation-congruent, *written $P \approx^c Q$, if for all $\alpha \in Act$, whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P' \approx Q'$, and similarly for P and Q interchanged.*

Most properties that relate $\approx$ and $\approx^c$ are captured in Hennessy's theorem [13, page 156]:

$$P \approx Q \text{ iff } P \approx^c Q, \text{ or } P \approx^c \tau.Q, \text{ or } \tau.P \approx^c Q \qquad (1)$$

Observation congruence is the relation selected to achieve the verification task. The proof system by which proofs are produced is shown below.

2.4. The Proof System

The proof system we use combines existing axioms for some classes of process behaviour: FSSs [13], value-passing systems [9], Basic Process Algebra [11] and Basic Parallel Processes [7], and adds to the combined axioms structural induction, or induction for short. A judgement is an expression of the form $\Gamma \vdash G$, where Γ is a set of formulae. Its intended meaning is: the formula G is provable in the proof system 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 to CCS terms, then, the intended meaning of G is: $T \approx^c U$ for every interpretation Γ that satisfies b.

Table 1 shows the set of basic axioms; it comprises axioms for **0**, Prefix, Binary Summation, Indexed Summation over natural numbers, Parallel Composition, and expansion. Expansion allows us to compute all process transitions. Algebraically, it is written as follows [13, page 69]: Let P be an abbreviation of the composite process $(P_1 \mid \cdots \mid P_n) \setminus L$, with $n \geq 1$, then

$$P = \sum \left\{ \alpha.P^i : P_i \xrightarrow{\alpha} P_i', 1 \leq i \leq n, \alpha, \overline{\alpha} \notin L \right\} \qquad (2)$$
$$+ \sum \left\{ \tau.P^{ij} : P_i \xrightarrow{\ell} P_i', P_j \xrightarrow{\overline{\ell}} P_j', 1 \leq i < j \leq n \right\}$$

where P^i (P^{ij}) abbreviates $(P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_n) \setminus L$ ($(P_1 \mid \cdots \mid P_i' \mid \cdots \mid P_j' \mid \cdots \mid P_n) \setminus L$). Note that, in general,

$$\text{Ac}\quad \frac{\Gamma \vdash P = S \vee P = \tau.S \vee \tau.P = S}{\Gamma \vdash \alpha.P = \alpha.S}$$

$$\sum_1\quad \frac{\Gamma, x{:}\text{nat} \vdash P_x = Q_x}{\Gamma \vdash \sum_{i\in\text{nat}} P_i = \sum_{i\in\text{nat}} Q_i}$$

$$\sum_2\quad \frac{\Gamma, x{:}\text{nat} \vdash x > n \to m > x \to P_x = Q_x}{\Gamma \vdash \sum_{i=n}^{m} P_i = \sum_{i=n}^{m} Q_i}$$

$$\sum_3\quad \frac{\Gamma, l{:}\text{nat list}, \ldots \vdash \mathit{member}(x,l) \to P_x = Q_x}{\Gamma, l{:}\text{nat list} \vdash \sum_{i\in l} P_i = \sum_{i\in l} Q_i}$$

$$+\quad \frac{\Gamma \vdash P_1 = S_1 \quad \cdots \quad \Gamma \vdash P_n = S_n}{\Gamma \vdash P_1 + \cdots + P_n = S_1 + \cdots + S_n}$$

$$\mid\quad \frac{\Gamma \vdash P_1 = S_1, \ldots, \Gamma \vdash P_n = S_n}{\Gamma \vdash P_1 \mid \cdots \mid P_n \;=\; S_1 \mid \cdots \mid S_n}$$

Table 2. Rules for manipulating process terms

each P_i might take the form $Q_i[f_i]$, for some relabelling function f_i.

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 Gentzen's sequent calculus (though they take their names from Prawitz's natural deduction). General purpose rules include induction, cut, rewrite-rule, α- and β- conversion. We only give the rules for manipulating process terms,[4] and omitting the usual rules for *reflexivity*, *symmetry*, *transitivity*, and *substitution* of equality. This completes our revision on the logic used throughout this paper. Now we consider proof planning.

3. Program Verification

We use CCS both as a programming language and as a specification language. Thus, the properties that we wish processes to satisfy need to be expressible as processes themselves; moreover, such properties should convey some (presumably useful) behaviour, which might not be recursive.[5] Behaviour is often succinctly captured using only Prefix and Binary Summation, as they provide no hint as to the internal working of a system. The sorts of processes we use as specifications take the form $S_1 + \cdots + S_n$, where each S_i, for all $i \in \{1, \ldots, n\}$, is either a prefixed process, or an indexed sum ($\sum$), possibly infinite, of prefixed processes. We call these kinds of expressions *sum forms*. For example, to specify that a buffer of size n, at any state, may input or output an element, provided that it is neither empty nor full, we write:

$$n > s(k) \to \mathit{Buf}_{\langle n,s(k)\rangle} \stackrel{\text{def}}{=} \mathit{in}.\mathit{Buf}_{\langle n,s(s(k))\rangle} + \overline{\mathit{out}}.\mathit{Buf}_{\langle n,k\rangle}$$

The systems under consideration, on the other hand, are arbitrary CCS terms. They reflect, at the required level of detail, all concurrency issues. What is more, they may contain recursive functions, which are used to capture the structure of one or more process subcomponents. We call these kinds of expressions *concurrent forms*. For example, we can implement a buffer of size n by linking n buffers of size 1:

$$\begin{array}{l} n = 0 \to C^{s(n)} = C \\ n \neq 0 \to C^{s(n)} = C \frown C^{(n)} \end{array} \quad \text{where} \quad \begin{array}{l} C \stackrel{\text{def}}{=} \mathit{in}.D \\ D \stackrel{\text{def}}{=} \overline{\mathit{out}}.C \end{array}$$

and where $P \frown Q \stackrel{\text{def}}{=} (P[c/\mathit{out}] \mid Q[c/\mathit{in}])\{c\}$. Then, for example, we could attempt to verify that if we link a cell (holding an element) to a generic buffer (holding k elements), then, by means of internal communication, the element held by the cell would propagate leaving it empty, returning a buffer holding $k+1$ elements:

$$\forall n, k{:}\text{nat}.n > k \to D \frown \mathit{Buf}_{\langle n,k\rangle} = \tau.(C \frown \mathit{Buf}_{\langle n,s(k)\rangle}) \qquad (3)$$

We will return to this example in §5.3. Having defined the verification problem, we attempt to give the reader a flavour as to the difficulties of automating proof search. In designing a proof strategy we are faced with two major challenges:

1. Termination. There are two sources for nontermination: i) Expansion can be applied infinitely many times, since it yields a process sum in which each summand contains a concurrent form; and ii) Rewriting with either rules of the form $T :\Rightarrow E[T]$, or

   rules that come from equations used both ways round is in general necessary (see for example [14]).

2. Controlled the use of the expansion law. Being an axiom scheme, the expansion law can be instantiated in many different ways, and hence can be applied to *any* proper subexpression. This increases the search branching rate.

These issues explain why radical measures are called for; yet they are further magnified in the presence of recursive functions. Induction is required to reason about recursive functions. Unfortunately, this makes automated verification

[4] For the sake of brevity, we omitted the free-variable (FV) side condition, $x \notin \mathit{FV}(\Gamma, P, Q)$, in Rules $\sum_i$, for $1 \leq i \leq 3$, of Table 2.

[5] Elsewhere, we show how to deal with the verification problem in the context of recursive processes by means of unique fixpoint induction.

Method	Description
`elementary`	tautology checker
`equal`	applies a hypothesis of type equality
`eval_def`	applies rewrite rules
`generalise`	generalise common subterms
`normalise`	normalises sequent formulae
`wave`	applies wave-rules
`casesplit`	splits a proof into cases
`fertilize`	appeals to induction hypothesis
`induction`	applies structural induction

Table 3. The standard method data-base

much harder, since it adds to the problem the difficulties of automating inductive theorem proving—issues here include deciding when to apply induction, how to select an appropriate induction scheme, and how to guide the search in inductive cases so that an appeal to the induction hypothesis can be made.

Fortunately, as discussed below, inductive proof planning is, at least partially, an answer to these problems.

4. Proof Planning

Proof planning [3] is a meta-level reasoning technique, especially developed as a search control engine 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. Moreover, proof planning provides a way to incorporate proof search strategies, tailored to a particular domain. Proof planning has been successfully applied in several different domains, including the use of formal methods for program synthesis [1] and hardware verification [6].

Methods are the building-blocks of proof planning. A *method* is a high-level description of a tactic, containing an input formula, preconditions, output formulae, and effects or postconditions. 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, describing under what circumstances it is appropriate to apply the method. The postconditions describe the effect of the associated tactic, without running the tactic proper. (Preconditions and postconditions are expressed in a meta-logic, where a subset of Prolog is the meta-language.) The ultimate result of method application is the output formulae, a list containing the new subgoals, if any (when the list is empty, the method is said to be *terminating*). Table 3 shows *Clam*'s standard method database. Methods are composed into proof plans using traditional planning techniques. The proof planner develops the plan by selecting a method applicable to the current goal. If no methods are applicable the planner backtracks. In the experiments reported here *Clam* used a depth-first planning strategy.

Proof planning involves the use of the heuristic known as *rippling* [4], which guides the search for a proof of inductive cases and supports the selection of appropriate induction schemata. Rippling guides the manipulation of the induction conclusion to enable the use of a hypothesis or *fertilization*. The key idea behind rippling lies in the observation that an initial induction conclusion is a copy of one of the hypotheses, except for extra terms, e.g., the successor function s, wrapping the induction variables. By marking such differences explicitly, rippling can attempt to place them at positions where they no longer preclude the conclusion and hypothesis from matching. Rippling is therefore an annotated term-rewriting system. It applies a special kind of rewrite rules, called *wave-rules*, which manipulate the differences between two terms (*wave-fronts*), while keeping their common structure (*skeleton*) intact. (Readers are referred to the MRG home page, `http://dream.dai.ed.ac.uk`, for further details.)

This completes our revision of proof planning. We are now ready to introduce `Equation`, the central contribution of this paper.

5. The `Equation` Method

Given the goal $P = S$, `Equation` aims at transforming P into S; accordingly, we call P and S the source and target, respectively. `Equation` approaches the problem in two steps: i) the verification that both source and target offer the same initial capabilities of interaction; and ii) the verification that, after action execution, P and S evolve into equivalent agents.

The source of inspiration of this two-step approach is situated in the definition of observation congruence (Definition 2). The first step, called `local`, involves P and Q, while the second one, called `global`, involves their derivatives. This observation has been used to define `Equation`, as illustrated below.

5.1. The Sub-method `local`

The aim of `local` is to transform the goal so that the source, S, and the target, P, satisfy the following condition:

(†) for all $\alpha \in \mathcal{A}ct$, P and S have the same number of process summands with prefix α.

Upon `local` success, P and S are proved to offer the same immediate possibilities for interaction: (†) implies that, for all $\alpha \in \mathcal{A}ct$, P has an α transition, written $P \xrightarrow{\alpha}$, if and only if S does, written $S \xrightarrow{\alpha}$.

To fulfill (†), first we transform P into a sum form, and next we *balance* the resulting equation, with respect to its *weight*:

Definition 3 (Weight) *Let P be a sum form, possibly empty. The* weight *of P, written weight(P), is the number of summands in P, each indexed sum, possibly infinite, counting as one process summand.*

Definition 4 (Balanced equation) *Let P and S be sum forms, possibly empty. Then $P = S$ is said to be a* balanced equation *if and only if weight(P)=weight(S); otherwise, it is said to be* unbalanced.

So, we make use of two properties of equality: expansion (2), and (a stronger version of) the *absorption lemma* [10] (see below). So, `local` is split into two methods, `Expansion` and `Absorption`.

`Expansion` enables the transformation of a concurrent form into a sum form. so it is applicable only when the target is a sum form, but the source is not. This avoids repeated use of `Expansion`. We omit the definition of `Expansion` and give our attention to `Absorption`.

5.1.1. `Absorption`

One condition for `Absorption` to be applicable is that the weight of the source must be greater than the weight of the target. On each application, `Absorption` removes one of the source's summands, hence decreasing its weight.

The absorption property asserts that whenever E can do an α action, accompanied by several τ actions, to reach F, then it can do the same without them [13, page 64]: If $E \stackrel{\alpha}{\Rightarrow} F$ then $\mathcal{A} \vdash E = E + \alpha.F$. Bearing this in mind, it should be apparent that `Absorption`'s applicability preconditions also involve the identification of two of source's subexpressions, as indicated below:

Definition 5 (Solvent, Absorbent, and Excess) *Take E to be a sum form such that $E \stackrel{\alpha}{\Rightarrow} F$, and consider the expression $E + \alpha.F$. We call E* solvent*; reciprocally, we call $\alpha.F$ the* excess*. Furthermore, the* absorbent *is a top-level summand of E capable of absorbing $\alpha.F$.*

Once the absorbent and the excess have been classified, `Observant` [14] is used. `Observant` is a decision procedure for transforming $E + \mu.F$ into E, provided that $E \stackrel{\mu}{\Rightarrow} F$. `Observant` is an annotated term rewriting system; the annotations are used to capture meta-level information that carefully directs proof search, while giving an intuitive account about why and how rewriting is expected to work. `Observant` is terminating and complete. Figure 1 depicts the definition of `Absorption`.

By using `Absorption`, we know precisely when we can remove a summand; by using `Observant`, we are certain that the summand will be removed. Thus we reduce proof search while avoiding non-termination.

Input: $H \vdash C \rightarrow P = S$
Preconditions:
weight_greater(P, S), *absorbent*(P_i, Pos_i, P, S),
irrelevant_smnd$(\mu.P_j, Pos_j, P, P_i)$, $P_i \stackrel{\mu}{\Rightarrow} P_j$
Effects: *absorbs*$(P_i, \mu.P_j, P, P')$
Output: $H \vdash C \rightarrow P' = S$
Meanings of the meta-logic (Prolog) predicates:

- *weight_greater*(P, S): that the weight of P is greater than the weight of S.
- *absorbent*$(?P_i, ?Pos, P, S)$: that $?P_i$, a subprocess of P that appears at position $?Pos$, is an absorbent w.r.t. S.
- *irrelevant_smnd*$(?Q, ?Pos, P, R)$: $?Q$, a subprocess of P at position $?Pos$, is excess of R.
- *absorbs*$(?Q, ?R, P, -P')$ P' is as P except that subprocess $?R$ has been absorbed by subprocess $?Q$, by means of `Observant`.

Figure 1. The `Absorption` Method

5.2. The Sub-method `global`

`global` is used only to complete the search task initiated by `local`. It is concerned with the proper association of each summand of P with only one summand of S. An association is said to be proper, if it leads to a proof of the associated expressions. This, of course, cannot be decided without completing the proof.

`global` distinguishes two cases, according to the weight of either side of the equation: unary weight equations, and multiple weight equations.

5.2.1. `Action`

`Action` is concerned with goals of the form $\vdash \alpha.P = \alpha.S$. To prove one such goal, we have to prove $P \approx S$; but, by Hennessy's theorem (1), to prove $P \approx S$, we may prove that either $P = Q$, $P = \tau.Q$, or $\tau.P = Q$. To resolve which of these OR-branches we should pursue first, we use a look-ahead strategy, based on process behaviour, namely: Let $P_{trans} = \{\alpha_i : P \stackrel{\alpha_i}{\rightarrow}\}$, and $S_{trans} = \{\alpha_j : S \stackrel{\alpha_j}{\rightarrow}\}$, then i) If $P_{trans} = S_{trans}$ and if the number of transitions of P is greater than or equal to those of S, then select $P = S$; and ii) If the condition above does not hold and if $P_{trans} \backslash \{\tau\} \subset S_{trans}$, then select $P = \tau.S$.

5.2.2. `Goalsplit`

`Goalsplit` is concerned with the case dual to that of `Action`. It divides a goal into a conjunction of subgoals, each of the form $\alpha.P = \alpha.S$, hence enabling `Action`. When `global` fails, the verification planner will back-

track. Backtracking occurs rarely in the context of *determinate agents*, where behaviour is always predictable.

The `Equation` methods should be added to the standard method data-base (see Table 3), and placed before `wave`. The rational being that `Equation` is seen as an extension to symbolic evaluation, `eval_def`, only that it allows us to reason about CCS agents. Furthermore, `eval_def` must be extended with the rewrite rule set extracted from the axioms $\mathcal{A}$ (Table 1), in the direction left to right.

5.3. A Worked Example

In this section, we show one example verification problem uniting the use of the `Equation` methods, as well as their interaction with each other and other methods of inductive proof planning. Consider again (3). We assume the following rules:

$$X = s(Y) \to Buf_{\langle X, \boxed{s(\underline{Y})}^{\uparrow} \rangle} :\Rightarrow \boxed{\overline{out}.\underline{Buf_{\langle X,Y\rangle}}}^{\uparrow} \quad (4)$$

$$X > s(Y) \to Buf_{\langle X, \boxed{s(\underline{Y})}^{\uparrow} \rangle} :\Rightarrow \boxed{in.Buf_{\langle X, s(s(Y))\rangle} + \overline{out}.\underline{Buf_{\langle X,Y\rangle}}}^{\uparrow} \quad (5)$$

$$0 > X :\Rightarrow \text{false} \quad (6)$$

$$X > 0 :\Rightarrow X \neq 0 \quad (7)$$

$$\boxed{s(\underline{X})}^{\uparrow} > \boxed{s(\underline{Y})}^{\uparrow} :\Rightarrow X > Y \quad (8)$$

To attempt to prove (3), *Clam* suggests a one step induction on k, since n does not occur at recursive positions in the definition of *Buf*. Let us look into the proof steps of each induction subgoal.

5.3.1. Base Case ($k = 0$)

The base case initially takes the form:[6]

$$\cdots \vdash \forall n{:}\text{nat}. n > 0 \to D^\frown Buf_{\langle n,0\rangle} = \tau.(C^\frown Buf_{\langle n,s(0)\rangle})$$

Symbolic evaluation, together with normalisation, simplifies this equation to

$$n{:}\text{nat}, n \neq 0 \vdash D^\frown Buf_{\langle n,0\rangle} = \tau.(C^\frown Buf_{\langle n,s(0)\rangle})$$

Now it is `local`'s turn: while the target is a sum form, the source is a concurrent form. `Expansion` is therefore in order, leaving a trivially provable subgoal.

$$\cdots \vdash \tau.(C^\frown Buf_{\langle n,s(0)\rangle}) = \tau.(C^\frown Buf_{\langle n,s(0)\rangle})$$

`elementary` closes this proof branch tree; so, the proof search engine gives attention to the step case.

[6] In what follows, ellipsis are used to denote expressions that remain unchanged.

5.3.2. Step Case

The step case has the schematic form:

$$\ldots, n > v \to D^\frown Buf_{\langle n,v\rangle} = \tau.(C^\frown Buf_{\langle n,s(v)\rangle}) \quad (9)$$
$$\vdash$$
$$\forall n{:}\text{nat}. n > \boxed{s(\underline{v})}^{\uparrow} \to D^\frown Buf_{\langle n, \boxed{s(\underline{v})}^{\uparrow} \rangle} = \tau.(C^\frown Buf_{\langle n, s(\boxed{s(\underline{v})}^{\uparrow})\rangle}) \quad (10)$$

We shall omit that part of the story relevant to the inductive proof plan, and concentrate on what is original about our work. *Clam* ends up with a number of subgoals, which arise according to the case analysis suggested by the side conditions of the rewrite rules. We consider the following case only, as the others are just as easy and of the same pattern.

Case $n = s(s(v))$ Consider then the goal

$$n = s(s(v)), n > s(v) \vdash$$
$$D^\frown Buf_{\langle n, \boxed{s(\underline{v})}^{\uparrow} \rangle} = \tau.(C^\frown \boxed{\overline{out}.\underline{Buf_{\langle n,s(v)\rangle}}}^{\uparrow})$$

At this point, `Expansion` is applicable, leaving a goal of the form:[7]

$$\cdots \vdash \boxed{\tau.(C^\frown Buf_{\langle n,s(s(v))\rangle}) + \overline{out}.(\underline{D^\frown Buf_{\langle n,v\rangle}})}^{\uparrow} = \cdots$$

Now, `fertilisation` is applicable, modulo the antecedent of the implication formula; so, it asks for a proof of the proviso $\cdots \vdash n > s(v) \to n > v$. Proof planning deals with the new proof obligation. It also proof plans the main result as follows. By using (9), as a rewrite rule from left to right, it gets the following induction conclusion:

$$\cdots \vdash \tau.(C^\frown Buf_{\langle n,s(s(v))\rangle}) + \overline{out}.\tau.(C^\frown Buf_{\langle n,s(v)\rangle}) = \tau.(C^\frown Buf_{\langle n,s(v)\rangle})$$

Simplification, using `eval_def`, discards the internal action, leaving a formula where `Absorption`'s applicability preconditions hold. $\tau.(C^\frown Buf_{\langle n,s(s(v))\rangle})$ is regarded as absorbent, while $C^\frown Buf_{\langle n,s(v)\rangle}$ is regarded as irrelevant summand. Furthermore, the absorbability test succeeds, for

$$\cdots \vdash \tau.(C^\frown Buf_{\langle n,s(s(v))\rangle}) \stackrel{\overline{out}}{\Rightarrow} C^\frown Buf_{\langle n,s(v)\rangle}$$

`Absorption` will therefore get rid of the irrelevant summand, leaving

$$\cdots \to \tau.(C^\frown Buf_{\langle n,s(s(v))\rangle}) = \tau.(C^\frown \overline{out}.Buf_{\langle n,s(v)\rangle})$$

[7] It is worth noting that `Expansion` respects the rippling. That is, `Expansion` is applied to an annotated term if and only if the expanded term can be annotated so that the skeleton is preserved, and the rippling termination measure is decreased.

Further simplifications are in order; this time `Action` strips off the prefixes, while a cancellation operation eliminates the linking operator, yielding

$$\cdots \vdash \cdots \rightarrow Buf_{\langle n, s(s(v))\rangle} = \overline{out}.Buf_{\langle n, s(v)\rangle}$$

which `Expansion` further transforms, resulting in a readily provable equation. □

6. Results and Comparison to Related Work

There are different grounds upon which the performance of reasoning systems can be judged; according to [3], the most important evaluation criteria are *generality*, and *expectancy*. Generality means usefulness in a large number of examples amongst the intended proof family. Expectancy pertains to the ability of predicting the successful outcome of a proof plan: there should be a story about why the technique works. More subjective metrics can also be considered. Two examples are the succinctness of the description of a proof plan, and the quality of proofs yielded by a plan, e.g., naturalness, readability, length, etc. Naturally, efficiency can be an evaluation criterion too. In the assessment of the proposed verification plan, emphasis will be given to expectancy and generality, though figures concerning planning CPU time will also be enumerated.

§5 implicitly provided a discussion of expectancy: it includes an account as to why `Equation` should work. Moreover, it has initiated already our attempt to evidence the generality of the verification plan by showing its behaviour on a a worked example. As far as generality is concerned, this section takes a step further: it provides empirical evidence in the form of results obtained from testing the plan on a set of examples. Naturally, for this to be meaningful, the test set must be representative. To guarantee representativeness, we kept the examples as dissimilar and unbiased as possible, gathering them from different sources. The system was *developed* with a set of examples entirely disjoint from the *testing* set presented here.

In the next section we summarise some of both the experimental work and the results obtained throughout our investigations.

6.1. Some Illustrative Example Conjectures

Table 4 gives some of the example systems with which we tested *Clam*. For each item, we provide the definition of the system, together with the definition of its subcomponents.

Table 5 shows some of the conjectures used in the test set. Considering the whole test set, which includes more than 30 conjectures, the success rate of *Clam* was 86%, with an average total elapsed planning time of 226 seconds, and standard deviation of 192. 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, including *Clam* and the methods for CCS verification, is available upon request from the first author.

6.2. Comparison

Clam found a proof plan on all the test cases shown in table 5, non of which can even be input to any other automated tool. Most of the verification conjectures included in the test set remain outside the scope of current automatic verification tools. For example, the Concurrency Workbench [8] is restricted to a very simple language. Value-passing, even if confined to finite data domains, is not directly expressible. By contrast, we model parameterised behaviour and consider infinite Summation.

[11, 12] have shown that $\sim$ over normed-BPA and normed-BPP is decidable in polynomial time; their algorithms are special purpose; they cannot be used to analyse FSS. This is in contrast with our method, which attempts to provide a unified verification framework. The algorithms of [11, 12] all deal with the equivalence problem, provided the input processes are of the same and of the intended class; they cannot handle the verification problem, since parameterised specifications are not BPA, or BPP terms. Conversely, while the verification plan deals with the verification problem, it cannot handle the equivalence of arbitrary BPA, or BPP terms. Hence, in this respect, our work and the work of [11, 12] are complementary.

The symbolic bisimulation method is an extension to the bisimulation method. It adds a means for modelling and analysing value-passing agents. [9] shows that symbolic bisimulation equivalence is decidable; naturally, this result is relative to the decidability of proving that the current interpretation is satisfiable. In this respect, the symbolic bisimulation method is bound to rely upon external theorem provers, through which it could discharge these extra problems. In contrast, providing a unified verification framework has been one of the main emphases of our work. Moreover, while we are able to deal with infinite-state and parameterised systems, nor are Hennessy and Lin; nor have they considered the use of primitive recursive functions to represent the structure of a system at any state during execution. However, we include no truly value-passing mechanism, but a modest mechanism, centred around the use of Summation, and a partial translation of full CCS into basic CCS that simulates value-passing.

#	System	Agents
1	$C^0 = B$ $C^{s(n)} = C \frown C^{(n)}$	$B \stackrel{\text{def}}{=} inc.(C \frown B) + zero.B$ $C \stackrel{\text{def}}{=} inc.(C \frown C) + dec.D$ $D \stackrel{\text{def}}{=} \overline{d}.C + \overline{z}.B$
	$P \frown Q \stackrel{\text{def}}{=} (P[i'/i, z'/z, d'/d] \mid Q[i/inc, z'/zero, d'/dec]) \setminus \{i', \overline{i'}, z', \overline{z'}, d', \overline{d'}\}$	
2	$B^{(0)} = B$ $B^{s(2\times n)} = D_0 \frown B^{(n)}$ $B^{s(s(2\times n))} = D_1 \frown B^{(n)}$ $B^{s(n)} = V^{(n)} \frown B^{half(n)}$	$B \stackrel{\text{def}}{=} inc.(D_0 \frown B) + zero.B$ $D_1 \stackrel{\text{def}}{=} inc.Cy + dec.D_0$ $D_0 \stackrel{\text{def}}{=} inc.D_1 + dec.Bw$ $Cy \stackrel{\text{def}}{=} \overline{i}.D_0$ $Bw \stackrel{\text{def}}{=} \overline{d}.D_1 + \overline{z}.B$
	$P \frown Q \stackrel{\text{def}}{=} (P[i'/i, z'/z, d'/d] \mid Q[i/inc, z'/zero, d'/dec]) \setminus \{i', \overline{i'}, z', \overline{z'}, d', \overline{d'}\}$	
3	$S^{(\text{nil})} = B$ $S^{(h::t)} = C_{(h)} \frown S^{(t)}$	$B \stackrel{\text{def}}{=} \sum_{n \in \text{nat}} (push_n.(C_n \frown B)) + empty.B$ $C_{(n)} \stackrel{\text{def}}{=} \sum_{x \in \text{nat}} (push_x.(C_x \frown C_n)) + \overline{pop_n}.D$ $D \stackrel{\text{def}}{=} \sum_{x \in \text{nat}} (o_x.C_x) + \overline{e}.B$
	$P \frown Q \stackrel{\text{def}}{=} (P[i'/i, o'/o, e'/e] \mid Q[i'/push, o'/pop, e'/empty]) \setminus \{i', o'\}$	
4	$C^{s(0)} \stackrel{\text{def}}{=} C$ $C^{s(s(n))} \stackrel{\text{def}}{=} C \frown C^{s(n)}$	$C \stackrel{\text{def}}{=} in.D$ $D \stackrel{\text{def}}{=} \overline{out}.C$
	$P \frown Q \stackrel{\text{def}}{=} (P[c/out] \mid Q[c/in]) \setminus \{c\}$	
5	$\prod_{=0}^{n-k} S \mid \prod_{j=0}^{k} V$	$S \stackrel{\text{def}}{=} get.V \qquad V \stackrel{\text{def}}{=} put.S$
6	$\prod_{i \in l} D_{\langle i,n \rangle}$	$A_{\langle i,n \rangle} \stackrel{\text{def}}{=} a_i.C_{\langle i,n \rangle}$ $C_{\langle i,n \rangle} \stackrel{\text{def}}{=} c_i.E_{\langle i,n \rangle}$ $E_{\langle i,n \rangle} \stackrel{\text{def}}{=} b_i.D_{\langle i,n \rangle} + \overline{c_{i-1 \bmod n}}.B_{\langle i,n \rangle}$ $B_{\langle i,n \rangle} \stackrel{\text{def}}{=} b_i.A_{\langle i,n \rangle}$ $D_{\langle i,n \rangle} \stackrel{\text{def}}{=} \overline{c_{i-1 mod n}}.A_{\langle i,n \rangle}$

Table 4. Process definitions

7. Conclusions

We have investigated the use of proof planning for problems of automatic verification, in the context of CCS. We have conducted the verification task using equational reasoning, and centred on value-passing, infinite-state, parameterised systems (VIPSs).

To reason about such systems, we have advocated the use of induction in order to exploit the structure and/or the behaviour of a system during its verification. To automate this reasoning, we have used proof plans for induction—built within *Clam*, and extended with special CCS proof plans. We have validated our working hypotheses, which we characterise by the following list of contributions:

1. We have captured and formalised general knowledge of a family of CCS proofs;
2. We have shown that the heuristics developed in this paper, all of which use meta-level information, provide a means for guiding the search for CCS verifications, significantly reducing the search space;
3. We have shown that by including the heuristics mentioned above, inductive proof planning can encompass a uniform search strategy to produce, automatically, CCS verification plans under the equational approach. Moreover, the resulting proof plan provides a uniform method to analyse CCS systems, regardless of the size of their state space; it can plan a verification of some problems involving finite-state systems and, more importantly, value-passing, infinite-state, parameterised systems.

#	Conjecture
1	$n = 0 \rightarrow C^n = inc.C^{s(n)} + zero.C^n$ $n \neq 0 \rightarrow C^n = inc.C^{s(n)} + dec.C^{p(n)}$
2	$n = 0 \rightarrow B^n = inc.B^{s(n)} + zero.B^n$ $n \neq 0 \rightarrow B^n = inc.B^{s(n)} + dec.B^{p(n)}$
3	$l = \text{nil} \rightarrow S^{(l)} = \sum_{n \in \text{nat}} (push_n.S^{(n::l)}) + empty.S^{(l)}$ $l \neq \text{nil} \rightarrow S^{(l)} = \sum_{n \in \text{nat}} (push_n.S^{(n::l)}) + \overline{pop_{hd(l)}}.S^{tl(l)}$
4	$n > k \rightarrow D^\frown Buf_{\langle n,k \rangle} = \tau.(C^\frown Buf_{\langle n,s(k) \rangle})$
5	$n \neq 0 \wedge k \neq 0 \rightarrow \prod_{i=0}^{n} S \prod_{j=0}^{k} =$ $get.(\prod_{i=0}^{p(n)} S \,\big\vert\, V \,\big\vert \prod_{j=0}^{k}) + put.(\prod_{i=0}^{n} S \,\big\vert\, S \,\big\vert \prod_{j=0}^{p(k)})$
6	$unique(l) \rightarrow \prod_{i \in l} D_{\langle i,n \rangle} =$ $\sum_{i \in l} \overline{c_{i-1 \bmod n}}.(A_{\langle i,n \rangle} \,\big\vert \prod_{j \in del(i,l)} D_{\langle j,n \rangle})$ $unique(l) \rightarrow \prod_{i \in l} E_{\langle i,n \rangle} =$ $\sum_{i \in l} b_i.(D_{\langle i,n \rangle} \,\big\vert \prod_{j \in del(i,l)} E_{\langle j,n \rangle})$ $+ \sum_{i \in l} \overline{c_{(i-1) \bmod n}}.(B_{\langle i,n \rangle} \,\big\vert \prod_{j \in del(i,l)} E_{\langle j,n \rangle})$ where $member(h,t) \rightarrow unique(h :: t) = \bot$ $\neg member(h,t) \rightarrow unique(h :: t) = unique(t)$

Table 5. Example verification conjectures

References

[1] A. Armando, A. Smaill, and I. Green. Automatic synthesis of recursive programs: The proof-planning paradigm. In *12th IEEE International Automated Software Engineering Conference*, pages 2–9, Lake Tahoe, Nevada, USA, 1997.

[2] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[3] A. Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[4] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[5] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[6] F. Cantu, A. Bundy, A. Smaill, and D. Basin. Experiments in automating hardware verification using inductive proof planning. In M. Srivas and A. Camilleri, editors, *Proceedings of the Formal Methods for Computer-Aided Design Conference*, number 1166 in Lecture Notes in Computer Science, pages 94–108. Springer-Verlag, 1996.

[7] S. Christensen, Y. Hirschfeld, and F. Moller. Decomposability, decidability and axiomatisation for bisimulation equivalence on basic parallel processes. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *LICS'93*, pages 386–396, New York, 1993. IEEE Computer Society Press.

[8] R. Cleaveland, P. J., and B. Steffen. 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, 1989. Lecture Notes in Computer Science, v.407. Also available from Edinburgh, as ECS-LFCS-89-83.

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

[10] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, 1985.

[11] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Journal of Theoretical Computer Science*, 158:143–159, 1996.

[12] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimulation equivalence of normed basic parallel processes. *Mathematical Structures is Computer Science*, To appear, 1996.

[13] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

[14] R. Monroy, A. Bundy, and I. Green. Annotated term rewriting for deciding observation congruence. In H. Prade, editor, *13th European Conference on Artificial Intelligence, ECAI'98*, pages 393–397, Brighton, England, 1998. Wiley & Sons. To appear.