Mechanizing a Proof by Induction of Process Algebra Specifications in Higher Order Logic *

Monica Nesi
Istituto di Elaborazione dell' Informazione, C.N.R.
via Santa Maria 46, I-56126 Pisa, Italy
&
University of Cambridge, Computer Laboratory
New Museums Site, Pembroke Street, Cambridge CB2 3QG, U. K.

Abstract

When dealing with *inductively defined systems*, correctness proofs of different specifications of the same system cannot be accomodated in a framework based on finite state automata. Instead, these systems can be naturally analysed and verified by manipulating the process algebra specifications by means of equational reasoning. In this paper, we describe an attempt to mechanize a proof by mathematical induction of the correctness of a simple buffer. To achieve this goal, we use the interactive theorem prover HOL to support the theory of observational congruence for CCS, and provide a set of axiomatic proof tools which can be used interactively.

1 Introduction

In the past few years, several verification tools based on process algebras have been proposed for proving properties of concurrent systems [15]. Most of them resort to a finite state automata representation of specifications, which is used to verify equivalences of specifications and to show that a specification satisfies a logical (modal) property by means of some reasonably efficient automatic algorithms. This completely automatic approach has, however, a few problems (e.g. state explosion) and some limitations (e.g. it can deal with only finite state specifications). In such a framework, there is no easy way to accommodate the verification of processes with infinite states or, more generally, to perform incremental or interactive proofs, even though the theory behind the process algebras supports such reasoning. Moreover, even when dealing with (finite state) parameterized systems, the specifications cannot be verified by using finite state machines.

*Research supported by Progetto Finalizzato Informatica, I.E.I.-C.N.R., Pisa, Italy.

In [3, 6] a verification environment which relies on the algebraic nature of the concurrent specification language CCS [13], is described and fully motivated. Such an environment provides the user with facilities to control the analysis and verification phases and to perform proofs automatically. It also allows interactive control when automation is not desirable and permits the user to define sound verification strategies, thus allowing for a better understanding of both the specifications and the correctness criteria one is attempting to verify. This verification environment for CCS is based on the interactive theorem prover HOL [7]. The formal theory for a specific semantics of CCS, namely *observational congruence*, is represented in the logic, and the resulting representation is the basis for higher level verification strategies by mechanized formal proof.

In this paper, we address a particular kind of reasoning, namely *proofs by induction*. We consider concurrent systems with inductive structure and show how a proof of correctness by mathematical induction can be mechanized in HOL. The mechanization exploits the rich set of proof tactics available in the HOL system and the facility for defining new tactics from the built-in ones. It also takes advantage of the subgoal package for backward proofs, thus resulting in quite natural and simple proofs.

In what follows, we first give a brief description of the HOL system. We then introduce the subset of CCS under consideration and show how the syntactic definitions and the axioms for the observational semantics can be formalized in HOL. Next, we illustrate how reasoning by induction can be done in the resulting framework, by proving the correctness of an implementation of a simple buffer. Finally, we discuss related work and possible extensions to the described approach.

2 The HOL System

Higher order logic is a good formalism for mechanizing other mathematical languages because it is both powerful and general enough to allow sound and practical formulations. It has been used to mechanize several logics [8] and process algebras, e.g. CSP [1, 2] and CCS [3]. The theorem prover used in these mechanizations is the HOL system [7], developed by Gordon, which is based directly on the LCF theorem prover [14].

The HOL logic is a variety of higher order logic based on Church's formulation of type theory [7]. In the HOL logic, the standard predicate calculus is extended by allowing variables to range over functions, the arguments of functions can themselves be functions, functions can be written as λ-abstractions and terms can be polymorphic.

The HOL logic is mechanized using the programming language ML [5], which is used to manipulate HOL logic terms and, in particular, to prove that certain terms are theorems. Theorems proved in the system are distinguished from ordinary terms by being assigned a built-in ML type `thm`. To introduce values of type `thm`, they must either be postulated as axioms or deduced from existing theorems by ML programs called *inference rules*.

Certain kinds of axioms are classed as *definitions*. These are axioms of the form $\vdash c = t$ where c is a constant not previously defined and t is a term containing no free variables. Definitions form a conservative extension to the logic, i.e. a sound extension.

A collection of logical types, type operators, constants, definitions, axioms and theorems is called a *theory*. To make a definition, prove a theorem, or declare a new HOL type, one must first enter a theory and, if facts from other theories are to be used, the

relevant existing theories must be declared as *parents*. Theories, therefore, enable a hierarchical organization of facts. A library of theories is available in the HOL system to enable the reuse of established and commonly used theorems. The availability of such a library greatly aids the task of mechanization and reasoning.

To prove a theorem in a theory, one must apply a sequence of steps to either axioms or previously proved theorems by using inference rules (forward proof). The core of the HOL system is made up of a small set of *primitive inference rules* and a small number of definitions and axioms from which all the standard rules of logic are derived.

The HOL system supports another way of carrying out a proof called goal directed proof or backward proof. The idea is to start from the desired result (*goal*) and manipulate it until it is reduced to a subgoal which is obviously true. ML functions that reduce goals to subgoals are called *tactics* and were developed by Milner [14].

As regards goal directed proofs, the HOL system provides a *subgoal package* due to Paulson [14], which implements a simple framework for interactive proofs. A goal can be set by invoking the function g, which initializes the subgoal package with a new goal. The current goal can be expanded using the function e which applies a tactic to the top goal on the stack and pushes the resulting subgoals onto the goal stack. When a tactic solves a subgoal (i.e. returns an empty subgoal list), the package computes a part of the proof and presents the user with the next subgoal. When a theorem is proved, it can be stored in the current theory using several functions. Among the others, `TAC_PROOF` takes a goal and a tactic, and applies the tactic to the goal in an attempt to prove it.

The HOL system also provides functions called *conversions* [14] that map terms t to theorems expressing the equality of that term with some other term, $\vdash t = u$. Various built-in conversions and operators for constructing conversions from smaller ones, and several tactics and operators for constructing tactics from smaller ones and from conversions, played a fundamental role in our mechanization of proof strategies for CCS. Examples of the use of some of these conversions and tactics are given in later sections.

3 CCS in HOL

In this section, familiarity with some of the concepts behind CCS is assumed, so only essential information is presented. We consider *pure* CCS, a subset of the language which does not involve value passing and consists of the inactive process nil, and the following operations on processes: *prefix* (.), *summation* (+), *parallel composition* (|), *restriction* (\), *relabelling* ([]) and *recursion* (rec). The syntax of pure CCS is given below:

$$E ::= \mathsf{nil} \mid u.E \mid E+E \mid E|E \mid E \setminus l \mid E[f] \mid X \mid \mathsf{rec}X.E$$

where X ranges over process variables, l ranges over visible actions, called *labels*, u ranges over actions which are either labels or the invisible action τ, and f ranges over relabelling functions on labels. Labels consist of *names* and *co-names* where, for any name a, the corresponding co-name is written $\overline{a}$. The *complement* operation has the property that $\overline{\overline{l}} = l$, and relabelling co-names has the property that $f(\overline{l}) = \overline{f(l)}$.

The formal interpretation of the above operators is given via an operational semantics [13]. In addition, in the literature several behavioural semantics have been defined, and then characterized in terms of axiomatizations which have been proved sound and

complete for subsets of CCS. The axioms concerning the internal action τ, referred to as *τ-laws*, distinguish the various equivalences. In this paper, we address the theory of *observational congruence* and refer to [13] for the axioms concerning this theory.

Finally, we recall a result which will be used in the correctness proof (Section 4.1). Let $E\{F/X\}$ denote the substitution of F for all free occurrences of X in E. When dealing with recursive equations, two processes P and Q which are observational congruent to the expressions $E\{P/X\}$ and $E\{Q/X\}$ respectively, denote the (unique) solution of the recursive equation $X = E$, if X is sequential and guarded in the expression E [13].

3.1 Mechanization of CCS

In this section, we recall briefly some aspects about the formalization of pure CCS in HOL; the reader should refer to [3] for more details about the mechanization of the axioms.

The CCS syntax presented earlier can be mechanized by defining in HOL a concrete data type *CCS* in terms of all its possible constructors. This is done by using a built-in facility for automatically defining concrete recursive data types from a specification of their syntax [11]. The type definition for *CCS* can be done as follows:

CCS = nil |
var *string* |
prefix *action CCS* |
sum *CCS CCS* |
restr *CCS label* |
relab *CCS relabelling* |
par *CCS CCS* |
rec *string CCS*

where nil, var, prefix, sum, restr, relab, par and rec are distinct constructors, *label* and *action* are syntactic types defined as follows:

label = const *string* | compl *label* $\qquad$ *action* = tau | label *label*

and *relabelling* is an abbreviation for the function type *label*→*label*.

To avoid using a verbose prefix notation, the parsing and pretty-printing facilities in HOL are extended to accept input, and print output, almost identical to that usually associated with CCS.[1] Since the above prefix constructors are therefore only used internally by the system, we re-adopt the standard CCS syntax for the rest of the paper.

The next steps in the formalization should be to mechanize the operational semantics of the CCS operators, define the notion of observational congruence, and derive its axiomatization. As stated in [3], some work has already shown that it is feasible to mechanize process algebra semantics in HOL and to mechanically prove their axiomatization [1, 2, 12]. In the present work, we are interested in practical reasoning tools at the axiomatic level. Thus, we directly assert the axioms for observational congruence in the HOL logic. We intend to mechanize the operational semantics and to derive the axioms later on, to remove the inconsistency of asserting axioms on a free type such as *CCS* and to ensure that our mechanization of the CCS theory is sound.

[1] Modulo ascii syntax, e.g. $\overline{a}$ is written $-a$, and τ is written tau.

Having defined the appropriate types and syntactic constructors in the HOL logic, it is straightforward to assert most of the axioms. Due to lack of space, we present only some of them below, namely associativity for summation, distributivity of relabelling with respect to summation and the law for the restriction of a prefix process [13, 3]:

$$\vdash \forall P\, Q\, R : CCS.\ P + (Q + R) = (P + Q) + R$$
$$\vdash \forall (P\, Q : CCS)\ (f : relabelling).\ (P + Q)\,[f] = P\,[f] + Q\,[f]$$
$$\vdash \forall (P : CCS)\ (l : label).$$
$$(\tau.P) \setminus l = \tau.(P \setminus l) \wedge$$
$$\forall l_1 : label.\ (l_1.P) \setminus l = ((l_1 = l) \vee (l_1 = \overline{l})) \Rightarrow \mathsf{nil} \mid l_1.(P \setminus l)$$

The above formalization demonstrates the suitability of HOL for supporting embedded notations, the axioms being very similar to their conventional presentation. On the other hand, more work than is originally expected can be involved when mechanizing some definitions or axioms, e.g. the expansion law for parallel composition and the unfolding law for recursive expressions [3], because axioms written by hand are often packed with notation which itself needs to be formalized.

4 Verification of a Simple Buffer by Induction

Below we illustrate how the formalization of CCS described in the preceding section is used to reason about CCS specifications by presenting transcripts of a HOL session. In particular, we consider inductive reasoning and apply mathematical induction to prove the correctness of an implementation of a simple buffer [13].

The behaviour $Buffer_n$ of a buffer of capacity n can be simply specified as follows:

$$\begin{array}{lcll} Buffer_n(0) & \equiv & in\,.\,Buffer_n(1) & \\ Buffer_n(k) & \equiv & in\,.\,Buffer_n(k+1) + \overline{out}\,.\,Buffer_n(k-1) & (0 < k < n) \\ Buffer_n(n) & \equiv & \overline{out}\,.\,Buffer_n(n-1) & \end{array}$$

Such a specification is parameterized on the capacity n of the buffer and the number k of the values presently stored in the buffer. An implementation of the buffer can be built by composing in parallel n copies of a buffer cell

$$C \equiv \mathsf{rec}\,X.\,in\,.\,\overline{out}\,.\,X$$

and hiding the internally synchronizing actions *in* and *out* by using a new action *mid*, thus obtaining the chain $Impl(n)$ given by:

$$\begin{array}{lcl} Impl(1) & \equiv & C \\ Impl(n+1) & \equiv & C^\frown Impl(n) \end{array}$$

where, given two arbitrary processes P and Q, $^\frown$ is a linking operator defined as follows:

$$P^\frown Q \equiv (P\,[mid/out] \mid Q\,[mid/in]) \setminus mid$$

To show that $Impl(n)$ is a correct implementation of the buffer $Buffer_n$, we shall prove that for all $n \geq 1$

$$Impl(n) = Buffer_n(0)$$

where $=$ stands for the observational congruence. The proof is by induction on n, and in the proof of the inductive step, a lemma is needed which is itself proved by induction.

4.1 Mechanizing the proof in HOL

One interacts with the HOL system via ML. The ML prompt is #, so lines beginning with # show the user's input (always terminated by two successive semi-colons), and other lines show the system's response. Terms in the HOL logic are distinguished from ML expressions by enclosing them in double quotes. To help readability, the HOL transcripts are edited to show proper logical symbols instead of their ascii representations.

After having entered a theory in which we reason about the buffer, and declared the mechanized theory for CCS described earlier as a parent of this theory, we define the behaviour of a buffer cell and the linking operator. Throughout the proof, a buffer cell will be considered in its two possible states: as an empty cell C and as a full cell C'.

```
#new_definition ('C', "C = rec X. 'in'.-'out'.X");;
⊢ C = rec X. 'in'.-'out'.X

#new_definition ('C'', "C' = rec X. -'out'.'in'.X");;
⊢ C' = rec X. -'out'.'in'.X

#new_infix_definition
  ('Link', "∀ P Q:CCS. P Link Q = (P['mid'/'out'] | Q['mid'/'in'])\'mid'");;
⊢ ∀P Q.
  P Link Q = (P['mid'/'out'] | Q['mid'/'in'])\'mid'
```

The specification of the buffer is not primitive recursive, but it can be defined by invoking the ML function new_constant to introduce a function BUFF_SPEC and then by using the ML function new_axiom to assert the properties of such a BUFF_SPEC.

```
#new_constant ('BUFF_SPEC', ":num → num → CCS");;
() : void

#new_axiom ('BUFF_SPEC',
            "((0 < n) ⇒
              (BUFF_SPEC n 0 = 'in'.(BUFF_SPEC n 1)) ∧
               ((0 < k) ∧ (k < n) ⇒
               (BUFF_SPEC n k =
               'in'.(BUFF_SPEC n (SUC k)) + -'out'.(BUFF_SPEC n (PRE k)))) ∧
               (BUFF_SPEC n n = -'out'.(BUFF_SPEC n (PRE n))))");;
⊢ ∀n k.
    0 < n ⇒
    (BUFF_SPEC n 0 = 'in'.(BUFF_SPEC n 1)) ∧
    (0 < k ∧ k < n ⇒
     (BUFF_SPEC n k =
      'in'.(BUFF_SPEC n(SUC k)) + -'out'.(BUFF_SPEC n(PRE k)))) ∧
    (BUFF_SPEC n n = -'out'.(BUFF_SPEC n(PRE n)))
```

The implementation of the buffer is a primitive recursive definition starting from 1, thus we want to apply induction starting with 1. Since recursion and induction are defined on natural numbers in HOL, we must derive a recursive definition starting with 1 from that starting with 0. We first prove the existence of a recursive implementation IMPL0 starting with 0, and then prove that there exists a function fn satisfying the recursive definition

starting with 1. Finally, we give a name to fn by invoking the function **new_specification** which allows the new constant BUFF_IMPL to be introduced in a consistent way.

```
#new_prim_rec_definition
    ('IMPL0', "(IMPL0 0 = C) ∧ (IMPL0 (SUC n) = (C Link (IMPL0 n)))");;
⊢ (IMPL0 0 = C) ∧ (∀n. IMPL0(SUC n) = C Link (IMPL0 n))

#let IMPL1 = TAC_PROOF
             (([], "∃fn :num → CCS.
                      (fn 1 = C) ∧
                      (∀n. fn (SUC(SUC n)) = C Link (fn (SUC n)))"),
              STRIP_ASSUME_TAC IMPL0 THEN
              EXISTS_TAC "λn. IMPL0 (PRE n):CCS" THEN
              CONV_TAC (ONCE_DEPTH_CONV BETA_CONV) THEN
              ASM_REWRITE_TAC [PRE]);;
IMPL1 =
⊢ ∃fn. (fn 1 = C) ∧ (∀n. fn(SUC(SUC n)) = C Link (fn(SUC n)))

#new_specification 'BUFF_IMPL' [('constant','BUFF_IMPL')] IMPL1;;
⊢ (BUFF_IMPL 1 = C) ∧
  (∀n. BUFF_IMPL(SUC(SUC n)) = C Link (BUFF_IMPL(SUC n)))
```

To prove that the implementation meets the specification, we apply several tactics. Some of them are built-in and some have been implemented in the system specially for manipulating CCS specifications. The built-in tactic INDUCT_TAC applies induction on natural numbers and the induction assumption is indicated with set brackets.

```
#g "∀n. BUFF_IMPL(SUC n) = BUFF_SPEC(SUC n)0";;
"∀n. BUFF_IMPL(SUC n) = BUFF_SPEC(SUC n)0"

() : void

#e (INDUCT_TAC);;
OK..
2 subgoals
"BUFF_IMPL(SUC(SUC n)) = BUFF_SPEC(SUC(SUC n))0"
    [ "BUFF_IMPL(SUC n) = BUFF_SPEC(SUC n)0" ]

"BUFF_IMPL 1 = BUFF_SPEC 1 0"

() : void
```

To prove the basis subgoal, we expand with the definition of BUFF_IMPL and of C. Next, the resulting recursive expression is unfolded once, by means of the tactic REC_EXP_TAC derived from the unfolding law for recursion, and then the current goal is folded back by using the definition of C and the first clause of the definition of BUFF_IMPL.

```
#e (ONCE_REWRITE_TAC [BUFF_IMPL] THEN ONCE_REWRITE_TAC [C] THEN REC_EXP_TAC
    THEN ONCE_REWRITE_TAC [SYM C] THEN SUBST1_TAC (SYM IMPL_CLAUSE1));;
OK..
"'in'.-'out'.(BUFF_IMPL 1) = BUFF_SPEC 1 0"

() : void
```

Now we manipulate the specification of the buffer by expanding twice with the definition of BUFF_SPEC, each time selecting the right definition clause based on the value of k.

```
#e (ONCE_REWRITE_TAC [CONJUNCT1 SPEC_SUCO_SUCO] THEN
    ONCE_REWRITE_TAC [CONJUNCT2 SPEC_SUCO_SUCO]);;
OK..
"'in'.-'out'.(BUFF_IMPL 1) = 'in'.-'out'.(BUFF_SPEC 1 0)"

() : void
```

The next step is to check if BUFF_IMPL 1 and BUFF_SPEC 1 0 denote the (unique) solution of the same recursive equation. This can be achieved by applying the tactic UNIQUE_SOL_TAC that mechanizes the proof rule for the unique solution of recursive equations (Section 3).

```
#e (UNIQUE_SOL_TAC "BUFF_IMPL 1 :CCS" "'in'.-'out'.(BUFF_IMPL 1)"
                   "BUFF_SPEC 1 0 :CCS" "'in'.-'out'.(BUFF_SPEC 1 0)");;
OK..
goal proved
⊢ 'in'.-'out'.(BUFF_IMPL 1) = 'in'.-'out'.(BUFF_SPEC 1 0)
⊢ 'in'.-'out'.(BUFF_IMPL 1) = BUFF_SPEC 1 0
⊢ BUFF_IMPL 1 = BUFF_SPEC 1 0

Previous subproof:
"BUFF_IMPL(SUC(SUC n)) = BUFF_SPEC(SUC(SUC n))0"
    [ "BUFF_IMPL(SUC n) = BUFF_SPEC(SUC n)0" ]

() : void
```

Once the basis subgoal has been proved, the HOL system presents us with the induction step subgoal. Note that, since we started the proof by induction from 1, the inductive hypothesis holds for $n+1$ and we prove the induction step for $n+2$. We expand with the definition of BUFF_IMPL and of the linking operator, and we then apply the inductive hypothesis by rewriting with the equation in the assumption list of the goal.

```
#e (ONCE_REWRITE_TAC [BUFF_IMPL] THEN ONCE_REWRITE_TAC [Link] THEN
    ONCE_ASM_REWRITE_TAC []);;
OK..
"((C['mid'/'out']) | ((BUFF_SPEC(SUC n)0)['mid'/'in']))\'mid' =
 BUFF_SPEC(SUC(SUC n))0"
    [ "BUFF_IMPL(SUC n) = BUFF_SPEC(SUC n)0" ]

() : void
```

At this point, the goal will be proved if we show that the two sides of the above equivalence denote the (unique) solution of the same recursive expression. This means that we have to prove that the defining equations of $Buffer_{n+2}$ are satisfied when replacing

$Buffer_{n+2}(k)$	by	$C^\frown Buffer_{n+1}(k)$	$(0 \leq k \leq n+1)$
$Buffer_{n+2}(n+2)$	by	$C'^\frown Buffer_{n+1}(n+1)$	$(k = n+2)$

By case analysis on k, this requires one to prove the following observational congruences:

$$\begin{array}{lcll}
C^\frown Buffer_{n+1}(0) & = & in.(C^\frown Buffer_{n+1}(1)) & \\
C^\frown Buffer_{n+1}(k) & = & in.(C^\frown Buffer_{n+1}(k+1)) & (0 < k < n+1) \\
 & & + \overline{out}.(C^\frown Buffer_{n+1}(k-1)) & \\
C^\frown Buffer_{n+1}(n+1) & = & in.(C'^\frown Buffer_{n+1}(n+1)) & (k = n+1) \\
 & & + \overline{out}.(C^\frown Buffer_{n+1}(n)) & \\
C'^\frown Buffer_{n+1}(n+1) & = & \overline{out}.(C^\frown Buffer_{n+1}(n+1)) &
\end{array}$$

These congruences can be proved by rewriting each left-hand side with the definitions of the processes occurring in it and applying the axioms for relabelling, restriction and parallel composition operators, until a suitable form is reached and the key lemma of the whole proof can be applied. This lemma is the following:

$$C'^\frown Buffer_n(k) \;=\; \tau.(C^\frown Buffer_n(k+1)) \qquad (0 \leq k < n)$$

The specification $Buffer_n(k)$ can be expressed as the linking of k full buffer cells C' and $(n-k)$ empty cells C. When an empty cell inputs a value and becomes a full cell, then its value can percolate to the right by a sequence of internal actions, thus obtaining $Buffer_n(k+1)$.

The above lemma is proved by induction on k, by applying the usual rewriting strategy. Below, we present the mechanization of this proof. To help readability, the ML code for the tactic that proves the lemma has been replaced by an informal English description.

```
#let LEMMA =
   TAC_PROOF
    (([], "∀k n :num.
          ((0 < n) ∧ (k < n)) ⇒
           (C' Link (BUFF_SPEC n k) = tau.(C Link (BUFF_SPEC n (SUC k))))"),
      Rewrite using the definition of the linking operator
      THEN Apply mathematical induction on the variable k
      THENL [Strip off the universally quantified variable n and
             move the antecedent of implication to assumption list
             THEN Use the theorem FULL_TO_EMPTY_CELL
             THEN Rewrite using the definition of BUFF_SPEC
             THEN Apply axioms for relabelling, parallel and restriction;
             Strip off the universally quantified variable n and move
             conjuncts of antecedent of implication to assumption list
             THEN Use the theorem FULL_TO_EMPTY_CELL
             THEN Rewrite using the definition of BUFF_SPEC
             THEN Apply axioms for relabelling, parallel and restriction
             THEN Use the theorem TRANSF_FULL_CELL
             THEN Apply the inductive hypothesis
             THEN Apply the τ-law μ.τ.E = μ.E
             THEN Use the theorem EXP_ABS_THM
             THEN Apply TAU_STRAT]);;
LEMMA =
⊢ ∀k n.
    0 < n ∧ k < n ⇒
    (C' Link (BUFF_SPEC n k) = tau.(C Link (BUFF_SPEC n(SUC k))))
```

In this proof, various tactics and theorems, e.g. FULL_TO_EMPTY_CELL, TRANSF_FULL_CELL and EXP_ABS_THM, are used which we have previously defined and proved in HOL, to manipulate subexpressions of the goal and make the application of some axioms concerning the action τ possible. Moreover, we use the rewriting strategy TAU_STRAT which implements a term rewriting system equivalent to the axiomatization of observational congruence for finite CCS, [9], and which has been mechanized in HOL, [3]. In the proof of the lemma, this strategy applies the derived τ-law, $E + \tau.(F + E) = \tau.(F + E)$.

The above congruences can now be proved, but we do not present the proofs here. Actually, only the second congruence needs the application of the lemma; the remaining ones may also be proved by the usual rewriting strategy.

5 Conclusion

We have presented an attempt to use a verification environment based on the HOL theorem prover for reasoning about CCS specifications. In particular, we have considered inductively defined systems, and we have described how a higher level verification proof, such as a proof by induction, can be mechanized in such an environment.

We believe that this attempt demonstrates evidence that mathematical proof techniques - based on the axiomatic representation of process algebras - provide a promising approach to the mechanical verification of concurrent systems. Works on a similar approach include an investigation into mechanizing CCS using NUPRL [4], a formalization of CSP failure-divergence semantics in HOL [2], an ongoing mechanization of Milner's π-calculus in HOL [12], and the development of a process algebra manipulator [10]. With respect to these works, we are more interested in the formalization of higher level verification strategies in a theorem proving framework, where mathematical proof techniques are naturally available and the facility for defining user's verification strategies is provided.

In this paper, we have focussed attention on a subset of CCS, on the theory of observational congruence, and on the facility that HOL provides for performing proofs by induction, thus making it possible to reason about parameterized or indexed specifications. It is also possible, due to the facilities for modularity in HOL, to mechanize different process algebras, various behavioural semantics for the same process algebra, and to derive axiomatizations and proof tools for them.

Current work concerns the mechanization of the operational semantics of CCS, the definition of observational congruence, and the mechanical derivation of its axiomatization. Future work will mainly be devoted to extend the functionality of the HOL-CCS environment, e.g. enriching the language under consideration by incorporating data with processes (*value passing*).

Acknowledgements

I should like to thank several members of the Cambridge hardware verification group for many useful discussions. I am especially grateful to Mike Gordon, Tom Melham, Sara Kalvala, Brian Graham, John Van Tassel, John Harrison and Richard Boulton for their advice on mechanization in HOL. Thanks are also due to Paola Inverardi (I.E.I., Pisa) and Albert Camilleri (Hewlett-Packard Labs, Bristol) for their continued support.

References

[1] Camilleri A. J., 'Mechanizing CSP Trace Theory in Higher Order Logic', *IEEE Transactions on Software Engineering*, Special Issue on Formal Methods, N. G. Leveson (ed.), September 1990, Vol. 16, No. 9, pp. 993-1004.

[2] Camilleri A. J., 'A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics', Proc. of the *4th Banff Higher Order Workshop*, G. Birtwistle (ed.), Springer-Verlag, London, 1991, (to appear).

[3] Camilleri A. J., Inverardi P., Nesi M., 'Combining Interaction and Automation in Process Algebra Verification', Proc. of TAPSOFT '91, Lecture Notes in Computer Science, Springer-Verlag, 1991, Vol. 494, pp. 283–296.

[4] Cleaveland R., Panangaden P., 'Type Theory and Concurrency', *International Journal of Parallel Programming*, November 1988, Vol. 12, No. 2, pp. 153–206.

[5] Cousineau G., Huet G., Paulson L., 'The ML Handbook', INRIA, 1986.

[6] De Nicola R., Inverardi P., Nesi M., 'Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Specifications', Proc. of the Workshop on *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, Springer-Verlag, 1990, Vol. 407, pp. 54–67.

[7] Gordon M. J. C., 'HOL—A Proof Generating System for Higher-Order Logic', *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. Subrahmanyam (eds.), Kluwer Academic Publishers, Boston, 1988, pp. 73–128.

[8] Gordon M. J. C., 'Mechanizing Programming Logics in Higher Order Logic', *Current Trends in Hardware Verification and Automated Theorem Proving*, G. Birtwistle and P. Subrahmanyam (eds.), Springer-Verlag, 1989, pp. 387–439.

[9] Inverardi P., Nesi M., 'A Rewriting Strategy to Verify Observational Congruence', *Information Processing Letters*, 1990, Vol. 35, pp. 191-199.

[10] Lin H., 'PAM: A Process Algebra Manipulator', this volume.

[11] Melham T. F., 'Automating Recursive Type Definitions in Higher Order Logic', *Current Trends in Hardware Verification and Automated Theorem Proving*, G. Birtwistle and P. Subrahmanyam (eds.), Springer-Verlag, 1989, pp. 341–386.

[12] Melham T. F., 'A Mechanized Theory of the π-calculus in HOL', Proc. of the *2nd Annual Esprit BRA Workshop on Logical Frameworks*, Univ. of Edinburgh, 1991.

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

[14] Paulson L. C., *Logic and Computation—Interactive Proof with Cambridge* LCF, Cambridge Tracts in Theoretical Computer Science (2), Cambridge Univ. Press, 1987.

[15] Proc. of the Workshop on *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, Springer-Verlag, 1990, Vol. 407.