A Formalization of the Process Algebra CCS in Higher Order Logic

Monica Nesi †

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, England

Technical Report 278
December 1992

Abstract: *This paper describes a mechanization in higher order logic of the theory for a subset of Milner's* CCS*. The aim is to build a sound and effective tool to support verification and reasoning about process algebra specifications. To achieve this goal, the formal theory for pure* CCS *(no value passing) is defined in the interactive theorem prover* HOL*, and a set of proof tools, based on the algebraic presentation of* CCS*, is provided.*

†Research supported by Consiglio Nazionale delle Ricerche (C.N.R.), Italy.

Contents

1 Introduction

Process algebras [2, 3, 22, 29] are generally recognized to be a convenient tool for specifying concurrent systems at different levels of abstraction. These formalisms are usually equipped with one or more notions of behavioural semantics and with modal/temporal logics, which can be used to reason about process algebra specifications. On one hand, behavioural equivalences and preorders allow one to study the relationships between different descriptions of the same system by verifying if they are equivalent when "uninteresting" details are ignored, or if a low level description is a "satisfactory" implementation of a more abstract one. On the other hand, logics for process algebras can be used to check if a specification has a given modal/temporal property.

The operational semantics of a process algebra is defined via labelled transition systems, and then behavioural relations and modal/temporal logics can be defined and interpreted in terms of transition systems. Behavioural equivalences/preorders have also been characterized through sets of (in)equational laws, which can be used to manipulate specifications and reason about them. In the literature, sets of laws have been proved to characterize various behavioural equivalences/preorders over subsets of several process algebras in a correct and complete way [5, 14, 17, 21, 29].

In this paper we use higher order logic and the general purpose theorem prover HOL [15] to develop an interactive verification environment for the CCS process algebra [29]. The aim is to build a verification system based on theorem proving which is logically sound, i.e. it is built up by following a purely definitional approach, in order to avoid introducing inconsistencies in the logic being used. In the HOL system, this can be achieved by means of primitive definition mechanisms for introducing new entities in a sound way, and by deriving all other kinds of definitions by formal proof.

This methodology also allows users to take advantage of all components of the formal theory for process algebras, such as the labelled transitions, the operational and axiomatic characterizations of behavioural semantics, and modal/temporal logics, in a unified framework, and to define their own verification strategies. In this work the HOL theorem prover is mainly used as a suitable framework for building a practical and sound proof-assistant tool for applications specified in a given process calculus. But HOL also allows one to perform metatheoretic reasoning about the process calculus itself, and can help investigate variants of a given process calculus and develop the formal theory for new calculi.

Our approach is based on the algebraic nature of the CCS language and on the axiomatic characterization of behavioural semantics. The formal theory for *observational congruence* [29] and for a slight extension of *Hennessy-Milner modal logic* [21, 35] over *pure* CCS (no value passing) is embedded in the HOL logic, and the resulting formalization supports verification strategies based on mechanized formal proof. These include strategies that exhibit different degrees of user interaction depending on the subsets of CCS under consideration [10], proofs of correctness by mathematical induction for parameterized specifications [30], and verification of modal properties [31]. 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. Then, the syntax and the operational semantics of pure CCS are introduced, followed by the theory of the observational semantics and its axiomatic characterization, and the modal logic. Next, we show how all these definitions can be formalized in HOL, and how verification can be performed in the resulting framework by presenting the HOL proofs of the correctness of a buffer, and of a modal property for a simple process, such as a vending machine. 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 [20] and process algebras, e.g. CSP [8, 9] and π-calculus [27]. The HOL theorem prover [15], developed by Gordon [19] and used in these mechanizations, is based on the LCF methodology [33] for interactive and secure theorem proving by mechanizing the logic in the strongly-typed programming language ML [13].

The HOL logic is a variety of higher order logic based on Church's formulation of type theory [11]. The standard predicate calculus is extended in the HOL logic by allowing variables to range over functions. The arguments of functions can themselves be functions. Moreover, functions can be written as λ-abstractions, terms can be polymorphic, and Hilbert's choice operator, ε, is included in the logic.

The ML language is used to manipulate HOL logic terms. In particular, ML is used to prove that certain terms are theorems. A theorem is represented by a finite set of terms called *assumptions* and a term called *conclusion*. Given a set of assumptions Γ and a conclusion t, we write $\Gamma \vdash t$ to represent the corresponding theorem or, if Γ is empty, $\vdash t$. To introduce theorems into ML, 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*. The HOL logic includes primitive rules of definition for extending the logic in a consistent way. These primitive rules are of very restricted forms, and this means that all other kinds of definitions must be derived from the primitive ones by formal proof. This can sometimes lead to rather complex formalizations. However, several derived rules of definition have been mechanized in HOL and are supported in a fully automatic way. These rules include recursive concrete type definitions, primitive recursive function definitions over these types, and certain forms of inductive definition, all developed by Melham [26, 28].

The derived rule of *recursive type definition* allows one to define arbitrary concrete recursive types in terms of their constructors [26]. The input to this definition mechanism is a specification of the syntax of the operators written in terms of existing types and recursive calls to the type being defined. Then, the system performs all the formal inference necessary to define the type in higher order logic and derives an abstract characterization of the newly-defined type in a fully automatic way.

The derived mechanism of *primitive recursive function definition* automates existence proofs for primitive recursive functions defined over concrete recursive types. The system proves the existence of a total function satisfying the recursive defining equations, and then a *constant specification* introduces a new constant to denote such a total function. A derived inference rule is also provided to prove a structural induction theorem for any concrete recursive type.

The derived rule for *inductive definitions* allows one to define relations which are inductively defined by a set of rules [28]. Any such relation is simply defined as the intersection of all relations closed under that set of rules. The system automatically proves that the resulting relation is itself closed under the set of rules and is the least such relation. The theorems resulting from such a definition mechanism constitute a complete characterization of the defining properties of the newly-defined relation. They also include a principle of "rule induction", which allows proofs by induction to be performed over the structure of the derivations defined by the set of rules.

A collection of logical types, type operators, constants, definitions, axioms and theorems is called a *theory*. Theories enable a hierarchical organization of facts, i.e. if facts from other theories are to be used, the relevant existing theories must be declared as *parents*.

To prove a theorem in a theory, one must apply a sequence of steps (constituting a proof) to either axioms or previously proved theorems by using inference rules (forward proof). The HOL system supports another way of carrying out a proof, called goal directed proof or backward proof. The idea is to do the proof starting from the desired result (*goal*) and manipulating 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 [33].

As far as goal directed proofs are concerned, the HOL system provides a *subgoal package* due to Paulson [33], which implements a simple framework for interactive proofs. A goal given by an assumption list Γ and a term t, written $\Gamma \mathrel{?\!\!\vdash} t$ (if Γ is empty, we write $\mathrel{?\!\!\vdash} t$), can be set by invoking either the function `set_goal` or the function `g`, which initialize 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, 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 standard 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; or one can use the function `prove_thm` which takes a string s, a boolean term t and a tactic tac, and attempts to prove the goal $\mathrel{?\!\!\vdash} t$ by applying tac. If it succeeds, the resulting theorem is saved under the name s in the current theory.

The HOL system also provides functions called *conversions* [32], 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, have played a fundamental role in our mechanization of CCS.

3 CCS

In this section we recall only the essential information about CCS (*Calculus of Communicating Systems*) and refer to [29] for more details about the calculus.

3.1 Syntax and Operational Semantics

We consider *pure* CCS, a subset of the language which does not involve value passing and consists of the inactive agent nil, and the following operations on agent expressions: *prefix* (.), *summation* (+), *restriction* (\), *relabelling* ([ ]), *parallel composition* (|) and *recursion* (rec). The syntax of pure CCS expressions, ranged over by $E, F, E', E1, \ldots$ is as follows:

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

where L is a subset of visible actions, called labels (ranged over by l), u ranges over actions, which are either labels or the invisible action τ, and X ranges over agent variables (which will be bound in recursive definitions). Labels consist of names and co-names where, for any name a, the corresponding co-name is written $\overline{a}$. This complement operation has the property that $\overline{\overline{l}} = l$. A relabelling function f is a function from labels to labels such that relabelling co-names has the property that $f(\overline{l}) = \overline{f(l)}$. A relabelling function f is then extended to actions by defining $f(\tau) = \tau$.

The expression nil represents an agent which cannot perform any action. The agent $u.E$ can only perform the action u and then behaves like E. The agent $E_1 + E_2$ behaves like either E_1 or E_2. The agent $E \setminus L$ behaves like E but cannot perform an action u if either u or $\overline{u}$ is in L. The actions of $E\,[f]$ are renamings of those of the agent E via the relabelling function f. The agent $E_1 \,|\, E_2$ can perform the actions of E_1 and E_2 in parallel; moreover, the agents E_1 and E_2 can synchronize through the action τ whenever they are able to perform complementary actions. The expression $\mathsf{rec}\, X.\, E$ denotes a recursive agent.

The operational semantics of the above CCS operators is given via a transitional semantics based on *labelled interleaving transitions* $\xrightarrow{u}$ over CCS expressions. The transition relation $E \xrightarrow{u} F$ is inductively defined by the following rules:

$$\text{PREFIX:} \quad \frac{}{u\,.E \xrightarrow{u} E}$$

$$\text{SUM1:} \quad \frac{E \xrightarrow{u} E'}{E{+}F \xrightarrow{u} E'} \qquad \text{SUM2:} \quad \frac{E \xrightarrow{u} E'}{F{+}E \xrightarrow{u} E'}$$

$$\text{RESTR:} \quad \frac{E \xrightarrow{u} E'}{E\backslash L \xrightarrow{u} E'\backslash L} \qquad u, \overline{u} \notin L$$

$$\text{RELAB:} \quad \frac{E \xrightarrow{u} E'}{E\,[f] \xrightarrow{f(u)} E'\,[f]}$$

$$\text{PAR1:} \quad \frac{E \xrightarrow{u} E'}{E|F \xrightarrow{u} E'|F} \qquad \text{PAR2:} \quad \frac{E \xrightarrow{u} E'}{F|E \xrightarrow{u} F|E'}$$

$$\text{PAR3:} \quad \frac{E \xrightarrow{l} E' \;\; F \xrightarrow{\overline{l}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

$$\text{REC:} \quad \frac{E\{\mathsf{rec}\,X.\,E/X\} \xrightarrow{u} E'}{\mathsf{rec}\,X.\,E \xrightarrow{u} E'}$$

where the notation $E\{\mathsf{rec}\,X.\,E/X\}$ denotes the substitution of $\mathsf{rec}\,X.\,E$ for all free occurrences of X in the expression E.

3.2 Observational Semantics

In the literature several behavioural semantics have been defined for CCS, such as *strong* and *observational congruences* [29], *trace* and *testing equivalences* [14], and *branching bisimulation* [17]. Each of these semantics has been characterized in terms of axiomatizations, which have been proved sound and complete for subsets of CCS expressions.

The distinction between the various behavioural semantics lies in the notion of *behaviour* and in the way the silent action τ is dealt with. In what follows, we address observational congruence, and recall the relevant definitions below.

The *weak transition* relation $\stackrel{s}{\Longrightarrow}$ for any sequence s of actions is first defined. Given agent expressions E, F and a sequence of actions $s = u_1 \ldots u_n$ ($n \geq 0$), then $E \stackrel{s}{\Longrightarrow} F$ if $E\,(\xrightarrow{\tau})^* \xrightarrow{u_1} (\xrightarrow{\tau})^* \ldots \xrightarrow{u_n} (\xrightarrow{\tau})^*\, F$, where $(\xrightarrow{\tau})^*$ denotes the reflexive-transitive closure of the transition relation $\xrightarrow{\tau}$. If $s = \epsilon$ (empty sequence), then $E \stackrel{\epsilon}{\Longrightarrow} F$ if and only if $E\,(\xrightarrow{\tau})^*\, F$.

The observational equivalence is defined in terms of a bisimulation relation between agents. A binary relation S over agent expressions is a *weak bisimulation* if for all E, F, $E\; S\; F$ implies that

(a) for all $u \neq \tau$
   (i) whenever $E \xrightarrow{u} E'$, then for some F', $F \stackrel{u}{\Longrightarrow} F'$ and $E'\; S\; F'$;
   (ii) whenever $F \xrightarrow{u} F'$, then for some E', $E \stackrel{u}{\Longrightarrow} E'$ and $E'\; S\; F'$;
(b)
   (i) whenever $E \xrightarrow{\tau} E'$, then for some F', $F \stackrel{\epsilon}{\Longrightarrow} F'$ and $E'\; S\; F'$;
   (ii) whenever $F \xrightarrow{\tau} F'$, then for some E', $E \stackrel{\epsilon}{\Longrightarrow} E'$ and $E'\; S\; F'$.

E and F are defined to be *observational equivalent*, $E \approx F$, if and only if $E\; S\; F$ for some weak bisimulation S.

This means that, in order to prove the observational equivalence of two agents E and F, it is sufficient to show that there exists a weak bisimulation which contains the pair (E, F). An alternative approach is to use equational reasoning: given a collection of basic equivalences which are known to hold of observational semantics, $E \approx F$ can be proved by applying them using the principle of "substituting equals for equals". This is possible if the behavioural equivalence is a *congruence*, but observational equivalence turns out not to be a congruence relation. A congruence over CCS expressions is a relation which is preserved by all CCS operators, and observational equivalence is not preserved by summation contexts. However, observational equivalence can be refined to a congruence relation by defining that two agent expressions E and F are *observational congruent*, $E = F$, if for all actions u

(i) whenever $E \xrightarrow{u} E'$, then for some F', $F \overset{u}{\Longrightarrow} F'$ and $E' \approx F'$;
(ii)whenever $F \xrightarrow{u} F'$, then for some E', $E \overset{u}{\Longrightarrow} E'$ and $E' \approx F'$.

This means that every first action (τ included) of an agent must be matched by an equal action of the other agent (plus some silent actions, possibly) and vice versa. Observational congruence is preserved by all CCS operators and its axiomatic presentation allows us to perform equational reasoning on CCS expressions by substituting equals for equals.

3.3 Axiomatic Characterization of Observational Congruence

The axiomatic presentations which characterize the behavioural equivalences for CCS can be separated into two sets of axioms: (1) those common to all equivalences, referred to as *basic axioms*, and (2) those concerning the silent action τ which distinguish the various equivalences, referred to as *τ-laws*.

By means of the basic axioms, any finite (i.e. without recursion) CCS expression can be proved equivalent to one containing only nil, prefix and summation operators. The basic axioms are shown below:

$$E + (F + G) = (E + F) + G \tag{A1}$$
$$E + F = F + E \tag{A2}$$
$$E + E = E \tag{A3}$$
$$E + \mathsf{nil} = E \tag{A4}$$

$$\mathsf{nil}\,[f] = \mathsf{nil} \tag{A5}$$
$$(E + F)\,[f] = E\,[f] + F\,[f] \tag{A6}$$
$$(u\,.\,E)\,[f] = f(u)\,.\,E\,[f] \tag{A7}$$

$$\mathsf{nil} \setminus L = \mathsf{nil} \tag{A8}$$
$$(E + F) \setminus L = (E \setminus L) + (F \setminus L) \tag{A9}$$
$$(u\,.\,E) \setminus L = u\,.\,(E \setminus L) \qquad \text{if } u, \overline{u} \notin L,\ \mathsf{nil} \text{ otherwise} \tag{A10}$$

$$\text{If } E = \sum_{i \geq 0} u_i.E_i \text{ and } F = \sum_{j \geq 0} v_j.F_j \text{ then} \tag{A11}$$

$$E \mid F = \sum_{i \geq 0} u_i.(E_i \mid F) + \sum_{j \geq 0} v_j.(E \mid F_j) + \sum \{\tau.(E_i \mid F_j) \mid u_i = \overline{v_j}\}$$

$$\mathsf{rec}\,X.\,E = E\{\mathsf{rec}\,X.\,E/X\} \tag{A12}$$

The τ-laws for the observational congruence are the following:

$$u\,.\,\tau\,.\,E = u\,.\,E \tag{T1}$$
$$E\ +\ \tau\,.\,E = \tau\,.\,E \tag{T2}$$
$$u\,.\,(E\ +\ \tau\,.\,F) + u\,.\,F = u\,.\,(E\ +\ \tau\,.\,F) \tag{T3}$$

The theory of observational congruence for finite CCS is characterized by the axioms (A1)–(A11) and (T1)–(T3). These axioms have been proved correct and complete with respect to bisimulation in [21].

Finally, we recall a result which will be used in the correctness proof in Section 6. When dealing with recursive equations, two agents 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 [29].

3.4 A Modal Logic

The modal logic under consideration is a slight extension of Hennessy-Milner logic [21] presented in [35]. Its formulas are defined by the following abstract syntax:

$$\Phi ::= \mathtt{tt} \mid \neg\Phi \mid \Phi \wedge \Phi \mid [A]\Phi$$

where A ranges over sets of actions. The meaning of the first three formulas is familiar. The modalized formula $[A]\Phi$, where the modal operator $[A]$ is sometimes referred to as *box*, means that Φ holds after every performance of any action in A.

For any formula Φ of the logic we define when an agent expression E has (or "satisfies") the property Φ. We write $E \models \Phi$ to mean E satisfies Φ, and $E \not\models \Phi$ to mean E fails to have the property Φ. The *satisfaction* relation $\models$ is inductively defined on the structure of formulas:

$$\begin{array}{lll} E \models \mathtt{tt} & & \\ E \models \neg\Phi & \textit{iff} & E \not\models \Phi \\ E \models \Phi_1 \wedge \Phi_2 & \textit{iff} & E \models \Phi_1 \textit{ and } E \models \Phi_2 \\ E \models [A]\Phi & \textit{iff} & \forall E'. \forall u \in A. \textit{ if } E \xrightarrow{u} E' \textit{ then } E' \models \Phi \end{array}$$

Every agent has the property $\mathtt{tt}$. An agent has the property $\neg\Phi$ when it fails to satisfy the property Φ, and it has the property $\Phi_1 \wedge \Phi_2$ when it has both properties Φ_1 and Φ_2. Finally, an agent satisfies $[A]\Phi$ if after every performance of any action in A all the resulting agents have the property Φ.

Derived operators, including the dual $\langle A\rangle$ (sometimes referred to as *diamond*) of $[A]$, are defined as follows:

$$\begin{array}{lll} \mathtt{ff} & \stackrel{\text{def}}{=} & \neg\mathtt{tt} \\ \Phi_1 \vee \Phi_2 & \stackrel{\text{def}}{=} & \neg(\neg\Phi_1 \wedge \neg\Phi_2) \\ \langle A\rangle\Phi & \stackrel{\text{def}}{=} & \neg[A]\neg\Phi \end{array}$$

The intended meaning of the diamond operator $\langle A\rangle$ is the following:

$$E \models \langle A\rangle\Phi \quad \textit{iff} \quad \exists E'. \exists u \in A. \; E \xrightarrow{u} E' \textit{ and } E' \models \Phi$$

This logic is a slight extension of Hennessy-Milner logic [21] in the sense that modalities are indexed by a set of actions instead of a single action. Properties such as *capacity* and *necessity* can be expressed within this logic. The modal formula $\langle A\rangle\mathtt{tt}$ expresses a capacity to perform an action in A, since:

$$E \models \langle A\rangle\mathtt{tt} \quad \textit{iff} \quad \exists E'. \exists u \in A. \; E \xrightarrow{u} E'$$

and $\langle A\rangle\mathtt{ff}$ expresses an inability to perform any action in A. Using the notation in which, given a set of actions Act, $[-]$ stands for $[Act]$ and $[-u]$ for $[Act - \{u\}]$ (and analogously for the diamond operator), the property that an agent expression E must perform a given action a (necessity) can be expressed as follows:

$$E \models \langle -\rangle\mathtt{tt} \wedge [-a]\mathtt{ff}$$

where the formula $\langle -\rangle\mathtt{tt}$ states that some action can be performed, and $[-a]\mathtt{ff}$ expresses that every action but a is impossible.

4 Mechanization of CCS in HOL

4.1 The Syntax

In this section we describe the formalization in HOL of the CCS syntax presented earlier. We begin with the mechanization of labels and actions by defining concrete data types using the derived principle for (recursive) type definitions [26]. The syntactic types *label* and *action* can be defined as follows:

$$label = \mathsf{name}\; string \mid \mathsf{coname}\; string$$

$$action = \mathsf{tau} \mid \mathsf{label}\; label$$

where name, coname, tau and label are distinct constructors. Given the above specifications, the derived rule for (recursive) type definition automatically derives a theorem of higher order logic for each type being defined, which characterizes the type in a complete and abstract way. These theorems for the types *label* and *action* are the following:

$$\vdash \forall f0\; f1.\; \exists!\, fn.\; (\forall s.\; fn\,(\mathsf{name}\; s) = f0\; s) \;\wedge\; (\forall s.\; fn\,(\mathsf{coname}\; s) = f1\; s)$$

$$\vdash \forall e\; f.\; \exists!\, fn.\; (fn\; \mathsf{tau} = e) \;\wedge\; (\forall l.\; fn\,(\mathsf{label}\; l) = f\; l)$$

They assert the admissibility of defining functions over labels and actions by primitive recursion. Structural induction theorems for both types are provided as well.

The notion of complement can be defined by a function over the type *label* as follows:

$$\forall s.\; \mathsf{Compl}\,(\mathsf{name}\; s) = \mathsf{coname}\; s \;\wedge\; \forall s.\; \mathsf{Compl}\,(\mathsf{coname}\; s) = \mathsf{name}\; s$$

and then extended to actions with the following definition:

$$\forall l.\; \mathsf{Compl\_Act}\,(\mathsf{label}\; l) = \mathsf{label}\,(\mathsf{Compl}\; l)$$

Using case analysis on the type *label*, we can then prove that $\bar{\bar{l}} = l$ for all l, thus obtaining the following theorem:

$$\vdash\; \forall l.\; \mathsf{Compl}\,(\mathsf{Compl}\; l) \;=\; l$$

The type *relabelling* for relabelling functions is defined as the set of functions of type $action \rightarrow action$ such that relabelling respects complements and τ is renamed as τ.

The type *CCS* of CCS agent expressions can now be defined by means of the derived HOL rule for automatically defining concrete recursive types as follows:

$$\begin{aligned} CCS = \;& \mathsf{nil} \mid \\ & \mathsf{var}\; string \mid \\ & \mathsf{prefix}\; action\; CCS \mid \\ & \mathsf{sum}\; CCS\; CCS \mid \\ & \mathsf{restr}\; CCS\; (label)set \mid \\ & \mathsf{relab}\; CCS\; relabelling \mid \\ & \mathsf{par}\; CCS\; CCS \mid \\ & \mathsf{rec}\; string\; CCS \end{aligned}$$

where nil, var, prefix, sum, restr, relab, par and rec are distinct constructors. Similarly to the types *label* and *action*, a complete characterization of the type *CCS* is automatically derived:

$$
\begin{array}{l}
\vdash \forall e\ f0\ f1\ f2\ f3\ f4\ f5\ f6. \\
\quad \exists!\ fn. \\
\quad\ (fn\ \mathsf{nil}\ =\ e) \wedge \\
\quad\ (\forall A\ C.\ fn(\mathsf{prefix}\ A\ C)\ =\ f0\ (fn\ C)\ A\ C) \wedge \\
\quad\ (\forall C1\ C2.\ fn(\mathsf{sum}\ C1\ C2)\ =\ f1\ (fn\ C1)\ (fn\ C2)\ C1\ C2) \wedge \\
\quad\ (\forall C\ s.\ fn(\mathsf{restr}\ C\ s)\ =\ f2\ (fn\ C)\ s\ C) \wedge \\
\quad\ (\forall C\ R.\ fn(\mathsf{relab}\ C\ R)\ =\ f3\ (fn\ C)\ R\ C) \wedge \\
\quad\ (\forall C1\ C2.\ fn(\mathsf{par}\ C1\ C2)\ =\ f4\ (fn\ C1)\ (fn\ C2)\ C1\ C2) \wedge \\
\quad\ (\forall s.\ fn(\mathsf{var}\ s)\ =\ f5\ s) \wedge \\
\quad\ (\forall s\ C.\ fn(\mathsf{rec}\ s\ C)\ =\ f6\ (fn\ C)\ s\ C)
\end{array}
$$

and this theorem of higher order logic is the basis for reasoning about the type *CCS*.

4.2 The Operational Semantics

The next step in our formalization is the definition of the labelled transition relation which gives the operational meaning of the CCS operators. This relation can be embedded in HOL by using the derived principle for inductively defined relations [28]. The transition relation $E \xrightarrow{u} E'$ is represented by $\mathsf{Trans}\ E\ u\ E'$, where the relation

$$\mathsf{Trans} : CCS \rightarrow action \rightarrow CCS \rightarrow bool$$

is defined as the intersection of all relations that satisfy the rules of the operational semantics. The mechanism for inductive definitions proves that this intersection is closed under the transition rules and is the least such relation. Proving that the relation Trans satisfies the transition rules results in the following list of theorems, which state the labelled transition rules given in Section 3.1:

$$
\begin{array}{ll}
\text{PREFIX:} & \vdash \forall u\ E.\ \mathsf{Trans}\ (\mathsf{prefix}\ u\ E)\ u\ E \\
\text{SUM1:} & \vdash \forall E\ u\ E1.\ \mathsf{Trans}\ E\ u\ E1\ \supset (\forall E'.\ \mathsf{Trans}\ (\mathsf{sum}\ E\ E')\ u\ E1) \\
\text{SUM2:} & \vdash \forall E\ u\ E1.\ \mathsf{Trans}\ E\ u\ E1\ \supset (\forall E'.\ \mathsf{Trans}\ (\mathsf{sum}\ E'\ E)\ u\ E1) \\
\text{RESTR:} & \vdash \forall E\ u\ E'\ L. \\
 & \quad (\exists l.\ \mathsf{Trans}\ E\ u\ E' \wedge \\
 & \quad ((u = \tau) \vee ((u = \mathsf{label}\ l) \wedge (l \notin L) \wedge (\mathsf{Compl}\ l \notin L))))\ \supset \\
 & \quad \mathsf{Trans}\ (\mathsf{restr}\ E\ L)\ u\ (\mathsf{restr}\ E'\ L) \\
\text{RELAB:} & \vdash \forall E\ u\ E'. \\
 & \quad \mathsf{Trans}\ E\ u\ E'\ \supset \\
 & \quad (\forall f.\ \mathsf{Trans}\ (\mathsf{relab}\ E\ f)\ (\mathsf{Apply\_Relab}\ f\ u)\ (\mathsf{relab}\ E'\ f)) \\
\text{PAR1:} & \vdash \forall E\ u\ E1. \\
 & \quad \mathsf{Trans}\ E\ u\ E1\ \supset (\forall E'.\ \mathsf{Trans}\ (\mathsf{par}\ E\ E')\ u\ (\mathsf{par}\ E1\ E')) \\
\text{PAR2:} & \vdash \forall E\ u\ E1. \\
 & \quad \mathsf{Trans}\ E\ u\ E1\ \supset (\forall E'.\ \mathsf{Trans}\ (\mathsf{par}\ E'\ E)\ u\ (\mathsf{par}\ E'\ E1)) \\
\text{PAR3:} & \vdash \forall E\ E1\ E'\ E2. \\
 & \quad (\exists l.\ \mathsf{Trans}\ E\ (\mathsf{label}\ l)\ E1\ \wedge\ \mathsf{Trans}\ E'\ (\mathsf{label}(\mathsf{Compl}\ l))\ E2)\ \supset \\
 & \quad \mathsf{Trans}\ (\mathsf{par}\ E\ E')\ \tau\ (\mathsf{par}\ E1\ E2) \\
\text{REC:} & \vdash \forall E\ X\ u\ E1. \\
 & \quad \mathsf{Trans}\ (\mathsf{CCS\_Subst}\ E\ (\mathsf{rec}\ X\ E)\ X)\ u\ E1\ \supset \\
 & \quad \mathsf{Trans}\ (\mathsf{rec}\ X\ E)\ u\ E1
\end{array}
$$

where $\mathsf{Apply\_Relab}\ f\ u$ performs the renaming of the action u via the relabelling function f, and the function $\mathsf{CCS\_Subst}$ implements the substitution $E\{\mathsf{rec}\,X.\,E/X\}$ of $\mathsf{rec}\,X.\,E$ for all free occurrences of X in E. Such a function can be defined in HOL through a primitive recursive definition over the type CCS.[1]

Proving that Trans is the least relation closed under the transition rules results in a "rule induction" theorem, from which a tactic is generated for proofs by induction over the structure of the derivations defined by the transition rules. The inductive definition package provides other tactics for supporting goal directed proofs about the relation Trans. A tactic that reduces a goal which matches the conclusion of a transition rule can be defined for each of the theorems corresponding to the transition rules. For example, a tactic `SUM1_TAC` is generated from the above theorem SUM1, such that to prove the goal $\Gamma \mathrel{?\!\!\vdash} \mathsf{Trans}\,(\mathsf{sum}\ E1\ E2)\ u\ E$ is reduced to prove $\Gamma \mathrel{?\!\!\vdash} \mathsf{Trans}\ E1\ u\ E$.

Finally, a theorem for performing exhaustive case analysis over the inductively defined relation is provided. In our case, this means that if there is a transition $\mathsf{Trans}\ E\ u\ E'$, this can only happen if one of the cases given by the transition rules holds. From this theorem many other useful theorems about the relation Trans can be derived. For instance, we can prove that the agent nil cannot perform any transition:

NIL_NO_TRANS: $\vdash \forall u\ E.\ \sim \mathsf{Trans}\ \mathsf{nil}\ u\ E$

and that, if an agent can perform an action, then that agent cannot be nil:

TRANS_IMP_NO_NIL: $\vdash \forall E\ u\ E'.\ \mathsf{Trans}\ E\ u\ E' \supset \sim (E = \mathsf{nil})$

In Section 7 we will use other theorems about the relation Trans which can be derived using all these proof tools. They are presented below:

TRANS_REC:
$\vdash \forall X\ E\ u\ E'.$
$\quad \mathsf{Trans}\,(\mathsf{rec}\ X\ E)\ u\ E' \;=\; \mathsf{Trans}\,(\mathsf{CCS\_Subst}\ E\,(\mathsf{rec}\ X\ E)\ X)\ u\ E'$

TRANS_SUM:
$\vdash \forall E\ E'\ u\ E''.\ \mathsf{Trans}\,(\mathsf{sum}\ E\ E')\ u\ E'' \supset \mathsf{Trans}\ E\ u\ E'' \lor \mathsf{Trans}\ E'\ u\ E''$

PREFIX_cases:
$\vdash \forall u\ E\ u'\ E'.\ \mathsf{Trans}\,(\mathsf{prefix}\ u\ E)\ u'\ E' \supset (u = u') \land (E = E')$

Many other theorems about Trans can be derived in a similar way. All theorems and tactics provided by the induction definition package greatly aid the task of proving properties of the transition relation and of deriving the algebraic laws for behavioural semantics.

4.3 The Observational Congruence

Having formalized the labelled transition relation for the CCS operators, the notions of bisimulation and observational congruence can now be defined by using the basic definition mechanisms of HOL, and then the algebraic laws for observational congruence can be derived by formal proof.

The reflexive-transitive closure of the transition relation $E \xrightarrow{\tau} E'$, represented in HOL by $\mathsf{Eps}\ E\ E'$, is first defined by invoking the derived rule for inductive definitions. The relation

$\mathsf{Eps} : CCS \rightarrow CCS \rightarrow bool$

[1]Note that, for the time being, $\mathsf{CCS\_Subst}$ works under the assumption that variables bound in recursive processes are distinct, in order to avoid capture of free variables.

is defined such that it satisfies the following rules for reflexive-transitive closure:

ONE_TAU: $\vdash \forall E\ E'.\ \mathsf{Trans}\ E\ \tau\ E' \supset \mathsf{Eps}\ E\ E'$

EPS_REFL: $\vdash \forall E.\ \mathsf{Eps}\ E\ E$

EPS_TRANS: $\vdash \forall E\ E'.\ (\exists E1.\ \mathsf{Eps}\ E\ E1 \wedge \mathsf{Eps}\ E1\ E') \supset \mathsf{Eps}\ E\ E'$

As for the transition relation Trans, the derived HOL rule for inductive definition proves that the relation Eps is the least relation closed under the above rules, thus resulting in the following rule induction theorem:

EPS_IND:
$$\begin{array}{l}\vdash \forall P.\\ \quad (\forall E\ E'.\ \mathsf{Trans}\ E\ \tau\ E' \supset P\ E\ E') \wedge \\ \quad (\forall E.\ P\ E\ E) \wedge \\ \quad (\forall E\ E'.\ (\exists E1.\ P\ E\ E1 \wedge P\ E1\ E') \supset P\ E\ E') \\ \qquad \supset \\ \quad (\forall E\ E'.\ \mathsf{Eps}\ E\ E' \supset P\ E\ E')\end{array}$$

Tactics that reduce goals matching the conclusion of the rules for Eps are provided as well, and the theorem to perform exhaustive case analysis over the relation Eps is the following:

EPS_cases:
$$\begin{array}{l}\vdash \forall E\ E'.\\ \quad \mathsf{Eps}\ E\ E' = \\ \quad \mathsf{Trans}\ E\ \tau\ E' \vee (E' = E) \vee (\exists E1.\ \mathsf{Eps}\ E\ E1 \wedge \mathsf{Eps}\ E1\ E')\end{array}$$

which states that $E \stackrel{\tau}{\longrightarrow} E'$ if and only if one of the cases given by the rules for Eps holds.

The weak transition relation $E \stackrel{u}{\Longrightarrow} E'$, represented by $\mathsf{Weak\_Trans}\ E\ u\ E'$, is formalized using a basic HOL definition mechanism. The relation

$$\mathsf{Weak\_Trans} : \mathit{CCS} \rightarrow \mathit{action} \rightarrow \mathit{CCS} \rightarrow \mathit{bool}$$

is defined by making a constant definition as follows:

$$\begin{array}{l}\forall E\ u\ E'.\\ \quad \mathsf{Weak\_Trans}\ E\ u\ E' = \\ \quad (\exists E1\ E2.\ \mathsf{Eps}\ E\ E1 \wedge \mathsf{Trans}\ E1\ u\ E2 \wedge \mathsf{Eps}\ E2\ E')\end{array}$$

At this point we are already able to prove a theorem which will be very useful when deriving properties and algebraic laws for observational congruence. This theorem asserts that a transition $\mathsf{Trans}\ E\ u\ E'$ is a particular weak transition $\mathsf{Weak\_Trans}\ E\ u\ E'$:

TRANS_IMP_WEAK_TRANS:
$$\vdash \forall E\ u\ E'.\ \mathsf{Trans}\ E\ u\ E' \supset \mathsf{Weak\_Trans}\ E\ u\ E'$$

The proof of this theorem is very simple and makes use of rewriting with the definition of the relations Weak_Trans and Eps.

The notion of weak bisimulation

$$\mathsf{Weak\_Bisim} : \mathit{CCS} \rightarrow \mathit{CCS} \rightarrow \mathit{bool}$$

can then be defined in HOL by means of a constant definition, thus obtaining the following:

$$
\begin{array}{l}
\forall\, Wbsm. \\
\quad \mathsf{Weak\_Bisim}\; Wbsm = \\
\quad (\forall E\; E'. \\
\qquad Wbsm\; E\; E' \;\supset \\
\qquad (\forall l. \\
\qquad\quad (\forall E1. \\
\qquad\qquad \mathsf{Trans}\; E\; (\mathsf{label}\; l)\; E1 \;\supset \\
\qquad\qquad (\exists E2.\; \mathsf{Weak\_Trans}\; E'\; (\mathsf{label}\; l)\; E2 \;\wedge Wbsm\; E1\; E2)) \wedge \\
\qquad\quad (\forall E2. \\
\qquad\qquad \mathsf{Trans}\; E'\; (\mathsf{label}\; l)\; E2 \;\supset \\
\qquad\qquad (\exists E1.\; \mathsf{Weak\_Trans}\; E\; (\mathsf{label}\; l)\; E1 \;\wedge Wbsm\; E1\; E2))) \wedge \\
\qquad (\forall E1.\; \mathsf{Trans}\; E\; \tau\; E1 \;\supset (\exists E2.\; \mathsf{Eps}\; E'\; E2 \;\wedge Wbsm\; E1\; E2)) \wedge \\
\qquad (\forall E2.\; \mathsf{Trans}\; E'\; \tau\; E2 \;\supset (\exists E1.\; \mathsf{Eps}\; E\; E1 \;\wedge Wbsm\; E1\; E2)))
\end{array}
$$

Theorems about weak bisimulation can now be derived. By making use of the above theorem TRANS_IMP_WEAK_TRANS, we can prove, among others, that the identity relation is a weak bisimulation, the converse of a weak bisimulation is a weak bisimulation, and that the union and the composition of two weak bisimulations are weak bisimulations.

The observational equivalence $E \approx E'$, represented by $\mathsf{Obs\_Equiv}\; E\; E'$, is mechanized by making a constant definition, where the relation

$$\mathsf{Obs\_Equiv} : CCS \rightarrow CCS \rightarrow bool$$

is defined as follows:

$$\forall E\; E'.\; \mathsf{Obs\_Equiv}\; E\; E' = (\exists Wbsm.\; Wbsm\; E\; E' \;\wedge \mathsf{Weak\_Bisim}\; Wbsm)$$

Using the above theorems about weak bisimulation, it is easy to show that the observational equivalence is an equivalence relation by proving that it is reflexive, symmetric and transitive. We can also prove that observational equivalence is preserved by all the CCS operators, except for summation.

Finally, based on the definition of observational equivalence, the observational congruence $E = E'$, represented in HOL by $\mathsf{Obs\_Congr}\; E\; E'$, is formalized using the constant definition mechanism by defining the relation

$$\mathsf{Obs\_Congr} : CCS \rightarrow CCS \rightarrow bool$$

as follows:

$$
\begin{array}{l}
\forall E\; E'. \\
\quad \mathsf{Obs\_Congr}\; E\; E' = \\
\quad (\forall u. \\
\qquad (\forall E1. \\
\qquad\quad \mathsf{Trans}\; E\; u\; E1 \;\supset \\
\qquad\quad (\exists E2. \mathsf{Weak\_Trans}\; E'\; u\; E2 \;\wedge\; \mathsf{Obs\_Equiv}\; E1\; E2)) \wedge \\
\qquad (\forall E2. \\
\qquad\quad \mathsf{Trans}\; E'\; u\; E2 \;\supset \\
\qquad\quad (\exists E1. \mathsf{Weak\_Trans}\; E\; u\; E1 \;\wedge\; \mathsf{Obs\_Equiv}\; E1\; E2)))
\end{array}
$$

Theorems similar to those proved for observational equivalence can be derived, such as showing that observational congruence is an equivalence relation and is preserved by all CCS operators. It can also be proved that observational congruence implies observational equivalence.

4.4 The Axioms for Observational Congruence

The algebraic laws for the CCS operators given in Section 3.3 can be derived starting from the definition of observational congruence. All of them have been formally derived in HOL. Most of the laws are proved using the above definitions and the theorems derived for the transition relations Trans and Weak_Trans, plus the theorem stating that observational equivalence is an equivalence relation, thus avoiding exhibiting an appropriate weak bisimulation. The only law which requires some explanation is, in fact, the *expansion* law (A11) for the parallel operator. We first present the formalization of some of the other laws to illustrate their similarity to the mathematical presentation given earlier, thus demonstrating the suitability of HOL for supporting other notations. The laws for the summation operator, the *unfolding* law for recursion and the τ-laws for observational congruence are as follows:

SUM_ASSOC: $\vdash \forall E\ E'\ E''.$
  $\mathsf{Obs\_Congr}\,(\mathsf{sum}\,(\mathsf{sum}\ E\ E')\ E'')\,(\mathsf{sum}\ E\,(\mathsf{sum}\ E'\ E''))$

SUM_COMM: $\vdash \forall E\ E'.\ \mathsf{Obs\_Congr}\,(\mathsf{sum}\ E\ E')\,(\mathsf{sum}\ E'\ E)$

SUM_IDEMP: $\vdash \forall E.\ \mathsf{Obs\_Congr}\,(\mathsf{sum}\ E\ E)\ E$

SUM_IDENT: $\vdash \forall E.\ \mathsf{Obs\_Congr}\,(\mathsf{sum}\ E\ \mathsf{nil})\ E$

UNFOLDING: $\vdash \forall X\ E.$
  $\mathsf{Obs\_Congr}\,(\mathsf{rec}\ X\ E)\,(\mathsf{CCS\_Subst}\ E\,(\mathsf{rec}\ X\ E)\ X)$

TAU_1: $\vdash \forall u\ E.\ \mathsf{Obs\_Congr}\,(\mathsf{prefix}\ u\,(\mathsf{prefix}\ \mathsf{tau}\ E))\,(\mathsf{prefix}\ u\ E)$

TAU_2: $\vdash \forall E.\ \mathsf{Obs\_Congr}\,(\mathsf{sum}\ E\,(\mathsf{prefix}\ \mathsf{tau}\ E))\,(\mathsf{prefix}\ \mathsf{tau}\ E)$

TAU_3: $\vdash \forall u\ E\ E'.$
  $\mathsf{Obs\_Congr}$
  $(\mathsf{sum}\,(\mathsf{prefix}\ u\,(\mathsf{sum}\ E\,(\mathsf{prefix}\ \mathsf{tau}\ E')))(\mathsf{prefix}\ u\ E'))$
  $(\mathsf{prefix}\ u\,(\mathsf{sum}\ E\,(\mathsf{prefix}\ \mathsf{tau}\ E')))$

To formalize the expansion law (A11), we first need to define the notation used for indexed summation. In particular, we define the indexed summation of prefixed agents, which is the one used in the law (A11). Given an index $n{:}num$ and a function $E : num \rightarrow CCS$, a function SIGMA_PREFIX is defined by primitive recursion such that SIGMA_PREFIX $n\ E$ denotes the summation PREF_ACT $(E\ 0)$. PREF_PROC $(E\ 0)$ + PREF_ACT $(E\ 1)$. PREF_PROC $(E\ 1) + \ldots +$ PREF_ACT $(E\ n)$. PREF_PROC $(E\ n)$:

$$
\begin{array}{l}
\vdash (\forall\, E.\\
\quad \mathsf{SIGMA\_PREFIX}\ 0\ E = \mathsf{prefix}\,(\mathsf{PREF\_ACT}\,(E\ 0))\,(\mathsf{PREF\_PROC}\,(E\ 0))) \wedge\\
\ \ (\forall\, n\ E.\\
\quad \mathsf{SIGMA\_PREFIX}\,(n{+}1)\ E\ =\\
\quad \mathsf{sum}\\
\quad (\mathsf{SIGMA\_PREFIX}\ n\ E)\\
\quad (\mathsf{prefix}\,(\mathsf{PREF\_ACT}\,(E\ (n{+}1)))\,(\mathsf{PREF\_PROC}\,(E\ (n{+}1)))))
\end{array}
$$

where PREF_ACT and PREF_PROC are the projection functions on prefixed agents used for extracting the action and the process, respectively:

$$
\begin{array}{l}
\vdash\ \forall\, u\ E.\ \mathsf{PREF\_ACT}\,(\mathsf{prefix}\ u\ E) = u\\
\vdash\ \forall\, u\ E.\ \mathsf{PREF\_PROC}\,(\mathsf{prefix}\ u\ E) = E
\end{array}
$$

We then define a function ALL_SYNC by primitive recursion which computes the summation of all possible synchronizations between two summation agents $E,\ E' : num \rightarrow CCS$ of length n, m respectively. This is done by using a function SYNC which computes the summation of all possible synchronizations between a single prefixed agent $u\,.\,P$ and a summation agent. Such a function is defined by primitive recursion as follows:

$$
\begin{array}{l}
\vdash\ (\forall\, u\, P\, E.\\
\quad \mathsf{SYNC}\ u\ P\ 0\ E =\\
\quad (((u = \mathsf{tau}) \lor (\mathsf{PREF\_ACT}\,(E\ 0) = \mathsf{tau})) \supset\\
\quad\ \ \mathsf{nil}\ |\\
\quad\ \ ((\mathsf{LABEL}\ u = \mathsf{Compl}\,(\mathsf{LABEL}\,(\mathsf{PREF\_ACT}\,(E\ 0)))) \supset\\
\quad\ \ \ \ \mathsf{prefix\ tau}\,(\mathsf{par}\ P\,(\mathsf{PREF\_PROC}\,(E\ 0)))\ |\ \mathsf{nil}))) \land\\
\ \ (\forall\, u\, P\, n\, E.\\
\quad\ \ \mathsf{SYNC}\ u\ P\,(n{+}1)\ E =\\
\quad\ \ (((u = \mathsf{tau}) \lor (\mathsf{PREF\_ACT}\,(E\,(n{+}1)) = \mathsf{tau})) \supset\\
\quad\ \ \ \ \mathsf{SYNC}\ u\ P\ n\ E\ \ |\\
\quad\ \ \ \ ((\mathsf{LABEL}\ u = \mathsf{Compl}\,(\mathsf{LABEL}\,(\mathsf{PREF\_ACT}\,(E\,(n{+}1))))) \supset\\
\quad\ \ \ \ \ \ \mathsf{sum}\\
\quad\ \ \ \ \ \ (\mathsf{prefix\ tau}\,(\mathsf{par}\ P\,(\mathsf{PREF\_PROC}\,(E\,(n{+}1)))))\\
\quad\ \ \ \ \ \ (\mathsf{SYNC}\ u\ P\ n\ E)\ \ |\\
\quad\ \ \ \ \ \ \mathsf{SYNC}\ u\ P\ n\ E)))
\end{array}
$$

where LABEL is a function which simply projects the label from an action:

$$\vdash\ \forall\, l.\ \mathsf{LABEL}\,(\mathsf{label}\ l) = l$$

The function ALL_SYNC is then defined as follows:

$$
\begin{array}{l}
\vdash\ (\forall\, E\, m\, E'.\\
\quad \mathsf{ALL\_SYNC}\ 0\ E\ m\ E'\ =\\
\quad \mathsf{SYNC}\,(\mathsf{PREF\_ACT}\,(E\ 0))\,(\mathsf{PREF\_PROC}\,(E\ 0))\ m\ E') \land\\
\ \ (\forall\, n\, E\, m\, E'.\\
\quad\ \ \mathsf{ALL\_SYNC}\,(n{+}1)\ E\ m\ E' =\\
\quad\ \ \mathsf{sum}\\
\quad\ \ (\mathsf{ALL\_SYNC}\ n\ E\ m\ E')\\
\quad\ \ (\mathsf{SYNC}\,(\mathsf{PREF\_ACT}\,(E\,(n{+}1)))\,(\mathsf{PREF\_PROC}\,(E\,(n{+}1)))\ m\ E'))
\end{array}
$$

The expansion law (A11) can then be derived, and its HOL formalization is the following:

$$
\begin{array}{l}
\vdash\ \forall\, n\, E\, m\, E'.\\
\quad \mathsf{Obs\_Congr}\\
\quad (\mathsf{par}\,(\mathsf{SIGMA\_PREFIX}\ n\ E)\,(\mathsf{SIGMA\_PREFIX}\ m\ E'))\\
\quad (\mathsf{sum}\\
\quad\ (\mathsf{sum}\\
\quad\ \ (\mathsf{SIGMA\_PREFIX}\\
\quad\ \ n\\
\quad\ \ (\lambda i.\\
\quad\ \ \ \ \ \mathsf{prefix}\,(\mathsf{PREF\_ACT}\,(E\ i))\\
\quad\ \ \ \ \ (\mathsf{par}\,(\mathsf{PREF\_PROC}\,(E\ i))\,(\mathsf{SIGMA\_PREFIX}\ m\ E'))))\\
\quad\ \ (\mathsf{SIGMA\_PREFIX}\\
\quad\ \ m\\
\quad\ \ (\lambda j.\\
\quad\ \ \ \ \ \mathsf{prefix}\,(\mathsf{PREF\_ACT}\,(E'\ j))\\
\quad\ \ \ \ \ (\mathsf{par}\,(\mathsf{SIGMA\_PREFIX}\ n\ E)\,(\mathsf{PREF\_PROC}\,(E'\ j))))))\\
\quad\ (\mathsf{ALL\_SYNC}\ n\ E\ m\ E'))
\end{array}
$$

4.5 The Modal Logic

The first step in the formalization in HOL of the modal logic is to represent its syntax. Again, this can be achieved by defining a concrete data type *eHML* of formulas of the extended Hennessy-Milner logic, using the derived rule for recursive type definition as follows:

$$
\begin{aligned}
eHML = {} & \mathtt{tt} \mid \\
& \mathsf{neg}\ eHML \mid \\
& \mathsf{conj}\ eHML\ eHML \mid \\
& \mathsf{box}\ (action)set\ eHML
\end{aligned}
$$

where $\mathtt{tt}$, neg, conj, and box are distinct constructors. Similarly to the definition of the types *label*, *action* and *CCS* (Section 4.1), a theorem which completely characterizes the type *eHML* is automatically derived.

The satisfaction relation $\mathsf{Sat} : CCS \rightarrow eHML \rightarrow bool$ can be defined using the derived principle for the definition of primitive recursion functions over the type *eHML*, thus obtaining the following list of theorems:

SAT_tt: $\vdash \forall E.\ \mathsf{Sat}\ E\ \mathtt{tt}\ =\ T$

SAT_neg: $\vdash \forall E\ Fm.\ \mathsf{Sat}\ E\ (\mathsf{neg}\ Fm)\ =\ \sim\ \mathsf{Sat}\ E\ Fm$

SAT_conj: $\vdash \forall E\ Fm\ Fm'.$
$\quad \mathsf{Sat}\ E\ (\mathsf{conj}\ Fm\ Fm')\ =\ \mathsf{Sat}\ E\ Fm\ \wedge\ \mathsf{Sat}\ E\ Fm'$

SAT_box: $\vdash \forall E\ A\ Fm.$
$\quad \mathsf{Sat}\ E\ (\mathsf{box}\ A\ Fm)\ =$
$\quad (\forall E'\ u.\ u \in A\ \wedge\ \mathsf{Trans}\ E\ u\ E'\ \supset\ \mathsf{Sat}\ E'\ Fm)$

The derived operators of the modal logic can then be defined through basic HOL definition mechanisms:

$$\mathtt{ff}\ =\ \mathsf{neg}\ \mathtt{tt}$$

$$\forall Fm\ Fm'.\ \mathsf{disj}\ Fm\ Fm'\ =\ \mathsf{neg}\ (\mathsf{conj}\ (\mathsf{neg}\ Fm)\ (\mathsf{neg}\ Fm'))$$

$$\forall A\ Fm.\ \mathsf{dmd}\ A\ Fm\ =\ \mathsf{neg}\ (\mathsf{box}\ A\ (\mathsf{neg}\ Fm))$$

The related theorems for the relation Sat can be easily proved by rewriting with these operator definitions and the above satisfaction rules for the basic operators:

SAT_ff: $\vdash \forall E.\ \mathsf{Sat}\ E\ \mathtt{ff}\ =\ F$

SAT_disj: $\vdash \forall E\ Fm\ Fm'.$
$\quad \mathsf{Sat}\ E\ (\mathsf{disj}\ Fm\ Fm')\ =\ \mathsf{Sat}\ E\ Fm\ \vee\ \mathsf{Sat}\ E\ Fm'$

SAT_dmd: $\vdash \forall E\ A\ Fm.$
$\quad \mathsf{Sat}\ E\ (\mathsf{dmd}\ A\ Fm)\ =$
$\quad (\exists E'\ u.\ u \in A\ \wedge\ \mathsf{Trans}\ E\ u\ E'\ \wedge\ \mathsf{Sat}\ E'\ Fm)$

A tactic that reduces a goal which matches the structure of formulas can still be obtained for each of the cases for the satisfaction relation. For example, a tactic `SAT_conj_TAC` is generated from SAT_conj such that to prove a goal Γ ?- $\mathsf{Sat}\ E\ (\mathsf{conj}\ Fm\ Fm')$ is reduced to prove the two subgoals Γ ?- $\mathsf{Sat}\ E\ Fm$ and Γ ?- $\mathsf{Sat}\ E\ Fm'$.

At the end of this section, it is worth noting that the above formalization demonstrates the suitability of HOL for supporting embedded notations, most of the definitions and axioms being very similar to their conventional presentation. On the other hand, more work than is originally expected can be involved when mechanizing definitions or axioms, because axioms written by hand are often packed with notation which itself needs to be formalized.

5 Reasoning about CCS

In this section we discuss briefly the mechanization of some verification strategies with different degrees of user interaction.

The degree of automation with which one might wish to reason about CCS can vary depending on the complexity of the specifications, and on the level of confidence one has in their correctness. For this reason, starting from the HOL formalization of the CCS theory, we have developed a set of inference rules, tactics and conversions to enable reasoning about CCS expressions by manipulating them according to their algebraic properties. The idea is that these rules, tactics and conversions can be used either interactively in a stepwise fashion, or composed together to give automatic strategies.

A wide range of verification strategies can be defined in this way based on our mechanization, depending on the subsets of CCS under consideration and on the kind of property to be proved. A strategy can be defined by using the algebraic laws to adopt selection criteria which depend on the state of the proof, as well as on the state of the expression being manipulated. In [23] a rewriting strategy has been defined which implements a term rewriting system equivalent to the axiomatization of observational congruence for finite CCS (Section 3.3). This strategy computes, fully automatically, the normal form of a finite CCS term with respect to the laws for observational congruence. Thus, the observational congruence of two finite CCS agents can be simply verified by checking their observational normal forms for equality. This strategy has been embedded in HOL [10] and will be referred to as `TAU_STRAT` from now on.

Another strategy is, instead, partially interactive. It deals with the unfolding law (A12) for the recursion operator and the expansion law (A11) for the parallel operator, and shows how it is possible to expand an expression, at the same time keeping its size to a minimum. This rewriting strategy, called *lazy expansion* in [10], manipulates an expression by expanding with the definition of the agents occurring in it, by rewriting with the laws for the CCS operators and by appropriately folding back some subexpressions with the definitions of the agents.

The above strategies can be used to verify several properties of CCS expressions, such as the behavioural equivalence between an abstract description of a system, usually referred to as *specification*, and a more detailed one, referred to as *implementation*. The verification problem consists of showing that the implementation is *correct* with respect to the specification or, equivalently, that the implementation *meets* the specification. This means showing the equivalence between implementation and specification with respect to a given behavioural semantics. The same rewriting strategies can also be used to prove that an agent expression satisfies a given logical formula.

In what follows we show how the HOL formalization of CCS described in the preceding section can be used to reason about CCS. In particular, we show how proofs of correctness by mathematical induction for parameterized CCS expressions can be mechanized, and how modal properties can be checked in our framework [30, 31].

6 Verification of a Simple Buffer by Induction

In this section we consider inductive reasoning and apply mathematical induction to prove the correctness of an implementation of a simple buffer with respect to its specification. This example is taken from [29].

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

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

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

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

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 $^\frown$ is a linking operator which, given two arbitrary agents E and E', is defined as follows:

$$E ^\frown E' \quad \equiv \quad (E\,[mid/out] \mid E'\,[mid/in]) \setminus \{mid\}$$

To show that $Impl(n)$ is a correct implementation of $Buffer_n$, we shall prove that $Impl(n)$ and $Buffer_n$ are observational congruent, i.e. for all $n \geq 1$

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

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.

6.1 Mechanizing the proof in HOL

Below we describe the HOL mechanization of the proof by presenting transcripts of a HOL session. 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. Moreover, to avoid using a verbose prefix notation, the parsing and pretty-printing facilities in the HOL system are extended to accept input and print output almost identical to the notation normally associated with CCS.[2]

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`, which can only input a value, and as a full cell `C'`, which can only output a value. These two specifications are defined in HOL by invoking the ML function `new_definition`, and the linking operator `Link` is defined as an infix operator by using the function `new_infix_definition`.

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

```
#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`,
   "∀E E'. E Link E' = (E[`mid`/`out`] | E'[`mid`/`in`])\{`mid`}");;
⊢ ∀E E'.
   E Link E' = (E[`mid`/`out`] | E'[`mid`/`in`])\{`mid`}
```

The implementation $Impl(n)$ of the buffer is a primitive recursive definition starting from 1, so 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 = prove_thm
             (`IMPL1`,
              "∃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)))
```

The specification of the buffer is not primitive recursive, and HOL does not yet provide a mechanism for defining mutually recursive functions. The existence of a function `BUFF_SPEC` of type `num → num → CCS` which satisfies the defining equations of $Buffer_n$ will be assumed in HOL, and in the rest of the paper all proofs and theorems involving the specification of the buffer will hold under the assumption `BUFF_SPEC_DEF`.

```
#let BUFF_SPEC_DEF =
     "∀n. 0 < n ⇒
        (BUFF_SPEC n 0 = `in`.(BUFF_SPEC n 1)) ∧
        (∀k. 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)))";;
```

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 hypothesis, like any assumption, is indicated with brackets `[ ]`. The assumption `BUFF_SPEC_DEF` will be shown only when setting the goal, and replaced by brackets with dots in the next steps.

```
#set_goal
 ([BUFF_SPEC_DEF],
  "∀n. Obs_Congr (BUFF_IMPL (SUC n)) (BUFF_SPEC (SUC n) 0)");;
"∀n. Obs_Congr (BUFF_IMPL(SUC n)) (BUFF_SPEC(SUC n)0)"
    [ "∀n. 0 < n ⇒
        (BUFF_SPEC n 0 = ‘in‘.(BUFF_SPEC n 1)) ∧
        (∀k. 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)))" ]

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

"Obs_Congr (BUFF_IMPL 1) (BUFF_SPEC 1 0)"
    [...]
```

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

```
#e (REWRITE_TAC [BUFF_IMPL; C] THEN REC_UNF_TAC THEN
    ONCE_REWRITE_TAC [SYM C] THEN
    SUBST1_TAC (SYM (CONJUNCT1 BUFF_IMPL)));;
OK..
"Obs_Congr (‘in‘.-‘out‘.(BUFF_IMPL 1)) (BUFF_SPEC 1 0)"
    [...]
```

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

```
#e (ONCE_REWRITE_TAC [CONJUNCT1 SPEC_SUC0_SUC0] THEN
    ONCE_REWRITE_TAC [CONJUNCT2 SPEC_SUC0_SUC0]);;
OK..
"Obs_Congr (‘in‘.-‘out‘.(BUFF_IMPL 1)) (‘in‘.-‘out‘.(BUFF_SPEC 1 0))"
    [...]
```

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 3.3). (Note that the dot before $\vdash$ abbreviates the assumption `BUFF_SPEC_DEF`.)

```
#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
⊢ Obs_Congr (`in`.-`out`.(BUFF_IMPL 1)) (`in`.-`out`.(BUFF_SPEC 1 0))
. ⊢ Obs_Congr (`in`.-`out`.(BUFF_IMPL 1)) (BUFF_SPEC 1 0)
. ⊢ Obs_Congr (BUFF_IMPL 1) (BUFF_SPEC 1 0)

Previous subproof:
"Obs_Congr (BUFF_IMPL(SUC(SUC n))) (BUFF_SPEC(SUC(SUC n))0)"
    [...]
    [ "Obs_Congr (BUFF_IMPL(SUC n)) (BUFF_SPEC(SUC n)0)" ]
```

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 then apply the inductive hypothesis by rewriting with one of the assumptions using the tactic `OC_SUBST_TAC` which performs substitution in terms of the form `Obs_Congr E E′`.

```
#e (REWRITE_TAC [BUFF_IMPL; Link] THEN
    OC_SUBST_TAC
    (ASSUME "Obs_Congr (BUFF_IMPL(SUC n)) (BUFF_SPEC(SUC n)0)"));;
OK..
"Obs_Congr
 (((C[`mid`/`out`]) | ((BUFF_SPEC(SUC n)0)[`mid`/`in`]))\{`mid`})
 (BUFF_SPEC(SUC(SUC n))0)"
    [...]
    [ "Obs_Congr (BUFF_IMPL(SUC n)) (BUFF_SPEC(SUC n)0)" ]
```

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

$$
\begin{array}{lll}
\mathit{Buffer}_{n+2}(k) & \text{by} \quad C^\frown \mathit{Buffer}_{n+1}(k) & (0 \leq k \leq n+1) \\
\mathit{Buffer}_{n+2}(n+2) & \text{by} \quad C'^\frown \mathit{Buffer}_{n+1}(n+1) & (k = n+2)
\end{array}
$$

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

$$
\begin{array}{lll}
C^\frown \mathit{Buffer}_{n+1}(0) & = \mathit{in}\,.(C^\frown \mathit{Buffer}_{n+1}(1)) & \\
C^\frown \mathit{Buffer}_{n+1}(k) & = \mathit{in}\,.(C^\frown \mathit{Buffer}_{n+1}(k+1)) & (0 < k < n+1) \\
 & \quad + \ \overline{\mathit{out}}\,.(C^\frown \mathit{Buffer}_{n+1}(k-1)) & \\
C^\frown \mathit{Buffer}_{n+1}(n+1) & = \mathit{in}\,.(C'^\frown \mathit{Buffer}_{n+1}(n+1)) & (k = n+1) \\
 & \quad + \ \overline{\mathit{out}}\,.(C^\frown \mathit{Buffer}_{n+1}(n)) & \\
C'^\frown \mathit{Buffer}_{n+1}(n+1) & = \overline{\mathit{out}}\,.(C^\frown \mathit{Buffer}_{n+1}(n+1)) &
\end{array}
$$

These congruences can be proved using the lazy expansion strategy, i.e. by rewriting each left-hand side with the definitions of the agents occurring in it and applying the laws 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:

$$
C'^\frown \mathit{Buffer}_n(k) \quad = \quad \tau\,.(C^\frown \mathit{Buffer}_n(k+1)) \qquad (0 \leq k < n)
$$

It represents the intuition behind Milner's proof. The specification $\mathit{Buffer}_n(k)$ 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 $\mathit{Buffer}_n(k+1)$.

The above lemma is proved by induction on k using the lazy expansion strategy. In the proof various tactics and theorems are used which we have previously defined and proved in HOL, to manipulate subexpressions of the goal and make the application of some laws concerning the action τ possible. Some of the theorems are the following:

FULL_TO_EMPTY_CELL: $\vdash$ $\mathsf{Obs\_Congr}\, C'\, (\overline{out}\,.\,C)$

TRANSF_FULL_CELL:
$\vdash$ $\mathsf{Obs\_Congr}\, (C'\,[mid/out])\, (\overline{mid}.\,(C\,[mid/out]))$

EXP_ABS_THM:
$. \vdash \forall n\, k.$
$(0 < n) \wedge (k + 2 \leq n) \supset$
$\mathsf{Obs\_Congr}$
$(C \frown (\mathit{Buffer}_n(k+2)))$
$(in.\,(C' \frown (\mathit{Buffer}_n(k+2))) + \overline{out}.\,(C \frown (\mathit{Buffer}_n(k+1))))$

Moreover, the rewriting strategy `TAU_STRAT` is used which, in the proof of the lemma, applies the derived τ-law, $E + \tau\,.\,(F + E) = \tau\,.\,(F + E)$.

Below we present the HOL mechanization of the 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
    (([BUFF_SPEC_DEF],
      "∀k n.
         ((0 < n) ∧ (k < n)) ⇒
          Obs_Congr
          (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 the assumption list
       THEN Use the theorem FULL_TO_EMPTY_CELL
       THEN Rewrite using the definition of BUFF_SPEC
       THEN Apply laws for relabelling, parallel and restriction
       ;
       Strip off the universally quantified variable n and move
       conjuncts of antecedent of implication to the assumption list
       THEN Use the theorem FULL_TO_EMPTY_CELL
       THEN Rewrite using the definition of BUFF_SPEC
       THEN Apply laws 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 ⇒
      Obs_Congr
      (C′ Link (BUFF_SPEC n k))
      (tau.(C Link (BUFF_SPEC n(SUC k))))
```

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 lazy expansion strategy.

7 Proving Modal Properties of CCS Specifications

In this section we show how modal properties of CCS agents can be checked in our HOL-CCS environment by means of a simple example.

Let us first see how the satisfaction relation for the notions of capacity and inability to perform actions of a given set A (Section 3.4) can be derived. After having entered a HOL theory in which we reason about the Hennessy-Milner logic, we can prove that the formula $\langle A\rangle$`tt` expresses a capacity to perform an action in A. This can be achieved by applying the following simple tactic which strips off the universally quantified variables E and A using the tactic `GEN_TAC`, and then rewrites the current goal with the theorems for Sat corresponding to the logical operators in the goal:

```
#let CAPC_ACT =
      prove_thm
        (`CAPC_ACT`,
         "∀E A. Sat E (⟨A⟩ tt) = ∃E′ u. u ∈ A ∧ Trans E u E′",
         REPEAT GEN_TAC THEN
         REWRITE_TAC [SAT_dmd; SAT_tt]);;

CAPC_ACT = ⊢ ∀E A. Sat E(⟨A⟩tt) = (∃E′ u. u ∈ A ∧ Trans E u E′)
```

In a similar way, we can easily prove that the formula $\langle A\rangle$`ff` expresses an inability to perform any action in A.

```
#let INAB_ACT =
      prove_thm
        (`INAB_ACT`,
         "∀E A. Sat E ([A] ff) = ∼(∃E′ u. u ∈ A ∧ Trans E u E′)",
         REPEAT GEN_TAC THEN
         REWRITE_TAC [SAT_box; SAT_ff] THEN
         CONV_TAC (TOP_DEPTH_CONV NOT_EXISTS_CONV) THEN
         ONCE_REWRITE_TAC[]);;

INAB_ACT = ⊢ ∀E A. Sat E([A]ff) = ∼(∃E′ u. u ∈ A ∧ Trans E u E′)
```

Note that the tactic `CONV_TAC (TOP_DEPTH_CONV NOT_EXISTS_CONV)` moves negation inwards through the existential quantifications.

Let us now show how to check that an agent satisfies a given modal property. Let a simple vending machine be defined by the following CCS agent expression (this example is taken from [35]):

$$V \stackrel{\text{def}}{=} \mathsf{rec}\, X.\, 2p.big.collect.X + 1p.little.collect.X$$

This agent can be defined in HOL by the following definition:

```
#new_definition
   (`V`,
    "V =
     rec
     `X`
     (`2p`.`big`.`collect`.`X` + `1p`.`little`.`collect`.`X`)");;
⊢ V =
   rec `X` (`2p`.`big`.`collect`.`X` + `1p`.`little`.`collect`.`X`)
```

Several properties can be proved of this vending machine. For example, we can show that a button cannot be depressed before money is deposited into the machine. This property can be expressed in Hennessy-Milner logic by the formula $[\{big, little\}]$`ff`. In our HOL-CCS environment to show $V \models [\{big, little\}]$`ff` means proving the following goal:

```
#g "Sat V ([{‘big‘, ‘little‘}] ff)";;
"Sat V([{‘big‘,‘little‘}]ff)"
```

We start the proof by rewriting the goal with the definition of `V` and with the theorem `INAB_ACT`:

```
#e (REWRITE_TAC [V; INAB_ACT]);;
OK..
"~(∃E′ u.
    u ∈ {‘big‘,‘little‘} ∧
    Trans
    (rec ‘X‘ (‘2p‘.‘big‘.‘collect‘.‘X‘ + ‘1p‘.‘little‘.‘collect‘.‘X‘))
    u E′)"
```

We unfold the recursive expression by rewriting with the theorem TRANS_REC (Section 4.2), then apply the substitution of agents with CCS_Subst, and finally fold back the obtained expression using the definition of `V`:

```
#e (REWRITE_TAC [TRANS_REC; CCS_Subst; SYM V]);;
OK..
"~(∃E′ u.
    u ∈ {‘big‘,‘little‘} ∧
    Trans (‘2p‘.‘big‘.‘collect‘.V + ‘1p‘.‘little‘.‘collect‘.V) u E′)"
```

Stripping quantified variables and moving antecedents into the assumptions of the goal results in the following:

```
#e (REPEAT STRIP_TAC);;
OK..
"F"
    [ "u ∈ {‘big‘,‘little‘}" ]
    [ "Trans
       (‘2p‘.‘big‘.‘collect‘.V + ‘1p‘.‘little‘.‘collect‘.V) u E′" ]
```

By applying the HOL resolution with the implicational theorem TRANS_SUM (Section 4.2), we derive new assumptions from those in the list of the current goal, and get two subgoals to prove:

```
#e (IMP_RES_TAC TRANS_SUM);;
OK..
2 subgoals
"F"
    [ "u ∈ {‘big‘,‘little‘}" ]
    [ "Trans
       (‘2p‘.‘big‘.‘collect‘.V + ‘1p‘.‘little‘.‘collect‘.V) u E′" ]
    [ "Trans (‘1p‘.little‘.‘collect‘.V) u E′" ]

"F"
    [ "u ∈ {‘big‘,‘little‘}" ]
    [ "Trans
       (‘2p‘.‘big‘.‘collect‘.V + ‘1p‘.‘little‘.‘collect‘.V) u E′" ]
    [ "Trans (‘2p‘.‘big‘.‘collect‘.V) u E′" ]
```

To prove these subgoals, the conclusion of which is `F` (false), we have to show that the assumptions are inconsistent. As regards the first subgoal, i.e. the one at the bottom, the HOL resolution can be applied using the theorem PREFIX_cases (Section 4.2), thus deriving new assumptions:

```
#e (IMP_RES_TAC PREFIX_cases);;
OK..
"F"
    [ "u ∈ {‘big‘,‘little‘}" ]
    [ "Trans
       (‘2p‘.‘big‘.‘collect‘.V + ‘1p‘.‘little‘.‘collect‘.V) u E′" ]
    [ "Trans (‘2p‘.‘big‘.‘collect‘.V) u E′" ]
    [ "‘2p‘ = u" ]
    [ "‘big‘.‘collect‘.V = E′" ]
```

The assumption $u \in \{big, little\}$ is now rewritten with the newly-derived $2p = u$, thus obtaining a new assumption $2p \in \{big, little\}$:

```
#e (ASSUME_TAC (ONCE_REWRITE_RULE [SYM (ASSUME "‘2p‘ = u")]
                    (ASSUME "u ∈ {‘big‘,‘little‘}")));;
OK..
"F"
    [ "u ∈ {‘big‘,‘little‘}" ]
    [ "Trans
       (‘2p‘.‘big‘.‘collect‘.V + ‘1p‘.‘little‘.‘collect‘.V) u E′" ]
    [ "Trans (‘2p‘.‘big‘.‘collect‘.V) u E′" ]
    [ "‘2p‘ = u" ]
    [ "‘big‘.‘collect‘.V = E′" ]
    [ "‘2p‘ ∈ {‘big‘,‘little‘}" ]
```

By applying the HOL resolution using a pre-defined conversion `Action_IN_CONV` for deciding membership in a set of actions, a contradiction is derived which proves the first subgoal:

```
#e (IMP_RES_TAC (Action_IN_CONV "‘2p‘" "{‘big‘, ‘little‘}"));;
OK..
goal proved
. |- F
.. |- F
.. |- F

Previous subproof:
"F"
    [ "u ∈ {‘big‘,‘little‘}" ]
    [ "Trans
       (‘2p‘.‘big‘.‘collect‘.V + ‘1p‘.‘little‘.‘collect‘.V) u E′" ]
    [ "Trans (‘1p‘.little‘.‘collect‘.V) u E′" ]
```

The second subgoal can be proved in a similar way, thus solving the initial goal. Both subgoals are proved with the same tactic: new assumptions are derived by applying the HOL resolution using the theorem PREFIX_cases, and then assumptions are manipulated in such a way that a contradiction is derived. A tactic `MODAL_TAC` can be defined parametrically and invoked with appropriate arguments to solve a class of goals related to checking that an agent has a given modal property:

```
#let MODAL_TAC as1 as2 =
     IMP_RES_TAC PREFIX_cases THEN
     ASSUME_TAC
      (ONCE_REWRITE_RULE [SYM (ASSUME as1)] (ASSUME as2)) THEN
     IMP_RES_TAC
      (Action_IN_CONV (fst (dest_eq as1)) (snd (dest_comb as2)));;
MODAL_TAC = - : (term -> term -> tactic)
```

For example, the first subgoal above can be solved by invoking `MODAL_TAC` with the assumptions $as1$ and $as2$ given by $2p = u$ and $u \in \{big, little\}$, respectively.

Many other and more complex properties of the vending machine can be checked in a similar way. Moreover, modal properties of parameterized specifications can be naturally checked using proofs by induction [31].

8 Related Work and Conclusions

Several verification tools based on process algebras, such as Concurrency Workbench [12], Auto [34], TAV [18], Aldebaran [16], Squiggles [4], have been proposed for proving properties of concurrent systems [36, 37, 38]. These tools work in the framework of CCS-like specifications and most of them resort to a finite state machine representation of processes. This internal representation is used to verify equivalences of processes and to show that a process satisfies a logical property by means of some reasonably efficient automatic algorithms.

An automata based approach has the well-known problem of state explosion (the number of states of a concurrent system potentially increases exponentially in the number of its parallel components) and the limitation that 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 process algebras supports such reasoning. Moreover, more general and powerful proof techniques are sometimes required, such as induction, contradiction, case analysis, etc., and it is often convenient to define proofs parametrically so that they can be used to deal with a class of processes and/or logical properties.

Recently, several investigations into verification environments based on the algebraic nature of the concurrency calculi have been carried out, which allow for a better understanding of the process algebra specifications one is trying to verify than the finite state machine approach. They include axiomatic tools in which the signature of a calculus and the laws for behavioural semantics are just entered and then used to construct proofs in specially designed proof tools [25, 38] or in general purpose theorem provers like LP, RRL and the Boyer-Moore theorem prover [24, 1]. Other work in this field includes the formalization in the HOL system of different CSP semantics [8, 9] and of Milner's π-calculus [27] following a purely definitional approach to using higher order logic. This means that only primitive definition mechanisms are used for introducing new entities in a sound way, and all other kinds of definitions, such as the laws for behavioural semantics, are derived by formal proof. To our knowledge, however, these tools have not yet addressed the issue of checking modal properties.

In this paper we have presented the formalization in HOL of some components of the CCS process algebra, i.e. its syntax and operational semantics, the observational semantics and its axiomatic presentation, and a modal logic. We have shown how this mechanization can be actually used to perform verification of behavioural equivalences and check logical properties of specifications.

Extensions to the subset of CCS and the process logic can be embedded in the HOL system. For example, a more expressive *temporal* logic [35, 6] can be represented in higher order logic, and proof tools, e.g. the *tableau system* (extended to deal with infinite state processes in [7]), can be soundly mechanized. The tableau system decision procedure has been implemented in some verification tools, e.g. the Concurrency Workbench [12]. On the other hand, such a technique is also naturally described as a goal directed proof system and, as such, is amenable to be formalized in a theorem proving system which provides goal directed proofs. We believe that this demonstrates further evidence that the formal theory for a process language can be embedded in a theorem proving system to provide an effective approach to the mechanical verification of concurrent systems.

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, Brian Graham, John Harrison and Richard Boulton for their advice on mechanization in HOL. Thanks are also due to Paola Inverardi (I.E.I.-C.N.R., Pisa) for her continued support.

References

[1] Aujla, S. S. and M. Fletcher, 'The Boyer-Moore Theorem Prover and LOTOS', in *Formal Description Techniques*, Proceedings of *FORTE'88*, K. J. Turner (ed.), North-Holland, 1989.

[2] Bergstra, J. A. and J. W. Klop, 'Process Algebra for Synchronous Communication', *Information and Control* **60**, 1984, pp. 109–137.

[3] Bolognesi, T. and E. Brinksma, 'Introduction to the ISO Specification Language LOTOS', in *Computer Networks and ISDN Systems* **14**, North-Holland, 1987, pp. 25–59.

[4] Bolognesi, T. and M. Caneve, 'Squiggles — A Tool for the Analysis of LOTOS Specifications', in *Formal Description Techniques*, K. Turner (ed.), North-Holland, 1989, pp. 201–216.

[5] Boreale, M., P. Inverardi, and M. Nesi, 'Complete Sets of Axioms for Finite Basic LOTOS Behavioural Equivalences', *Information Processing Letters* **43**, 1992, pp. 155–160.

[6] Bradfield, J. and C. Stirling, 'Verifying Temporal Properties of Processes', in Proceedings of *Concur'90*, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, pp. 115–125.

[7] Bradfield, J. and C. Stirling, 'Local Model Checking for Infinite State Spaces', *Journal of Theoretical Computer Science* **96**, 1992, pp. 157–174.

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

[9] Camilleri, A. J., 'A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics', in Proceedings of the *4th Banff Higher Order Workshop*, 1990, G. Birtwistle (ed.), Workshops in Computing Series, Springer-Verlag, London, 1991, pp. 123–150.

[10] Camilleri, A. J., P. Inverardi, and M. Nesi, 'Combining Interaction and Automation in Process Algebra Verification', in Proceedings of *TAPSOFT'91*, S. Abramsky and T. S. E. Maibaum (eds.), Lecture Notes in Computer Science, Springer-Verlag, 1991, Vol. 494, pp. 283–296.

[11] Church, A., 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, 1940, Vol. 5, pp. 56–68.

[12] Cleaveland, R., J. Parrow, and B. Steffen, 'The Concurrency Workbench', in [36], pp. 24–37.

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

[14] De Nicola, R. and M. C. Hennessy, 'Testing Equivalence for Processes', *Journal of Theoretical Computer Science* **34**, 1984, pp. 83–133.

[15] DSTO, The University of Cambridge, SRI International, 'The HOL System: DESCRIPTION', 1991.

[16] Fernandez, J. C., 'Aldebaran: Un système de vérification par réduction de processus communicants', Ph. D. Thesis, Université de Grenoble, 1988.

[17] van Glabbeek, R. J. and W. P. Weijland, 'Branching Time and Abstraction in Bisimulation Semantics', in Proceedings of the 11*th IFIP World Computer Congress*, San Francisco, 1989.

[18] Godskesen, J. C., K. G. Larsen, and M. Zeeberg, 'TAV Users Manual', Technical Report R-89-19, Ålborg University, 1989.

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

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

[21] Hennessy, M. and R. Milner, 'Algebraic Laws for Nondeterminism and Concurrency', *Journal of ACM* **32**, No. 1, 1985, pp. 137–161.

[22] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice Hall, London, 1985.

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

[24] Kirkwood, C. and K. Norrie, 'Some Experiments Using Term Rewriting Techniques for Concurrency', in *Formal Description Techniques III*, J. Quemada, J. Mañas and E. Vazquez (eds.), North-Holland, 1991, pp. 527–530.

[25] Lin, H., 'PAM: A Process Algebra Manipulator', in [38], pp. 136–146.

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

[27] Melham, T. F., 'A Mechanized Theory of the π-calculus in HOL', Technical Report No. 244, Computer Laboratory, University of Cambridge, 1992.

[28] Melham, T. F., 'A Package for Inductive Relation Definitions in HOL', in Proceedings of the *1991 International Workshop on the HOL Theorem Proving System and its Applications*, P. J. Windley, M. Archer, K. N. Levitt and J. J. Joyce (eds.), IEEE Computer Society Press, 1992, pp. 350–357.

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

[30] Nesi, M., 'Mechanizing a Proof by Induction of Process Algebra Specifications in Higher Order Logic', in [38], pp. 288–298.

[31] Nesi, M., 'Formalizing a Modal Logic for CCS in the HOL Theorem Prover', in Proceedings of the *International Workshop on Higher Order Logic Theorem Proving and Its Applications*, L. Claesen and M. Gordon (eds.), Leuven, Belgium, September 1992, pp. 495–507.

[32] Paulson, L. C., 'A Higher-Order Implementation of Rewriting', *Science of Computer Programming* **3**, 1983, pp. 119–149.

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

[34] de Simone, R. and D. Vergamini, 'Aboard AUTO', Technical Report 111, INRIA, 1989.

[35] Stirling, C., 'An Introduction to Modal and Temporal Logics for CCS', in Proceedings of the *Joint UK/Japan Workshop on Concurrency*, Oxford, 1989, Lecture Notes in Computer Science, Springer-Verlag, Vol. 491, 1991, pp. 2–20.

[36] Proceedings of the *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, 1989, J. Sifakis (ed.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 407, 1990.

[37] Proceedings of the *2nd Workshop on Computer Aided Verification*, New Brunswick, New Jersey, 1990, E. M. Clarke and R. P. Kurshan (eds.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 531, 1991.

[38] Proceedings of the *3rd Workshop on Computer Aided Verification*, Ålborg University, 1991, K. G. Larsen and A. Skou (eds.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 575, 1992.