Searching for a Solution to Program Verification=Equation Solving in CCS

Searching for a Solution to Program Verification=Equation Solving in CCS*

Raúl Monroy[1] and Alan Bundy[2] and Ian Green[2]

[1] Computer Science Department, ITESM Campus Estado de México
Apdo. Postal 50, Módulo de Servicio Postal Campus Edo. de México del ITESM,
52926 Atizapán, Edo. de México, México,
raulm@campus.cem.itesm.mx
[2] Division of Informatics, The University of Edinburgh,
80 South Bridge, EH1 1HN, Scotland, U.K,
{A.Bundy, I.Green}@ed.ac.uk

Abstract. Unique Fixpoint Induction, UFI, is a chief inference rule to prove the equivalence of recursive processes in CCS [7]. It plays a major role in the equational approach to verification. This approach 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.
The use of UFI, however, has been neglected, because automating theorem proving in this context is an extremely difficult task. The key problem with guiding the use of this rule is that we need to know fully the state space of the processes under consideration. Unfortunately, this is not always possible, because these processes may contain recursive symbols, parameters, and so on.
We introduce a method to automate the use of UFI. The method uses middle-out reasoning and, so, is able to apply the rule even without elaborating the details of the application. The method introduces 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 [8], we have implemented an automated 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.

1 Introduction

The Calculus of Communicating Systems [7] (CCS) is an algebra suitable for modelling and analysing processes. CCS is well-established in both industry and academia and has strongly influenced the design of LOTOS [5].

* The authors are supported in part by grants CONACyT-REDII w/n and EPSRC GR/L/11724.

Unique Fixpoint Induction (UFI) is an inference rule for reasoning about recursive processes in CCS and other process algebras [7]. UFI plays a major role in the equational approach to verification. This approach 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.

The use of UFI has been neglected. This is because automating theorem proving in this context is an extremely difficult task. The key problem with guiding the use of this rule is that we need to know fully the state space of the agents under consideration. Unfortunately, this is not always possible, for these agents may contain recursive symbols, parameters, and so on. We suggest that a proof planning approach can provide significant automation in this context.

Proof planning [1] is a meta-level reasoning technique. 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. We introduce a proof plan to automate the use of UFI, based on middle-out reasoning.

Middle-out reasoning (MOR) is the use of meta-variables to represent unknown, 'eureka' values. These meta-variables are gradually refined as further proof steps take place. Control of this refinement is the key in the success of this approach and proof planning provides the necessary means to guide the selection of instantiations and proof steps.

Using MOR, we can apply UFI even without elaborating the details of the application. We approach a verification problem by generalising the input conjecture in a way that the use of UFI becomes immediate. If successful, the proof of the generalised goal is then used to justify the initial one. To generalise the input conjecture, we create a new process family and replace it for the process under verification. The new process family and the specification have similar size and structure, hence facilitating the use of UFI. The new process familiy is under specified, because — via MOR — it contains meta-variables representing those bits of the process's state space that, at introduction time, were not known. We therefore change from equation verification to equation solving.

The proof plan presented here is an extension of the equational verification plan developed by Monroy, Bundy and Green [8]. The proof plan contains three additional strategies, with which it fully automatically decides when and how to apply UFI, as well as driving the solution of equations. The proof plan increases the number of verification problems that can be dealt with fully automatically, thus improving upon the current degree of automation in the field.

Overview of Paper In §2 we describe CCS and UFI. In §3 we characterise the kinds of proofs we shall automate, highlighting search control issues. In §4 we describe proof planning and the verification plan. In §5, §6 and §7 we introduce a method for guiding the use of UFI, the generalisation of the input goal and equation solving. We illustrate aspects of this method with a running example in §8. Finally, we summarise experimental results, discuss related work, as well as drawing conclusions in §9.

2 CCS

Terms of CCS represent agents. Agents are circumscribed by their entire capabilities of interaction. Interactions occur either between two agents, or between an agent and its environment. These communicating activities 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.

2.1 Syntax and Semantics

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: $b \rightarrow A_{\langle x_1,\ldots,x_n\rangle} \stackrel{\text{def}}{=} E$. E does not contain free parameter variables other than $x_1, \ldots, x_n$. $\rightarrow$ denotes logical implication.

We assume parameter expressions, e, built from parameter variables $x, y, \ldots$, parameter constants $v_1, v_2, \ldots$ and any operators we may require, $\times, \div, \text{even}, \ldots$. We also assume boolean expressions, b, with similar properties except that they are closed under the logical connectives. Parameter constants might be of any type.

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. 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*, $\mathbf{0}$, capable of no actions whatever; $\mathbf{0}$ 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*, $\sum_{i \in \{e\} \cup I} E_i = E(e) + \sum_{i \in I} E_i$; and iv) *Infinite Summation over natural numbers*.

Parallel Composition, ($\mid$), 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*, ($\backslash$), is used for internalising ports: $E \setminus L$ behaves like E, except that it cannot execute any action $\ell \in L \cup \overline{L}$.

Processes are given a meaning via the labelled transition system $(\mathcal{E}, \mathcal{A}ct, \stackrel{\alpha}{\rightarrow})$, where $\stackrel{\alpha}{\rightarrow}$ is the smallest transition relation closed under the following transition rules (the symmetric rule for $\mid$ has been omitted:)

$$\frac{}{\alpha.E \stackrel{\alpha}{\rightarrow} E} \qquad \frac{E_j \stackrel{\alpha}{\rightarrow} E'}{\sum_{i\in I} E_i \stackrel{\alpha}{\rightarrow} E'}\ (j \in I) \qquad \frac{E \stackrel{\alpha}{\rightarrow} E'}{A \stackrel{\alpha}{\rightarrow} E'}\ (A \stackrel{\text{def}}{=} E) \qquad \frac{E \stackrel{\alpha}{\rightarrow} E'}{E[f] \stackrel{f(\alpha)}{\rightarrow} E'[f]}$$

$$\frac{E \stackrel{\alpha}{\rightarrow} E'}{E \mid F \stackrel{\alpha}{\rightarrow} E' \mid F} \qquad \frac{E \stackrel{\ell}{\rightarrow} E' \quad F \stackrel{\overline{\ell}}{\rightarrow} F'}{E \mid F \stackrel{\tau}{\rightarrow} E' \mid F'} \qquad \frac{E \stackrel{\alpha}{\rightarrow} E'}{E \setminus L \stackrel{\alpha}{\rightarrow} E' \setminus L}\ (\alpha, \overline{\alpha} \notin L)$$

The interpretation of these rules should be straightforward and is not discussed here further.

Process (states) are related to each other: E' is a *derivative* of E, whenever $E \stackrel{\alpha}{\rightarrow} E'$. Similarly, E' is a *descendant* of E, whenever $E \stackrel{\alpha}{\Rightarrow} E'$, where $\stackrel{\alpha}{\Rightarrow}$ is given as $(\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha}{\rightarrow} (\stackrel{\tau}{\rightarrow})^*$.

2.2 Unique Fixpoint Induction

Unique Fixpoint Induction (UFI) is a rule for reasoning about recursive processes [7]. UFI states that two processes are equivalent, if they satisfy the same set of (recursive) equations, so long as the 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* in E if each occurrence of X is within some subexpression $\ell.F$ 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.

The notions of guardedness and sequentially are extended to sets of equations in the obvious way: the set of equations $\widetilde{X} = \widetilde{E}$ is guarded (respectively sequential) if $E_i (i \in I)$ contains *at most* the variables $X_i (i \in I)$ free, and these variables are all guarded (respectively sequential) in E_i.

Let the expressions $E_i\,(i \in I)$ contain at most the variables $X_i\,(i \in I)$ free, and let these variables be all guarded and sequential in each E_i. Then, the UFI inference rule is as follows:

$$\text{If } \widetilde{P} = \widetilde{E}\{\widetilde{P}/\widetilde{X}\} \text{ and } \widetilde{Q} = \widetilde{E}\{\widetilde{Q}/\widetilde{X}\}, \text{ then } \widetilde{P} = \widetilde{Q}$$

There is one aspect to the UFI rule that is worth mentioning: it reasons about process families. We cannot prove that some individuals satisfy a property without proving that such a property is satisfied by all. This is because the equations of each individual are incomplete and, by definition, to apply UFI the system of equations, $X_i = E_i (i \in I)$, must be such that each E_i contains *at most* the variables $X_i (i \in I)$ free.

3 Program Verification

We use CCS both as a programming language and as a specification language. Specifications are assumed to be explicitly given by means of C-declarations.

Definition 1 (C-declaration). *A set of definitions* $\widetilde{C} \to \widetilde{S} \stackrel{def}{=} \widetilde{E}\{\widetilde{S}/\widetilde{X}\}$ *is called a* C-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*;*
2. $S_i = S_j$ *implies* $i = j$*; and*
3. *the conditions* $C_i\,(i \in I)$ *are all fundamental, in that they cannot be given as the disjunction of two or more relations. For example,* $\geq$ *is not fundamental since* $x \geq y$ *means that* $x > y$ *or* $x = y$*.*

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

$$\begin{aligned} n \neq 0 \to \quad & Buf_{\langle n,0\rangle} \stackrel{\text{def}}{=} in.Buf_{\langle n,s(0)\rangle} \\ n > s(k) \to\ & Buf_{\langle n,s(k)\rangle} \stackrel{\text{def}}{=} in.Buf_{\langle n,s(s(k))\rangle} + \overline{out}.Buf_{\langle n,k\rangle} \\ n = s(k) \to\ & Buf_{\langle n,s(k)\rangle} \stackrel{\text{def}}{=} \overline{out}.Buf_{\langle n,k\rangle} \end{aligned}$$

where s stands for the successor function on natural numbers.

The systems under verification 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:

$$\begin{array}{l} n = 0 \ \to\ C^{s(n)} = C \\ n \neq 0 \ \to\ C^{s(n)} = C^\frown C^{(n)} \end{array} \quad \text{where} \quad \begin{array}{l} C \stackrel{\text{def}}{=} in.D \quad D \stackrel{\text{def}}{=} \overline{out}.C \\ P^\frown Q \stackrel{\text{def}}{=} (P[c/out] \mid Q[c/in]) \setminus \{c\} \end{array}$$

Let P be a concurrent form and let S be a C-declaration. Then we call $P = S$ an instance of the *verification problem*. This is an example verification problem:

$$\forall n\!:\!\text{nat}.\ n \neq 0 \to C^{(n)} = Buf_{\langle n,0\rangle} \tag{1}$$

it states that a chain of n copies of a buffer of size one behaves as a buffer of size n.

We conclude this section noting that, since specifications are given, we need not use the general form of UFI. Instead we use the following, stronger rule:

$$\frac{\widetilde{P} = \widetilde{E}\{\widetilde{P}/\widetilde{X}\}}{\widetilde{P} = \widetilde{S}} \quad \widetilde{S} \stackrel{\text{def}}{=} \widetilde{E}\{\widetilde{S}/\widetilde{X}\} \tag{2}$$

We call $\widetilde{P} = \widetilde{E}\{\widetilde{P}/\widetilde{X}\}$ the *output equation set* and $\{\widetilde{P}/\widetilde{X}\}$ the *process substitution*.

Having given the verification problem, we now attempt to give the reader a flavour as to the difficulties of automating the search for a verification.

3.1 Search Control Problems within Program Verification

How should one proceed to verify (1)? Clearly, $Buf_{\langle n,k\rangle}$ and $C^{(n)}$ are both recursive and, so, they suggest the use of induction. However, $C^{(n)}$ suggests the use of a sort of induction other than UFI, namely: structural induction, or induction for short.

Induction and UFI play different roles; a proof may resort to both. The use of these rules has be coordinated. Induction prior to UFI is not a good rule of thumb and vice versa. For our example, the use of UFI is a bad starting point, as it is difficult to compute the process substitution. The root of the problem is that the recursive symbol $C^{(n)}$ is not given in terms of the indexing scheme, $k \in \{1, \ldots, n\}$, used to define *Buf*. Fortunately, induction, if successfully applied, eliminates recursive symbols. Thus, heuristics are required in order to coordinate the use of UFI and induction.

The process substitution is the key for building the output equation set in an application of UFI. However, experience indicates that computing process substitutions is an extremely difficult task. Whenever the use of UFI is suggested but the process substitution is partially solved, we let an application of the UFI method introduce a new process family to replace the process under verification. The new process family and the specification are similar in both size and structure. So the use of UFI is immediate. The proof of the new goal is used to justify the initial one. The process of replacing the program to be verified for a new process family is called a *generalisation*. Generalisation increases the search branching rate and, hence, heuristics are required in order to control its application.

The new process family is in a way incomplete. In place of some $P_i\,(i \in I)$ we put meta-variables. We use the term *meta-variable* to denote a first-order (or higher-order) unknown entity in the object-level theory. We call $\boldsymbol{\mathcal{M}}$ the set of meta-variables, and let $\mathcal{M}_0, \mathcal{M}_1, \ldots$ range over $\boldsymbol{\mathcal{M}}$. Introducing meta-variables, we can use UFI still, but at the expense of solving each equation at verification time. Thus, heuristics are required in order to control equation solving.

Summarising, automating the use of UFI prompts three major challenges: i) when and how to use UFI; ii) when and how to use generalisation; and iii) guide equation solving. These issues explain why radical measures are called for. Fortunately, as discussed below, proof planning is, at least partially, an answer to these problems.

4 Proof Planning

Proof planning [1] is a meta-level reasoning technique, developed especially as a search control engine to automate theorem proving. Proof planning works in the context of a tactical style of reasoning. It uses AI planning techniques to build large complex tactics from simpler ones, hence outlining the proof while emphasising key steps and structure.

Methods are the building-blocks of proof planning. A *method* is a high-level description of a tactic. It specifies the preconditions under which the tactic is

applicable, without actually running it, and the effects of its application. The application of a method to a given goal consists in checking the preconditions against the goal and then determining the output new goals by computing the effects.

Proof planning returns a *proof plan* (i.e., a tactic), whose execution, in the normal case of success, guarantees correctness of the final proof. Proof planning is cheaper than searching for a proof in the underlying object theory. This is both because each plan step covers a lot of ground steps, and because the method preconditions dramatically restrict the search space.

Inductive proof planning [3] is the application of proof planning to inductive theorem proving. It has been successfully applied to various domains, including software and hardware development. Inductive proof planning is characterised by the following methods: The *induction* method selects the most promising induction scheme via a process called *rippling analysis*. 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 by means of exhaustive symbolic evaluation and other routine reasoning.

Similarly, the step case(s) of proofs by induction are dealt with by the *wave* and *fertilise* methods. Wave applies *rippling* [2], a heuristic that guides transformations in the induction conclusion to enable the use of an induction hypothesis. This use of an induction hypothesis, called *fertilisation* — hence the name of the method — is a crucial part of inductive theorem proving and it is handled by fertilise.

Rippling exploits the observation that an initial induction conclusion is a copy of one of the hypotheses, except for extra terms, e.g., the successor function, wrapping the induction variables. By marking such differences explicitly, rippling can attempt to place them at positions where they no longer prevent the conclusion and hypothesis from matching. Rippling is therefore an annotated term-rewriting system. It applies a special kind of rewrite rule, called a *wave-rule*, which manipulates the differences between two terms while keeping their common structure intact.

Monroy, Bundy and Green have extended inductive proof planning with a special CCS proof plan [8]. The CCS proof plan is circumscribed by the following methods: The *expansion* method transforms a concurrent form into a sum of prefixed processes. The *absorption* method gets rid of CCS terms that are redundant with respect to behaviour. The *goalsplit* method equates each process summand on one side of the equation with one on the other side, returning a collection of subgoals, each of the form $\alpha.P = \alpha.Q$. Finally, the *action* method solves equations of the form $\alpha.P = \alpha.S$ using Hennessy's theorem [7].

Monroy, Bundy and Green did not allow specifications to be recursive. In what follows, we shall show how to extend the verification plan to deal with recursion.

5 The UFI Method

The UFI method is concerned with when to apply the UFI rule. An application of the UFI method is *successful* if it introduces a minimum number of meta-variables, ideally zero. With this, it avoids unnecessary equation solving steps. The moral is that the more derivatives we associate, the more chances we have of finding a verification proof.

Concurrent forms often contain a lot of recursive symbols. Recursive symbols get in the way of an application of UFI, especially when they are not given in terms of the indexing scheme used by the specification. Then it is necessary to use induction prior to UFI in order to remove them. However induction prior to UFI is not generally a good rule of thumb.

Induction before UFI may yield a waste of effort if applied upon an index variable. This is because it would yield new goals involving P_k, for some $k \in I$, an individual of the process family. However, recall that we cannot prove that some individual satisfies a property without proving that such a property is satisfied by all. So, each goal returned by induction will need to be dealt with using UFI.

Summarising, the strategy to coordinate the use of induction and UFI is as follows. Use induction prior to UFI both if the verification problem contains recursive function symbols not given in terms of the indexing scheme, and if the selected induction variable is other than an index variable in the specification. If these conditions do not hold, UFI is applied first.

6 The Generalise Method

We now introduce a method, called *generalise*, which deals with the problem of using UFI under incomplete information. The rationale behind generalise is that we can apply UFI still, even without elaborating the details of the application. The computation of the process substitution is therefore postponed, leaving it partially defined. Subsequent planning steps are then used in order to elaborate upon the output plan, which will hopefully yield an object-level proof. The use of meta-variables to represent unknown, 'eureka' values is called *middle-out reasoning*.

Upon application, generalise builds a conditional, non-recursive process function, say $\mathcal{F}_{\mathcal{P}}$, which is put in place of the process family under verification. $\mathcal{F}_{\mathcal{P}}$ has one exceptional feature: not only does it embrace the original process, but it is also given by the same number of equations that define the specification. Thus, each equation in $\mathcal{F}_{\mathcal{P}}$ is pairwise related to one, and only one, equation in S: the application of UFI becomes immediate. $\mathcal{F}_{\mathcal{P}}$ is of course initially under-specified (and so will the output equation set). This is because all of the missing process derivatives are substituted with meta-variables, each of which it is hoped will be fixed at later planning steps. We hence change from equation verification to equation solving, which we shall discuss in §7.

Consider a verification problem, $P = S$, where the use of UFI is suggested, but where the process substitution is only partially solved. Then, to generalise the verification problem, proceed as follows:

1. Introduce a new process symbol, say $\mathcal{F}_{\mathcal{P}}$.
2. Define the new process family, so that it matches the size of S as well as its structure:

$$C_i \to H_i\{\widetilde{\mathcal{F}_{\mathcal{P}}}/\widetilde{S}\} \stackrel{\text{def}}{=} \mathcal{M}_i(P^{\to}) \text{ only if } C_i \to H_i \stackrel{\text{def}}{=} B_i \ \in \ \widetilde{S} \stackrel{\text{def}}{=} \widetilde{E}\{\widetilde{S}/\widetilde{X}\}$$

   where $P^{\to}$ denotes some P-derivative.
3. Refine the definition of $\mathcal{F}_{\mathcal{P}}$, using $P = S$ in order to fix at least one of the equation bodies, $\mathcal{M}_i$. If this process fails, so will generalise.
4. Simplify the equation system, getting rid of any redundant terms.
5. Finally, replace the original conjecture for $\mathcal{F}_{\mathcal{P}} = S$.

7 Verification = Equation Solving in CCS

Now we present a search control strategy to automatically solve the output equation set. The strategy has been designed to solve equations of the form $P = E$, where E may contain meta-variables, but P may not. Each equation in the output set is, thus, considered independently, one at a time. However, the result of any equation solving step is spread throughout the entire plan. The strategy's guiding factor is the expected behaviour that each meta-variable in E should satisfy. We call this (behavioural) constraint the *context*.

The context is nothing but a description of the process' intended behaviour, i.e., it is related to the specification. If the definition of the specification involves mutual recursion, we add the context to every subgoal so that the equation solving strategy can use it. Accordingly, we let each subgoal in the output equation set take the following schematic form:

$$\boxed{\{\, P_h = E_h\{\widetilde{P}/\widetilde{X}\} \ : \ h \in I \setminus \{i\}\,\}}\ldots \vdash \ P_i = E_i\{\widetilde{P}/\widetilde{X}\} \qquad (i \in I)$$

The context is a decoration. So, it is marked with a dashed box in order to make it distinguishable from the actual hypotheses.

Equation solving is applied only when the current equation subgoal is balanced. An equation is balanced iff it has the same number of process summands on each side of the equality. Equation solving involves both a process, and a meta-variable. The process and the meta-variable must be prefixed by the same action, and must appear at a different side of the equality.

The equation solving strategy is specified as follows: Solve $\mathcal{M} = P$, only if $\mathcal{M}$ and P have identical local behaviour, in symbols:

$$\forall \alpha \in \mathcal{A}ct.\ \mathcal{M} \stackrel{\alpha}{\to} \text{ if and only if } P \stackrel{\alpha}{\to}$$

While the intended behaviour of $\mathcal{M}$ is extracted from the context, the actual behaviour of P is computed using process expansion.

8 A Worked Example

We illustrate the strength of our approach by showing that full automation is achieved in proving (1). We have chosen this example because it is a challenge case study for state-of-the-art automated verification systems.

When input (1), the planner applied induction prior to UFI. This is as expected since n in $C^{(n)}$ is not an index variable in the definition of *Buf*. Afterwards, the planner solved both the base case and the step case, also fully automatically. In the step case, the planner made use of generalise when induction returned the goal below:

$$\forall n\!:\!\text{nat}.\ n \neq 0 \rightarrow C^\frown Buf_{\langle n,0\rangle} = Buf_{\langle s(n),0\rangle} \tag{3}$$

Generalise then automatically tackled the new problem outputting the formula $\forall n : \text{nat}.\ n \neq 0 \rightarrow \mathcal{F}_{\mathcal{P}\langle s(n),0\rangle} = Buf_{\langle s(n),0\rangle}$, where $\mathcal{F}_{\mathcal{P}}$ was initially given by:

$$\begin{aligned}
&s(n) \neq 0 \rightarrow \mathcal{F}_{\mathcal{P}\langle s(n),0\rangle} \stackrel{\text{def}}{=} C^\frown Buf_{\langle n,0\rangle}\\
&s(n) > j \rightarrow \mathcal{F}_{\mathcal{P}\langle s(n),j\rangle} \stackrel{\text{def}}{=} \mathcal{M}_1(\mathcal{M}_{11}(C,D), Buf_{\langle n,\mathcal{M}_{12}(j)\rangle})\\
&s(n) = j \rightarrow \mathcal{F}_{\mathcal{P}\langle s(n),j\rangle} \stackrel{\text{def}}{=} \mathcal{M}_2(\mathcal{M}_{21}(C,D), Buf_{\langle n,\mathcal{M}_{22}(j)\rangle})
\end{aligned}$$

Note that the $\mathcal{M}$s are all meta-variables. Also note that $\mathcal{M}_{11}(C,D)$ sufficed to represent the state space of C: $\{C,D\}$, and similarly for $\mathcal{M}_{21}(C,D)$ and $Buf_{\langle n,\mathcal{M}_{22}(j)\rangle}$.

With the revised conjecture, the use of UFI was immediate, yielding:

$$\begin{aligned}
&\vdash s(n) \neq 0 \ \rightarrow\ PF_{\langle s(n),0\rangle} = in.PF_{\langle s(n),s(0)\rangle}\\
&\vdash s(n) > k \ \rightarrow\ PF_{\langle s(n),s(k)\rangle} = in.PF_{\langle s(n),s(s(k))\rangle} + \overline{out}.PF_{\langle s(n),k\rangle}\\
&\vdash s(n) = k \ \rightarrow\ PF_{\langle s(n),s(k)\rangle} = \overline{out}.PF_{\langle s(n),k\rangle}
\end{aligned} \tag{4}$$

Each goal in the output equation was tackled successfully. For example, when the equation solving strategy had been already used to successfully fix $\mathcal{M}_1(\mathcal{M}_{11}(C,D), Buf_{\langle n,\mathcal{M}_{12}(j)\rangle})$ to $C^\frown Buf_{\langle n,j\rangle}$, the current definition of $\mathcal{F}_{\mathcal{P}}$ was as above except for the second case: $s(n) > j \rightarrow \mathcal{F}_{\mathcal{P}\langle s(n),j\rangle} \stackrel{\text{def}}{=} C^\frown Buf_{\langle n,j\rangle}$. The working subgoal then was (4):

$$\begin{aligned}
&\boxed{\begin{array}{l}
s(n) \neq 0 \rightarrow \quad \mathcal{F}_{\mathcal{P}\langle s(n),0\rangle} = in.\mathcal{F}_{\mathcal{P}\langle s(n),s(0)\rangle}\\
s(n) > k \rightarrow \mathcal{F}_{\mathcal{P}\langle s(n),s(k)\rangle} = in.\mathcal{F}_{\mathcal{P}\langle s(n),s(s(k))\rangle} + \overline{out}.\mathcal{F}_{\mathcal{P}\langle s(n),k\rangle}\\
s(n) = k \rightarrow \mathcal{F}_{\mathcal{P}\langle s(n),s(k)\rangle} = \overline{out}.\mathcal{F}_{\mathcal{P}\langle s(n),k\rangle}
\end{array}}\\
&n : \text{nat},\ n \neq 0\\
&\vdash\\
&s(n) > s(k)\ \rightarrow\ C^\frown Buf_{\langle n,s(k)\rangle} = in.\mathcal{F}_{\mathcal{P}\langle s(n),s(s(k))\rangle} + \overline{out}.\mathcal{F}_{\mathcal{P}\langle s(n),k\rangle}
\end{aligned}$$

With this goal, a further application of equation solving was required. It occurred after the application of casesplit upon the partition: $[n > s(k), n = s(k)]$. The

interesting case is $n = s(k)$, where the planner applied expansion, together with sym_eval, leaving:

$$\ldots \vdash \; in.(D^\frown Buf_{\langle n,s(k)\rangle}) + \overline{out}.(C^\frown Buf_{\langle n,k\rangle}) = \\ in.\mathcal{M}_2(\mathcal{M}_{21}(C,D), Buf_{\langle n,\mathcal{M}_{22}(s(k))\rangle}) + \overline{out}.(C^\frown Buf_{\langle n,k\rangle})$$

Then, second-order matching suggested the following substitution:

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

The solution met the constraint imposed by the context, because, by expansion, $D^\frown Buf_{\langle n,s(k)\rangle}$ equals $\overline{out}.(D^\frown Buf_{\langle n,k\rangle})$. So, it was accepted, yielding a refinement on the third defining equation of $\mathcal{F}_\mathcal{P}$: $s(n) = j \rightarrow \mathcal{F}_{\mathcal{P}\,\langle s(n),j\rangle} \stackrel{\mathrm{def}}{=} D^\frown Buf_{\langle n,j\rangle}$, which is just as required.

9 Results, Related Work and Conclusions

Table 1 gives some of the example systems with which we tested our proof plan. PT stands for the total elapsed planning time, given in seconds. Space constraints do not allow us to provide a full description of each verification problem. These example verification problems are all taken from [7]. They all involve the use of generalise, UFI and equation solving. The success rate of the proof plan was

Conjecture	Description	PT (sec)
n-bit counter	A chain of n cells, each acting as a 1-bit counter, implements a counter of size n	57
n-bit sorter	A chain of n cells, each acting as a filter implements a bubble-sort algorithm for a string of elements	614
semaphore array	A set containing n 1-bit semaphores implements a semaphore of size n	992
cycler array	A collection of cyclers computes the sequence $a_1.\cdots.a_n.a_1\cdots$ indefinitely	411

Table 1. Some example verification conjectures

83%, with an average total elapsed planning time of 750 seconds, and 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, including the planner, are available upon request, by sending electronic-mail to the first author.

CCS has been mechanised in several proof checkers. For example, Cleaveland and Panangaden describe an implementation of CCS in Nuprl [4]. 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 [9]. 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. Lin reported an implementation of a CCS like value-passing process algebra in VPAM [6], a generic proof checker. These verification 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 Lin, we do not accommodate truly value-passing systems, relying upon infinite Summation to simulate it.

Concluding, the Verification planner handles the search control problems prompted by the use of UFI 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. We intend to apply the verification planner to the verification of larger, industrial strength examples, and see what extensions are required.

References

1. 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.
2. 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.
3. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.
4. R. Cleaveland and P. Panangaden. Type Theory and Concurrency. *International Journal of Parallel Programming*, 17(2):153–206, 1988.
5. ISO. *Information processing systems - Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. ISO 8807, 1989.
6. H. Lin. A Verification Tool for Value-Passing Processes. In *Proceedings of* 13^{th} *International Symposium on Protocol Specification, Testing and Verification*, IFIP Transactions. North-Holland, 1993.
7. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.
8. R. Monroy, A. Bundy, and I. Green. Planning Equational Verification in CCS. In D. Redmiles and B. Nuseibeh, editors, *13th Conference on Automated Software Engineering, ASE'98*, pages 43–52, Hawaii, USA, 1998. IEEE Computer Society Press. Candidate to best paper award.
9. M. Nesi. Mechanizing a proof by induction of process algebra specifications in higher-order logic. In K. G. Larsen and S. A., editors, *Proceedings of the 3rd International Workshop in Computer Aided Verification (CAV'91)*. Springer Verlag, 1992. Lecture Notes in Computer Science No. 575.