Proof Planning the Verification of CCS Programs *

Raul Monroy, Alan Bundy, Andrew Ireland, & Jane Hesketh

Department of Artificial Intelligence
The University of Edinburgh
80 South Bridge, EH1 1HN
Scotland, U.K
(+44 131) 650 2724
raulm, bundy, air, & jane@aisb.ed.ac.uk

Abstract. The verification of CCS programs has often been characterised as an expensive, time-consuming, and error-prone task, where computer assistance is thought to be essential. Yet, existing theorem proving based frameworks, for the verification of programs using CCS, are no much use because their poor level of automation.
In this paper, we propose the use of proof plans as the chief mechanism for the automation of CCS program verification. We present a collection of proof plan methods that guide a proof plan formation for the verification of deterministic, and divergence-free CCS programs. The meta-level reasoning embedded in these methods takes full advantage of the operational interpretation of processes while outputting plan steps to the appropriate object-level representation.
Keywords: Proof planning, process algebras, CCS, automatic program verification.

1 Introduction

The specification and verification of communicating systems has often been characterised as an expensive, time-consuming, and error-prone task, where computer assistance is thought to be essential. There exist several paradigms for the modelling and analysis of such systems. One paradigm that has become very popular for the last decade is that of process algebras. Under this approach, components of communicating systems (e.g. processes, links) are modelled as mathematical objects susceptible to the application of basic operations that reflect an operational idea (e.g. communication, mobility, distribution). CCS (a Calculus for Communicating Systems) [20] is, perhaps, one of the most popular approaches for the algebra of processes.

Several tools for reasoning about CCS programs exist. They could be split into two classes [14]: automata theory based e.g. [9, 11], and theorem proving

* The research reported here was supported by CONACyT studentship 64745 to the first author, and EPSRC grants GR/J/80702 and GR/J/58619 to the second and third author, respectively.

based e.g. [17, 22]. Automata theory based tools are much more popular due to their level of automation, but are not able to deal with systems that contain infinite states or that comprise a finite but unbound number of components. Also, most of the existing tools of either kind make the process of correcting non-theorems difficult.

We believe that the proof planning technique [3] can effectively guide the search for a proof that a CCS program meets its specification, and hence bridge the gap that stops theorem proving based tools being used in this domain of application. Moreover, meta-level reasoning used in plan formation, together with an abduction mechanism, can be used to analyse the information derived from an (unsuccessful) attempt to find a proof of a (faulty) conjecture, and hence propose candidate patches that could turn it into a theorem, as in [21].

In this paper, we take a few steps towards this aim. In particular, we propose a collection of proof plan methods that guide the process of proof plan formation for the verification of deterministic, and divergence-free CCS programs.

The rest of this paper is organised as follows: Sect. 2 gives a brief description of the important notion of proof planning, and recalls the operational semantics of the CCS combinators. Sect. 3, the core of the paper, introduces the plan methods used to guide the construction of a proof plan for the verification of CCS programs. Sect. 4, outlines results, limitations, and further work, while Sect. 5 compares our approach to related work.

2 Preliminaries

2.1 Proof Plans

It has been gradually recognised that when it comes to automating the verification of CCS programs, one can often do better with non-equational proofs, i.e. by showing that the processes in question are in a bisimulation. We attribute this phenomenon to the following fact:

- In this context, program verification amounts to proving both the program and its intended specification equivalent. Processes equivalence is defined in terms of *observable* behaviour, where observation is the communicating process that takes place between the system in question and an external observer (the environment). Equational reasoning tends to obscure the operational idea above. It is therefore not surprising that it is easy to lose grip of the overall pattern that proofs should follow.

The chief mechanism we propose for bridging this gap consists of a new application of the proof planning technique [3].

Proof plans are a means of expressing the commonality between the members of the same family of proofs. This technique guides the search for a proof in the context of tactical style reasoning, using tactic specifications called methods. Tactics are programs which perform a theorem proving task, e.g. rewriting a subterm or applying an inference rule. Methods express the preconditions under which tactics are applicable and the effects of their application.

As the meta-level reasoning embedded in a plan method is independent of any object-level theory representation, it is possible to take advantage of the operational characterisation of processes to guide the search for equational proofs of a verification problem.

The proof planning technique has been implemented in a system called CLAM [6], and successfully applied to the domain of inductive proofs.

2.2 Operational Semantics of Pure CCS

Throughout this paper we assume the reader is familiar with CCS, or some other process algebra. In this section, however, we recall the operational semantics of the operators of the language.

Terms of CCS represent processes that perform actions. The set of actions, $\mathcal{A}ct$, contains names (e.g. in), co-names (e.g. $\overline{in}$), and the special action τ that results from any communication amongst processes. Processes are expressions with the following syntax:

$$E \ ::= \ \mathbf{0} \mid \alpha.E \mid E + E \mid E \mathbin{|} E \mid \ E \setminus L \mid E[F] \mid$$

where $\mathbf{0}$ is the deadlock process, and where L ranges over sets of actions, while F ranges over functions from actions to actions. We usually let $\sum_{i=1}^{n} \alpha_i.P_i$ stand for an abbreviation of $\alpha_1.P_1 + \ldots + \alpha_n.P_n$. Process constants $(A, B, \ldots)$ are also process terms. They have definitions of the form $A \stackrel{\text{def}}{=} E$. The set of all process expressions and process constants is referred to as simply *processes*; it is denoted by $\mathcal{P}$ and ranged over by meta-variables $P, Q, \ldots$.

Processes are given meaning by means of the labelled transition system $(\mathcal{P}, \mathcal{A}ct, \{\stackrel{\alpha}{\rightarrow}: \alpha \in \mathcal{A}ct\})$, where $\stackrel{\alpha}{\rightarrow}$ is a transition relation defined to be the least relation satisfying the rules below:

$$\text{Act}\ \frac{}{\alpha.A \stackrel{\alpha}{\rightarrow} A} \qquad \text{Sum}_1\ \frac{E_1 \stackrel{\alpha}{\rightarrow} E_1'}{E_1 + E_2 \stackrel{\alpha}{\rightarrow} E_1'} \qquad \text{Sum}_2\ \frac{E_2 \stackrel{\alpha}{\rightarrow} E_2'}{E_1 + E_2 \stackrel{\alpha}{\rightarrow} E_2'}$$

$$\text{Com}_1;\ \frac{E \stackrel{\alpha}{\rightarrow} E'}{E \mathbin{|} F \stackrel{\alpha}{\rightarrow} E' \mathbin{|} F} \qquad \text{Com}_2\ \frac{F \stackrel{\alpha}{\rightarrow} F'}{E \mathbin{|} F \stackrel{\alpha}{\rightarrow} E \mathbin{|} F'} \qquad \text{Com}_3\ \frac{E \stackrel{\alpha}{\rightarrow} E' \quad E \stackrel{\overline{\alpha}}{\rightarrow} E'}{E \mathbin{|} F \stackrel{\tau}{\rightarrow} E' \mathbin{|} F'}$$

$$\text{Res}\ \frac{E \stackrel{\alpha}{\rightarrow} E'}{E \setminus L \stackrel{\alpha}{\rightarrow} E' \setminus L} (\alpha, \overline{\alpha} \notin L) \qquad \text{Rel}\ \frac{E \stackrel{\alpha}{\rightarrow} E'}{E[F] \stackrel{f(\alpha)}{\rightarrow} E'[F]} \qquad \text{Con}\ \frac{E \stackrel{\alpha}{\rightarrow} E'}{A \stackrel{\alpha}{\rightarrow} P'}\ A \stackrel{\text{def}}{=} E$$

3 The Planner Methods

In this section, we shall introduce the methods used for planning the verification of CCS programs. We will see that the meta-level reasoning embedded in such methods is strongly based on the operational semantics of processes. We shall illustrate method application through a couple of simple theorems, namely:

$$CC = B_2^0 \tag{1}$$
$$CCC = B_3^0 \tag{2}$$

Where B_2^0 and B_3^0 are specifications for a 2-size buffer and a 3-size buffer, respectively:

$$\begin{array}{ll}
B_2^0 \stackrel{\text{def}}{=} in.B_2^1 & B_3^0 \stackrel{\text{def}}{=} in.B_3^1 \\
B_2^1 \stackrel{\text{def}}{=} in.B_2^2 + \overline{out}.B_2^0 & B_3^1 \stackrel{\text{def}}{=} in.B_3^2 + \overline{out}.B_3^0 \\
B_2^2 \stackrel{\text{def}}{=} \overline{out}.B_2^1 & B_3^2 \stackrel{\text{def}}{=} in.B_3^3 + \overline{out}.B_3^1 \\
 & B_3^3 \stackrel{\text{def}}{=} \overline{out}.B_3^2
\end{array}$$

while $CC = C^\frown C$ and $CCC = C^\frown(C^\frown C)$ are the proposed programs. We consider $C \stackrel{\text{def}}{=} in.C'$, $C' \stackrel{\text{def}}{=} \overline{out}.C$, and $P^\frown Q \stackrel{\text{def}}{=} (P[c/out] \mid Q[c/in]) \setminus \{c\}$, for arbitrary P and Q. Throughout the paper, when understood from the context, we shall use expressions like $CC'C'$ as abbreviations for $C^\frown(C'^\frown C')$

As even for simple theorems, proofs tend to be rather large, we shall only show the bits of the proof which are interesting, with the hope that the reader will grasp the pattern they follow.

3.1 The UFI Method

In equational axiomatisations of CCS, Uniqueness of Solutions of Equations (also referred to as Unique Fixpoint Induction (UFI) in the literature [12]) is the key property used for establishing processes equivalence.

$$\frac{\vdash\ P = E\{P/S\} \qquad\qquad \vdash S = E}{\vdash\ P = S}\quad \mathit{guarded}(S,E),\ \mathit{sequential}(S,E)$$

where $\mathit{guarded}(S,E)$ means that every occurrence of S in E is within some subexpression $\alpha.F$ of E, for $\alpha \neq \tau$, while $\mathit{sequential}(S,E)$ means that S only occurs within Prefix (.) or Summation (+) combinators in E. Informally, this rule says that two processes are equivalent if they satisfy the same set of *recursive* equations.

Both the PAM system [17], and the HOL system [22], have a tactic which applies UFI forwards, using an *AC matching* algorithm. Thus proving two process equivalent amounts to AC matching the sets of equations that such processes have been proved to satisfy. Certainly, a bunch of, most of the time tedious, equation manipulations must be performed for the UFI tactic to succeed. These systems are highly interactive. The user is responsible for constructing and manipulating a program's set of equations (in the context of program verification, the set of equations satisfied by the specification usually is given in advance).

The approach we have adopted differs from the above in that we reason backwards, from the goal to the axioms. Thus, we do not calculate the involved sets of equations prior to an application of UFI. Instead we use either side of the equality to be proved as the target to be matched by the other side. However, from now on, we shall assume that the specification (the target) is placed on the right-hand side (RHS) of the equation.

The version of UFI currently in use within our system is shown below:

$$\frac{P = S \;\vdash\; P = E}{\vdash\; P = S} \quad S \stackrel{\text{def}}{=} E,\; \mathit{guarded}(S, E),\; \mathit{sequential}(S, E) \tag{3}$$

Roughly speaking, to prove that $P = S$, considering that $S \stackrel{\text{def}}{=} E$, we prove that $P = E$ (the inductive conclusion) assuming that $P = S$ holds (the inductive hypothesis). Fertilisation [3], is the process of using an induction hypothesis to complete its proof. This process is embedded in CLAM's `fertilise` method, which we have borrowed for this new application of the planner.

One obvious advantage of using (3) is that we do not need to know the behaviour of P and S in advance, which fits well in a backwards reasoning proof framework. Thus, the definition of the UFI method depicted in Fig. 1 follows. As

Input:

$H \vdash P = S$

Preconditions:

Let E be the definition of S, then

guarded(S, E)

sequential(S, E)

Output:

$H \cup \{P = S\} \;\vdash\; P \;=\; E$

Meanings of the meta-logic terms:

- guarded(S, E) succeeds whenever every occurrence of S appears guarded in E
- sequential(S, E) succeeds whenever every occurrence of S appears sequential in E.

Fig. 1. The UFI Method

a simple example method application, consider (1) which this method transforms into

$$CC = B_2^0 \vdash \;CC = in.B_2^1 \tag{4}$$

The UFI method is clearly applicable a potentially infinite number of times. This is why it appears last in the hierarchy of methods (just the same as the CLAM `induction` method). A discussion on methods hierarchy is postponed until Sec. 3.5.

3.2 The Expansion Method

The expansion law aims to reduce parallelism to nondeterminism; that is, it embodies the so-called interleaving semantics of CCS. It calculates the behaviour of a composite process (i.e. a process which is the parallel composition of some other – possibly composite, process) in terms of the (possibly empty) sum of its immediate derivatives, where the parallel composition operator no longer dominates the term structure.

Our approach for the application of the expansion theorem follows the game theory characterisation of $=$ and $\approx$ given in [26]. In our case, however, specifications play a leading role in that programs select their move when specifications are expressions in the syntactic subclass $E^+ ::= \mathbf{0} \mid \alpha.E^+ \mid E^+ + E^+$. A turn consists of an application of the expansion law (recall that definition unfolding is a special case of expansion, c.f. the **Con** transition rule). The definition of the expansion method is depicted in Fig. 2. Clearly, an application of this method to (4) results in

$$\ldots \vdash \; in.(C'C) = in.B_2^1 \tag{5}$$

Input:

$\vdash P = S$

Preconditions:

sumform(S)

$\neg$ sumform(P)

Output:

expands_into$(P,\ ExP)$

$\vdash\ ExP\ =\ S$

Meanings of the meta-logic terms:

- expands_into$(P,\ ExP)$; ExP is the expansion of process P.
- sumform(P) succeeds if P is a sum of prefixed process; that is, if P is of the form: $\sum_{i=1}^{n} \alpha_i.P_i$. Note that, in particular, $\mathbf{0}$, the deadlock process, is a sumform by taking $n = 0$.

Fig. 2. The Expansion Method

We now draw attention to one of the pivotal plan methods for program verification within our approach.

3.3 The Goal Splitting Method

As its name suggests, the Goal Splitting method aims to divide a goal into a number of simpler subgoals. It has both the definition of observational congruence, and a paper by J Parrow [23], as its source of inspiration. As in [23], it only works for so-called deterministic processes[2]. We start by recalling the definition of observational congruence, [20].

Definition 1 Observational Congruence. Two processes, say P and Q, are *observationally congruent*, written $P = Q$, if for all $\alpha \in \mathcal{A}ct$,

Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \overset{\alpha}{\Rightarrow} Q'$ and $P' \approx Q'$; (*)
whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \overset{\alpha}{\Rightarrow} P'$ and $P' \approx Q'$. (**)

where $P \overset{\alpha}{\Rightarrow} P'$ iff $P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* P'$. The definition above suggests a way of splitting a goal into simpler equivalence subgoals, one for each derivative. Because we assume that the processes under verification are deterministic, identifying the corresponding Q's is straightforward. The new subgoals are, however, expressed in a different equivalence relation, namely *observational equivalence*, $\approx$. To get round this problem, we exploit the so-called Hennessy theorem: $P \approx Q$ iff either $P = Q$ or $P = \tau.Q$ or $\tau.P = Q$. Thus, the subgoals that we set up take either of the following forms: $P = \tau.Q$, if P has a τ-derivative, or $P = Q$, otherwise. Hence, the definition of the plan method for goal splitting shown in Fig. 3. Notice

Let P_{trans} be the set of transitions ($\rightarrow$) of process P, i.e.
$P_{trans} = \{\alpha_i : P \xrightarrow{\alpha_i} P_i\}$, with indexing set I; and
Let S_{Trans} be the set of experiments ($\Rightarrow$) that process S accepts, i.e.
$S_{Trans} = \{\alpha_j : S \overset{\alpha_j}{\Rightarrow} S_j\}$.
Input:
$\vdash P = S$
Preconditions:
1. sumform(P) and sumform(S);
2. $\forall i \in I.\ P \xrightarrow{\alpha_i} P_i$ implies that $\exists S_i.\ S \overset{\alpha_i}{\Rightarrow} S_i$; and
3. If $\tau \notin P_{trans}$ then $P_{trans} = S_{Trans}$,
Otherwise $P_{trans} \setminus \{\tau\} \subseteq S_{Trans}$

Output:
$$\bigwedge_{i \in I} \vdash P_i = Q_i \wedge \vdash P\{Q_i/P_i\} = S$$
where $Q_i = \tau.S_i$ if $\exists P_i'.\ P_i \xrightarrow{\tau} P_i'$, otherwise $Q_i = S_i$.

Fig. 3. The Goal Splitting Method

that precondition 3 ensures condition (**) in Def. 1. That this holds lies in the fact that we work under the assumption that the processes under verification are deterministic.

[2] Recall that a process P is said to be *deterministic* if for all P_1 and P_2, if $P \overset{\widehat{\alpha}}{\Rightarrow} P_1$ and $P \overset{\widehat{\alpha}}{\Rightarrow} P_2$ implies $P_1 = P_2$. Also recall that $P \overset{\widehat{\alpha}}{\Rightarrow} P'$ iff $P(\xrightarrow{\tau})^* P'$ when $\alpha = \tau$, or $P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* P'$ when $\alpha \neq \tau$.

An example application of this method to (5) gives rise to the following subgoals:

$$\vdash\ C'C = \tau.B_2^1 \tag{6}$$

$$\vdash\ in.\tau.B_2^1 = in.B_2^1 \tag{7}$$

Where the last subgoal is as (5) except that the replacement as indicated by the first subgoal has taken place already.

Notice that the second goal is almost provable. If we got rid of the internal action τ, we would just need appeal to the reflexivity of $=$ to establish that subgoal. This observation should serve as the motivation for the method depicted in Fig. 4, which applies a well-known axiom of the logic.

Interestingly, the four methods above, together with the methods `fertilise` and `elementary` (a simple tautology checker), are able to prove a lot of very simple conjectures. To illustrate, a partial proof plan outline for (1), starting from subgoals (6) and (7) though, is depicted below:

$$
\begin{array}{lrr}
(7) & CC = B_2^0 \vdash \underline{in.\tau.B_2^1} = in.B_2^1 & \tau\text{-elim} \\
 & \ldots \vdash \underline{in.B_2^1} = \underline{in.B_2^1} & \text{elementary} \\
 & & \\
(6) & \ldots \vdash \underline{C'C} = \tau.B_2^1 & \text{Expansion} \\
 & \ldots \vdash \underline{\tau.(CC')} = \tau.B_2^1 & \text{GS}[n_1, n_2] \\
 & & \\
n_1: & \ldots \vdash \tau.B_2^1 = \tau.B_2^1 & \text{elementary} \\
 & & \\
n_2: & \ldots \vdash \underline{CC'} = \underline{B_2^1} & \text{UFI} \\
 & \ldots, CC' = B_2^1 \vdash \underline{CC'} = in.B_2^2 + \overline{out}.B_2^0 & \text{Expansion} \\
 & \ldots \vdash \underline{in.(C'C') + \overline{out}.(CC)} = \underline{in.B_2^2 + \overline{out}.B_2^0} & \text{GS}[n_3, n_4, n_5] \\
 & & \\
n_3: & \ldots \vdash \underline{CC} = \underline{B_2^0} & \text{fertilise} \\
 & & \\
n_4: & \vdots &
\end{array}
$$

The current goal's subexpressions that trigger the application of a method appear underlined, while the method to be applied to the right. The labels n_i are meant to distinguish the subgoals which were given rise to by a method application (c.f. GS which stands for Goal Splitting).

The methods defined in the section below, together with the Goal Splitting method, are the core of this paper.

3.4 The *K*-Normal Method

In CCS, there exist two main (semantic) notions of what counts as the normal form of a expression: fully absorbed form [13], and fully saturated form [18, 19]. For reasons that will become clear soon, we have adopted the former notion.

Input:

$\vdash P = S$

Preconditions:

exp_at(P, Pos, $\alpha.\tau.P'$)

Output:

replace(Pos, $\alpha.P'$, P, Q),

$\vdash Q = S$

Meanings of the meta-logic terms:

- exp_at(E, Pos, $SExp$); $SExp$ is a subexpression of E at position Pos.
- replace(Pos, $SExP$, E, G); G is the result of replacing $SExp$ in E at position Pos.

Fig. 4. The τ-elim (Sub)method

Definition 2 (Fully Absorbed) Normal Form. P is in (fully absorbed) normal form iff

1. It is a sum of prefixed processes, $\sum_{i=1}^{n} \alpha_i.P_i$, where each P_i is a fully absorbed form;
2. For $i \neq j$, $\alpha_i.P_i$ *does not absorb* $\alpha_j.P_j$ and
3. Each P_i above is not of the form $\tau.Q$

The property of absorption is captured by the lemma below, which we state without proof, as it is standard in the literature:

Lemma 3 Absorption Lemma. *If* $P \stackrel{\alpha}{\Rightarrow} P_1$ *and* $P_1 = P_2$*, then* $P + P_2 = P$*.*

The absorption lemma [13], also referred to as the saturation lemma [18, 19, 20], played a pivotal role for proving the axioms systems for finite agents and finite-state agents complete; [13] and [18], respectively. Thus, if P absorbs P_2 then there exists a finite application of the basic axioms of the theory that reduces $P + P_2$ to P.

Obviously, it is also possible to use absorption as a derived lemma to shorten proofs. As we want the program to show the same recursive structure as the specification, any time the former exhibits a transition not contemplated by the latter, we attempt to remove it. Put it in other words, we wish to strip off those summands that can be absorbed by others. This process we call K-normalisation.

Definition 4 k-normal form. P is in K-normal form if it is of the form $\sum_{i=1}^{n} \alpha_i.P_i$, where each P_i is not of the form $\tau.Q$, and none of the summands can absorb one of the others.

The (sub)method shown in Fig. 5 mechanises an application of a stronger version of the absorption lemma:

Input:

$\vdash P = S$

Preconditions:

sumform(P)

behaviour$(P,\ PsB)$

$\exists \alpha_i.P_i, \alpha_j.P_j \in PsB.\ \alpha_i.P_i \stackrel{\alpha_j}{\Rightarrow} P_j$ with $i \neq j$

Output:

absorbs$(\alpha_i.P_i,\ \alpha_j.P_j,\ P,\ P')$

$\vdash P' = S$

Meanings of the meta-logic terms:

- behaviour$(P,\ L)$; $L = \{\alpha_i.P_i\ :\ P \stackrel{\alpha_i}{\rightarrow} P_i\}$
- absorbs$(P_1,\ P_2,\ P,\ NP)$; NP is the process that results after applying the absorption lemma to P, considering that subprocess P_1 absorbs subprocess P_2.

Fig. 5. The Absorption Submethod

1. It is applied to a pair of summands, not to an arbitrary process and a summand; and
2. P_i and P_j must be syntactically equal (matching), not $=$. We have found it easier to deal with this obligation separately, as this condition could be very strong to perform fully K-normalisation, as will see soon.

The definition of the K-normal method is shown in Fig. 6. As expected, it closely follows Def. 4 above.

Body:

iterate[Absorption,τ-elim]

Meanings of the meta-logic terms:

- iterate$[M_1, \ldots, M_n]$; Apply first applicable method in $[M_1, \ldots, M_n]$ until none of them is applicable.

Fig. 6. The K-Normal Method

Consider now for example, that we wish to prove (2). An attempt to proof plan this theorem follows, at the beginning, a pattern which is similar to that of (1). The interesting bit arises in the subgoal below

$$\tau.(CCC') + in.(C'C'C) = \tau.B_3^1 \tag{8}$$

which we aim at transforming into $\tau.(CCC') = \tau.B1$, so that both processes exhibit the same immediate behaviour.

Yet, the disappointing part of the story is that it is not the case that $\tau.(CCC') \stackrel{in}{\Rightarrow} C'C'C$, but $\tau.(CCC') \stackrel{in}{\Rightarrow} C'CC'$, with the remark that $C'C'C \approx$

$C'CC'$. Thus, to coerce the program's behaviour so that it follows the specification's one, we are obliged to *adjust* the process derivatives in the current goal in a way that K-normal applies.

One could be tempted to apply the Goal Splitting method at such point, but the reader should observe that it is hopeless, at least for our current example. To see why, recall that `fertilise` is the chief method that allows one to complete a branch of a proof plan tree. Now, note that at this point we have not applied UFI to a goal of the form $P = B_3^1$. Moreover, note that Goal Splitting would give rise to the goal $C'C'C = \tau.B_3^2$. The problem can now be identified by further noticing that B_3^2 is defined in terms of B_3^1. Therefore, if we applied Goal Splitting to (8), we would end up endlessly exploring a proof plan branch tree, because we have not enough information to close it!

This argument is the motivation for the Adjust method depicted in Fig. 7. Notice that the applicability preconditions of this method are stronger (i.e. cover) than the Goal Splitting ones, fig. 3.

Going back to our example, the proof planning for (8) proceeds as follows:

$$
\begin{array}{lrl}
 & \underline{in.C'C'C + \tau.CCC'} = \tau.B_3^1 & \text{Adjust}[n1, n2] \\
 & & \\
n1: & \underline{C'C'C} = \tau.C'CC' & \text{Expansion} \\
 & \underline{\tau.C'CC'} = \underline{\tau.C'CC'} & \text{elementary} \\
 & & \\
n2: & \underline{in.\tau.C'CC'} + \tau.CCC' = \tau.B_3^1 & \tau - \text{elim} \\
 & \underline{in.C'CC' + \tau.CCC'} = \tau.B_3^1 & \text{Absorption} \\
 & \tau.CCC' = \tau.B_3^1 & \\
 & \vdots &
\end{array}
$$

The reader should observe that, as it stands, the Adjust method cannot cope with *divergent* processes. Recall that P is divergent if $P \stackrel{\tau}{\Rightarrow} P$. Although it is possible to define a (fairly) simple method that could deal with divergence, we shall not pursue this issue here.

3.5 Hierarchy of the Methods

Although CLAM has got different search strategies, so far we have used the depth-first search planner. Hence, methods hierarchy is an important issue for the overall planner's performance. We have found two restrictions in this respect:

1. The UFI method must appear last in the hierarchy, as it could apply a potentially infinite number of times, as could any plan method that embodies some kind of induction;
2. Adjust should appear before Goal Splitting, as the applicability preconditions of the latter method are a subset of those of the former one. It should be noted, however, that there is also a bunch of simple theorems which can be proved without the Adjust method. The characteristic that such theorems seem to have in common is the absence of (very tight) mutual recursion. Example theorems of these kinds are multipliers, and schedulers.

Let $\Delta = \{\alpha_i.P_i \ : \ P \stackrel{\alpha_i}{\rightarrow} P_i$ and irrelevant_summand$(P, \alpha_i.P_i)\}$; and
Let $\Gamma = PsB \setminus \Delta$, where behaviour$(P, PsB)$.
Input:

$\vdash P = S$

Preconditions:

1. sumform(P), and sumform(S)
2. $\forall i.$ if $P \stackrel{\alpha_i}{\rightarrow} P_i$ then $\exists S_i.\ S \stackrel{\alpha_i}{\Rightarrow} S_i$;
3. For each $\mu_j.P_j \in \Delta$, there exists a pair $\langle P_j, Q_j\rangle$, s.t.
   $\alpha_i.P_i \in \Gamma$, and $\alpha_i.P_i \stackrel{\mu_j}{\Rightarrow} Q_j$ with $\neg\ P_j == Q_j$
4. Λ, the set containing all the pairs found in Step 3, with indexing scheme J, is not empty

Output:

$$\bigwedge_{j \in J} \vdash P_j = R_j \wedge \ \vdash P\{R_j/P_j\} = S$$

where $R_j = \tau.Q_j$ if $\exists P_j^{'}.\ P_j \stackrel{\tau}{\rightarrow} P_j^{'}$, otherwise $R_j = Q_j$.

Meanings of the meta-logic terms:

- $E == F$: Syntactic equality (matching).
- irrelevant_summand(P, P_i) means summand P_i in sumform P is irrelevant. A summand is said to be irrelevant iff it could potentially be absorbed by another. Note that all but one duplicated summands are classified as irrelevant.

Fig. 7. The Adjust Method

4 Results, Limitations and Further Work

Using the methods described in the previous section, we have managed to plan proofs for a bunch of simple theorems drawn from the literature: cyclers, buffers, multipliers, schedulers, semaphores, etc.

Yet, the real benefits of having a theorem proving based framework for the verification of CCS programs become apparent when the processes in question are expressed in the full language. In particular, these tools allow one to reason about systems that contain infinite states or that comprise, though a finite, an unbound number of components. A proof for conjectures of these kinds usually proceeds by induction on the number of components. Fortunately, there already exists a great deal of experience implemented in CLAM for proving theorems using structural induction [4]. By extending the UFI method in the obvious way to cope with parameterised processes, we have been able to plan proofs for most of the theorems that appear in [20], Chap. 6, e.g. counters, stacks, queues, etc[3], with the help of lemmas, namely: associativity and idempotency of linking operators: $P^\frown(Q^\frown R) = (P^\frown Q)^\frown R$, and $P^\frown P = P$, respectively. Proving these theorems is certainly beyond the current capabilities of the Edinburgh Concurrency Workbench [9].

[3] N.B. In all of these theorems, we have abstracted data away, and concentrate only in the actual state of the system, e.g. size of the queue, stack, etc.

To see how rippling [7] helps in the verification task, consider the counter theorem of [20][pages 132-133]. The step case of an inductive proof on n, the state of the counter, proceeds as follows:

$$C(n) = Counter(n) \vdash C(\boxed{\mathrm{s}(\underline{n})}^{\uparrow}) = Counter(\boxed{\mathrm{s}(\underline{n})}^{\uparrow})$$

`ripple` followed by `fertilise` transform the above into:

$$\ldots \vdash C \frown Counter(n) = Counter(\mathrm{s}(n))$$

After an application of the sequence of methods: UFI, Expansion, K-normal, and Goal Splitting, we end up with the subgoal below:

$$\ldots \vdash (C \frown C) \frown Counter(n) = C \frown (C \frown Counter(n)$$

After ground difference unifying [2] both sides of the equation, we get the required annotations for applying the wave-rule derived from the associativity of $\frown$, this leaves a readily provable subgoal.

So far, we have made our system to plan proofs for a total of 16 conjectures. It succeeded in all of the cases. These results are so encouraging that we intend to develop further methods to cope with the issues below.

Recursion Analysis. A successful application of the UFI inference rule also depends on the recursive structure of the processes under verification. To see why, consider the processes $A \stackrel{\mathrm{def}}{=} a.a.A$ and $B \stackrel{\mathrm{def}}{=} a.B$. Now, notice that we would not be able to prove that $A = B$, if we took B as the target behaviour, simply because A steps down in double steps, while B in single ones. Note that in situations like these, fertilisation is never applicable, even though the recursive terms have been unwound more than once.

Because we do not look ahead into the behaviour of the processes under verification, we miss this crucial information. We are exploring the possibility of using recursion analysis techniques [5] to get round this problem. Yet, as there is also the issue of how much analysis is to be done before rule application, we are considering the use of plan critics [16] to perform this analysis only when a proof plan attempt fails.

Miscellaneous. There are lots of issues which we want to consider: divergence, the full version of the language, etc. However, one of the avenues we shall definitely explore has to do with the detection, identification, and correction of faulty programs or specifications. As in [21], using the meta-level reasoning embedded in our methods, we shall analyse any failure or partial success in a proof plan attempt, to identify faults in the conjecture at hand and propose candidate patches.

Before going on to the next section, we shall remark again that we are only planning a proof for a theorem. We are exploring the possibility of mechanising the logic in a generic theorem prover [24]. With this, we would be able to test out the actual plans, and if successfull build up a fully automatic tool.

5 Conclusions and Comparison to Related Work

To the best of the authors' knowledge, no previous attempts have been made to fully automate the verification of CCS programs using a theorem-proving based framework. There exist methods that automate some of the steps involved, though. For example, [15] presents a strategy for normalising divergence-free recursive processes.

Some systems, e.g. PAM, make use of automata-theory based tools to extend or provide some level of automation, while some other systems, e.g. [10], allow the user to tackle a problem by using different proof methods. Yet, integrating different tools into a single unified framework is not usually straightforward, due to the differences in the object-level theory representations. This also has a negative impact when it comes to enhancing the capabilities of a system.

On the other hand, our approach benefits from the operational characterisation of processes to guide a proof plan formation. Moreover, not only does the modularity of proof planning make it easy the process of development and maintenance, but it also facilitates the design and implementation of general tactics. In principle, it could be used in cooperation with any tactic-based theorem prover.

References

1. Bergstra, J.A. and Klop, J.W.: 'Algebra for Communicating Processes', *Journal of Theoretical Computer Science* **37**, pages 77-121, 1985.
2. Basin, D. and Walsh, T.: 'Difference Unification', in the Proc. of the *13th Intern. Joint Conference on Artificial Intelligence*, Bajcsy, R. (Ed.), pages 116-122. Morgan Kaufmann, 1993.
3. Bundy, A.: 'The Use of Explicit Plans to Guide Inductive Proofs', in the Proceedings of the *9th Conference on Automated Deduction, CADE*, Lusk, R. and Overbeek, R.(Eds.), pages 111-120. Springer-Verlag 1988.
4. Bundy, A. and van Harmelen, F. and Hesketh, J. and Smaill, A.: 'Experiments with Proof Plans for Induction'. *Journal of Automated Reasoning* **7**, pages 303-324, 1991.
5. Bundy, A. and van Harmelen, F. and Hesketh, J. and Smaill, A. and Stevens, A.: 'A rational reconstruction and extension of recursion analysis', in the Proc. of the *11th International Joint Conference on Artificial Intelligence*, Sridharan, N.S. (Ed.), Morgan Kaufmann, pages 359-365, 1989.
6. Bundy, A. and van Harmelen, F. and Horn, C. and Smaill, A.: 'The OYSTER-CLAM system', in the Proc. of the *10th International Conference on Automated Deduction*, Stickel, M.E. (Eds.), Springer-Verlag, Lecture Notes in Artificial Intelligence No. 449, pages 647-648, 1990.
7. Bundy, A. and Stevens, A. and van Harmelen, F. and Ireland, A. and Smaill, A." 'Rippling: A Heuristic for Guiding Inductive Proofs', *Journal of Artificial Intelligence* **62**, pages 185-253, 1993.
8. Camilleri, A.J. and Inverardi, P. and Nesi, M.: 'Combining Interaction and Automation in Process Algebra Verification', in the Proc. of *TAPSOFT'91*, Abramsky, S. and Maibaum, T.S.E. (Eds.), pages 283-296. Springer-Verlag, Lecture Notes in Computer Science, v.494, 1991.

9. Cleaveland, R. and Parrow J. and Steffen B.: 'The Concurrency Workbench: A Semantics-based Verification Tool for Finite-State Systems', in [25], pages 24-37.
10. De Nicola, R. and Inverardi, P. and Nesi, M.: 'Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS specifications', in [25], pages 54-67.
11. Godskesen, J.C. and Larsen, K.G¿ and Zeeberg, M.: 'TAV Users Manual, Department of Computer Science, Internal Report, Aalborg University, 1989.
12. Hennessy, M.: *Algebraic Theory of Processes*, The MIT Press, Cambridge, Mass. 1988.
13. Hennessy, M. and Milner, R.: 'Algebraic Laws for Nondeterminism and Concurrency', *Journal of JACM* **32:1**, pages 137-161, 1987.
14. Inverardi, P. and Priami, C.: 'Evaluation of Tools for the Analysis of Communicating Systems', *Bulletin of EATCS* **October 45**, pages 158-185, 1991.
15. Inverardi, P. and Nesi, M.: 'Deciding Observational Congruence of Finite-State CCS expressions by rewriting', Journal of *Theoretical Computer Science* **139**, pages 315-354, 1990.
16. Ireland, A.: 'The Use of Planning Critics in Mechanizing Inductive Proofs'. International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg. Lecture Notes in Artificial Intelligence **624**. Voronkov A. (Ed.). Springer-Verlag. pages 178–189, 1992.
17. Lin, H.: 'PAM: A Process Algebra Manipulator', Computer Science Technical Report, University of Sussex, School of Cognitive and Computing Science, April, 1993.
18. Milner, R.: 'A Complete Axiomatisation for Observational Congruence of Finite-State Agents', Department of Computer Science, University of Edinburgh, LFCS Report Series, ECS-LFCS-86-8, 1986.
19. Milner, R.: 'Operational and Algebraic Semantics of Concurrent Processes', Department of Computer Science, University of Edinburgh, LFCS Report Series, ECS-LFCS-88-46, 1988.
20. Milner, R.: *Communication and Concurrency*. Prentice Hall. London, 1989.
21. Monroy, R. and Bundy, A. and Ireland, A.: 'Proof Plans for the Correction of False Conjectures', in the Proc. of *LPAR'94*, Pfenning, F. (Ed.) Springer-Verlag, Lecture Notes in Artificial Intelligence, v. 822, pages 54-68, 1994.
22. Nesi, M.: 'A Formalization of the Process Algebra CCS in Higher Order Logic', Computer Laboratory, University of Cambridge, Technical Paper 278, 1992.
23. Parrow, J.: 'Submodule Construction as Equation Solving in CCS', Department of Computer Science, University of Edinburgh, LFCS Report Series, ECS-LFCS-87-26. 1987.
24. Richards, B.L. and Kraan, I. and Smaill, A. and Wiggins, G.A.: 'Mollusc: a general proof development shell for sequent-based logics', in the Proc. of the *12th Conference on Automated Deduction, CADE*, Bundy, A. (Ed.), pages 826-830. Springer-Verlag, Lecture Notes in Artificial Intelligence, vol 814, 1994.
25. *Proceedings of the Workshop on Automated Verification Methods for Finite-State Systems*, Sifakis, J. (Ed.), Springer-Verlag, Lecture Notes in Computer Science, v. 407. Grenoble, France. 1989.
26. Stirling, C.: 'Modal and Temporal Logics for Processes', Department of Computer Science, University of Edinburgh, LFCS Report Series, ECS-LFCS-92-221, 1992.

This article was processed using the LaTeX macro package with LLNCS style