Technical Report

UCAM-CL-TR-411
ISSN 1476-2986

Number 411

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Formalising process calculi in Higher Order Logic

Monica Nesi

January 1997

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 1996 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Girton College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-411*

Abstract

In the past few years, several methods and tools based on *process calculi* have been developed for verifying properties of concurrent and communicating systems. In this dissertation the interactive theorem prover HOL is used as a framework for supporting reasoning about process calculi based on all the various components of their formal theory. The aim is to build a sound and effective tool to allow both verification of process specifications and meta-theoretic reasoning. In particular, the process calculus CCS is embedded in the HOL logic. This is achieved by first addressing the pure subset of this calculus (no value passing) and then extending it to its value-passing version. The CCS theory is mechanised in HOL by following a *purely definitional* approach. This means that new objects are embedded in HOL using definition mechanisms which guarantee that no inconsistencies are introduced in the logic, and by deriving new facts from definitions and/or previously proved theorems by formal proof.

Pure CCS agent expressions are encoded as a type in the HOL logic, in which initially actions are represented as strings, agents with infinite behaviour are given through the *rec*-notation and agent summation is the usual binary operator. Recursive agents are then allowed to be defined through systems of recursive equations and to be parameterised. This makes the type of CCS expressions polymorphic and parameterised on the parameters' type. Operational and behavioural semantics and a modal logic are defined and their properties and laws derived in HOL. Several proof tools, such as inference rules, conversions and tactics, are developed to enable users to carry out their proofs in an interactive way and to automate them whenever possible. Properties of infinite state systems, e.g. a counter which can expand indefinitely, can be formally verified in the resulting proof environment.

Then, value-passing CCS is mechanised in HOL by translating value-passing expressions into pure ones. This entails a more general polymorphic type for pure agent expressions that includes an indexed summation operator. The translation is proved to be *correct* with respect to the semantics of value-passing CCS and then used at *meta-level*, together with the HOL formalisation for pure CCS, for developing behavioural theories for the value-passing calculus. A proof environment is thus derived, in which users will directly work on value-passing specifications. A verification example illustrates how proofs about the data are neatly separated from proofs about the process behaviour and how *ω-data-rules* can be used in a practical way to reason about value-passing agents defined over an infinite value domain.

Contents

Chapter 1

Introduction

The top-down design of a complex system usually starts with an abstract specification of the task that the system is intended to achieve. This description is then transformed into a detailed implementation possibly through several specifications of the same system at different levels of abstraction. The correctness of the transformation steps can be guaranteed by verifying that the different descriptions of the system are equivalent when "uninteresting" details, e.g. internal communications between subsystems, are ignored, or that a low level description is a "satisfactory" implementation of a more abstract one.

Process calculi, such as ACP [9], CCS [95, 97], CSP [66], EPL [58], LOTOS [10], MEIJE [6] and π-calculus [99], are generally recognised to be a convenient tool for specifying concurrent and communicating systems at different levels of abstraction and for reasoning about them. These formalisms are based on an algebra of actions, i.e. a syntax with operators to construct actions and a set of laws giving their meaning, and a syntax for process specifications (or agent expressions) with operators to build them. The meaning of the process constructors is usually given through a structural operational semantics defined via labelled transition systems following Plotkin's SOS approach [110].[1]

Process calculi, also called *process algebras* due to their largely algebraic nature, are then equipped with one or more notions of *behavioural semantics* (such as bisimulations and behavioural equivalences/preorders) and with *modal/temporal logics*, which are defined and interpreted in terms of labelled transition systems. Behavioural semantics have also been characterised through sets of (in)equational laws. Examples in the literature are strong equivalence and observation (weak) equivalence/congruence [97], trace and testing equivalences/preorders [34] and branching bisimulation [45]. Suitable sets of laws have been proved to be correct and complete *axiomatisations* for the associated behavioural equivalences/preorders over subsets of the various process algebras.

The analysis of concurrent and communicating systems can be very delicate and

[1] Amongst the process calculi mentioned above, CSP is an exception to this.

error-prone. Mechanical support is thus essential, both to make the analysis feasible and to ensure correctness, even when dealing with moderate size systems. In the past few years, various approaches have been proposed for verifying concurrent and communicating systems, and several verification tools have been developed which work in the framework of process calculi [72, 88]. Most of these tools deal with *pure* process algebras, where process communication is just synchronisation; only a few of them are able to cope with *value-passing* calculi, where data are exchanged during communication.

In this dissertation, an interactive proof environment for supporting both reasoning about the process algebra CCS (*Calculus of Communicating Systems*) and verification of CCS agents[2] is developed using higher order logic and the general purpose theorem prover HOL [51]. The formal theory of the process calculus is embedded in the HOL system by encoding CCS agent expressions as a type in the HOL logic. Based on this mechanisation, operational and behavioural semantics and modal logics can be defined and their properties and laws derived in the HOL system. When formalising the CCS theory in HOL, the principle of a *purely definitional* mechanisation is adopted. This means that new objects are embedded in HOL using definition mechanisms which guarantee that no inconsistencies are introduced in the logic, and by deriving new facts from definitions and/or previously proved theorems by formal proof.

The resulting formalisation supports both reasoning about the process algebra itself (*meta-theoretic* reasoning) and verification strategies for CCS specifications based on mechanised formal proof. In the work described in this dissertation, the theorem proving methodology is applied to the CCS language, but it can be adopted to formalise any other process calculus. This approach enables one to deal with both the operational and algebraic/axiomatic components of a process calculus in a unified framework. Different versions and extensions of a calculus are also mechanised in the same logical environment. The HOL formalisation of both pure and value-passing versions of the CCS calculus is presented in the following chapters. Other versions of the calculus could be treated as well, such as the ones dealing with notions like time, probability, priority.

The following section contains a brief overview of the approaches and (some of the) tools for process algebra verification. Their main characteristics are recalled and discussed, thus serving as an introduction and a motivation for the approach taken in this dissertation to reasoning about process calculi. Besides the references reported in the paragraphs below, the reader is referred to the surveys in [72, 88] for a more detailed description, comparison and evaluation of the various verification tools.

[2]The process language under consideration is Milner's calculus presented in [97]. There, the author prefers to call his theory 'process calculus' rather than 'CCS' as in [95] or 'process algebra'. Nevertheless, in what follows the terms 'CCS' and 'process algebra' will often be (ab)used when referring to Milner's process calculus.

1.1 Background and Related Work

The correctness of process algebra specifications can be expressed and verified in different ways using the various semantics, logics and/or algebraic laws. One typical verification problem considers two different descriptions of the behaviour of a system, an abstract one usually referred to as a *specification* (*Spec*), and a more detailed one, referred to as an *implementation* (*Impl*). The aim is showing that the implementation is *correct* with respect to the specification or, in other words, that the implementation *meets* the specification (*behavioural verification*). This means proving that a certain behavioural relation R exists between the two descriptions, i.e. *Spec* R *Impl* where R can be either an equivalence/congruence or a preorder relation. Proving the equivalence/congruence of two specifications may be too strong a requirement, even if the behavioural semantics under consideration is weak. In this case, it might be sufficient to verify that *Spec* $\sqsubset$ *Impl* with respect to a given preorder relation $\sqsubset$, namely proving that the low level description performs everything which is also performed by the abstract one.

A different verification problem considers process logics. Given a description of a system and a modal/temporal formula providing a partial specification of that system through some property (such as capacity, safety, liveness, deadlock, etc.), the problem is to check whether or not the system specification has that modal/temporal property (*model checking*).

The behaviour of a process specification can also be investigated by running the operational semantics of the process calculus under consideration (*simulation*). Thus, the behaviour of a system is simulated by executing its transitions and compared with the behaviour of another specification of the same system. This alternative description can just be a trace of events and the aim is checking whether or not the trace occurs in the behaviour of the given system.

Other kinds of reasoning about process algebra specifications include *minimisation*, i.e. transforming a specification to its reduced form with no redundant transitions with respect to a given behavioural relation, and *equation solving*, i.e. a refinement strategy which tries to build the missing part in the implementation of a system, when given its abstract specification and the part already implemented.

The implementation methods of process algebra verification can be roughly divided into two classes. The first class consists of *automata based approaches*, where specifications are analysed by (wholly or partially) constructing and exploring the associated transition systems. Efficient decision procedures from automata theory based on partitioning algorithms [105] can then be used to automatically minimise specifications, verify behavioural equivalences and show that a specification has a given property. The second class of verification methods includes *theorem proving based approaches*, which work on the syntactic representation of specifications without resorting to any other internal representation. These methods largely rely on the

algebraic nature of the process calculi and support symbolic manipulation, equational reasoning and powerful proof techniques, like induction, in a more interactive way.

1.1.1 Automata Based Tools

The Concurrency Workbench

The Concurrency Workbench [30] is a verification tool for CCS agents, where different verification techniques and styles can be combined together in a flexible and modular way. It supports the verification of a variety of behavioural equivalences/preorders, including strong equivalence, observation equivalence/congruence, testing equivalences/preorders and branching bisimulation. There is only one routine for verifying equivalences, based on bisimulation equivalence [97], and only one routine for checking preorders, based on the divergence preorder [123]. The various equivalences/preorders are verified by combining these general routines with suitable process transformations. Moreover, automata can be minimised with respect to observation equivalence and, when generating the automata representing CCS agents, term rewriting techniques are used for reducing their state space, so that automata can sometimes be made finite rather than infinite.

The Concurrency Workbench also supports model checking in a very expressive temporal logic, (a version of) the *propositional μ-calculus* [78, 118]. Other temporal logics can be introduced by defining new constructors in terms of existing ones using a *macro* mechanism.

Another facility provided by the Concurrency Workbench is an algorithm for synthesising solutions to equations of the form $(E \mid X)\backslash L = E'$, where E is a finite state agent that represents an incomplete implementation, X is an agent variable that stands for the missing part, L is a set of actions and E' is a finite state (deterministic) agent specification that the implementation is to satisfy. The solution is determined in a semi-automatic way by transforming equations into simpler ones.

The Concurrency Workbench has also been used for reasoning about value-passing CCS specifications under the assumption that the value domain is finite. In [19] the syntax and the operational semantics of a language for value-passing CCS is defined independently of pure CCS. This language is then implemented via a translation into pure CCS. Because the value domain is finite, the Concurrency Workbench can be used to perform verification on the (finite state) agents resulting from the translation.

AUTO

The AUTO system [14, 117] is a verification environment for MEIJE finite state processes. It includes computation of automata from MEIJE terms, reduction of automata with respect to several behavioural semantics and transformations of automata according to abstraction criteria. Strong and weak bisimulation congruences

are used to reduce the components of a system before composing them in parallel. Finiteness of MEIJE processes is obtained through a two-layer structure for input terms. At the lower layer, one defines processes (or automata) using the dynamic operators (i.e. action prefix, summation and recursion). At the upper layer, these processes are composed into networks using the static operators (i.e. parallel composition, relabelling and restriction).

The MAUTO system generalises AUTO to a large class of process algebras, which are compiled from the rules of their structural operational semantics. One application, developed inside the LITE toolset (see below), is an instantiation of MAUTO for the Basic LOTOS process algebra (i.e. LOTOS without abstract data types).

More efficient partitioning algorithms for checking behavioural equivalences are implemented in the FCTOOL system. A characteristic feature is its interface based on a *common format* for labelled transitions systems, which allows easy use and interaction with other tools.

The AUTO package is also provided with a graphical editor, AUTOGRAPH, dealing with hierarchical networks of automata. It is able to produce input terms for AUTO and display automata generated by AUTO and MAUTO.

Finally, ECRINS is a parameterised tool for manipulating process algebra terms. A process calculus is described by its syntax and operational semantics. This description mechanism is general enough to allow the definition of most process calculi. Recursive definitions are provided through a built-in operator for recursion. A behaviour evaluator computes the semantics of closed and open terms via transition systems and sets of conditional rules respectively. The evaluator can be used interactively and controlled with tactic programs. Algebraic laws for strong bisimulation can be proved and their proofs mixed with axiomatic reasoning based on rewriting. ECRINS can also be seen as a transition system generator for AUTO, thus allowing AUTO to perform verification on a variety of process calculi.

EMC

Finite state concurrent systems can be automatically verified to meet logical specifications using the model checker EMC [28]. Concurrent systems are specified in a restricted subset of the CSP calculus and interpreted as Kripke structures, and logical specifications are expressed in the propositional branching time temporal logic CTL. The model checker EMC determines that a system meets its specification by checking whether the structure of the system is a model of the given CTL formula by means of a very efficient algorithm, the complexity of which is linear in both the size of the structure and the size of the logical specification. A counterexample facility is also provided: when the model checker determines that a formula is false, it will try to find a path in the structure for which the negation of the formula is true.

The EMC system has then been modified to encode a state graph by means of

BDD's (Binary Decision Diagrams [18]). This symbolic representation avoids the explicit construction of the global state graph, which is thus checked by a symbolic model checking algorithm. Whenever the BDD representation is able to capture some of the regularity in the state space, finite state systems with an extremely large number of states can be automatically verified [20]. A model checking algorithm for formulas of the μ-calculus has then been implemented in EMC, from which CTL model checking and verification of strong and observation equivalences can be derived.

Combining AUTO and EMC

In [33] a verification environment is presented which allows both behavioural and logical properties of concurrent systems to be verified by integrating the AUTO and EMC tools. The process logic used in this framework is ACTL, namely an action based version of the branching time logic CTL. The integration of the two tools is achieved by means of two translation functions: a *model translator* from the labelled transition systems built by AUTO into labelled state Kripke structures and a *logic translator* from ACTL to CTL. In this way, a process algebra specification is verified to have an ACTL property by checking the satisfiability of the CTL formula (resulting from the logic translation) using the EMC model checker on the Kripke structure corresponding to the transition system for the given specification.

TAV

The TAV system [47] is another verification tool for process algebra specifications, essentially CCS agent expressions. It provides both equivalence checking and model checking. Bisimulation equivalences are verified by computing the minimal bisimulation rather than the maximal one as the previously described verification tools do. The modal logic under consideration is Hennessy-Milner logic extended with recursion, thus resulting in a rather expressive language [81]. A distinctive feature of the TAV system is that explanations are always provided for the answers given to the user. For example, whenever two specifications are not bisimilar, a modal formula is returned which distinguishes them. Moreover, if a specification has a given modal property, a proof in the proof system in [81] is returned, otherwise a proof for the dual property can be obtained.

The TAV system has then been extended with *modal transition systems* to allow the refinement of a given specification to a single implementation [83]. The operational semantics of CCS is interpreted in terms of modal transition systems, a refinement ordering is defined by extending the notion of bisimilarity, and strong and weak refinements can be checked and constructed.

Finally, an equation solving mechanism is provided [84], which automatically synthesises solutions to systems of equations of the form $C_1(X) \sim E_1, \ldots, C_n(X) \sim E_n$, where C_i and E_i $(1 \leq i \leq n)$ are arbitrary finite state CCS contexts and agents respec-

tively, and X is an agent variable. The solution set is characterised by a (disjunctive) modal transition system.

Aldébaran

The Aldébaran tool [41] offers facilities for minimising and comparing transition systems modulo several behavioural equivalences and preorders. It also includes techniques, such as "on the fly" verification [42], for the comparison of transition systems without generating their global state spaces. Aldébaran can be used from within the CÆSAR system [44] for reasoning about LOTOS specifications, once the CÆSAR compiler has produced labelled transition systems from LOTOS terms.

MEC

The verification tool MEC [4] allows the construction and analysis of transition systems representing communicating processes. A global transition system is constructed as the *synchronised product* of the component processes, in which the states of the system are the tuples of the states of its components and the transitions are the tuples of the transitions of the components, provided that such transitions are allowed to be executed simultaneously. These systems can be automatically checked to have logical properties, which are defined in a language that can express most of the branching time temporal logic properties.

Squiggles

The tool Squiggles [12] allows one to verify behavioural equivalences of Basic LOTOS processes by implementing the partition algorithms in [105]. Finite state transition systems are built and minimised with respect to strong, weak and testing equivalence. LOTOS specifications can also be input directly in their graphical format and the CÆSAR compiler is used to translate processes into transition systems for Squiggles.

LITE

LITE (LOTOS Integrated Tool Environment) is a toolset resulting from the ESPRIT LotoSphere project [11]. In particular, the verification of LOTOS specifications is carried out by the LOLA tool [111]. In LOLA (possibly parameterised) LOTOS behaviour expressions can be analysed by expanding them and testing whether certain traces of events can occur (*property testing*). Behavioural equivalences and their laws are used to reduce the expansion process. Moreover, a *parameterised expansion* allows some full LOTOS specifications to be dealt with, by keeping variable definitions as such without expanding them over the value domain and identifying behaviours which are equal except for some value expressions.

FDR

FDR (Failures Divergence Refinement) [43] is a tool for proving properties of CSP programs. The theory of refinement in CSP allows correctness conditions, such as safety and liveness, to be encoded as the most non-deterministic process satisfying them. Whether an implementation *Impl* meets a specification *Spec* can be verified by checking that *Impl* refines *Spec*. A normalisation procedure transforms a CSP specification into a form such that the implementation can be checked against it by exploring the reachable states using model checking techniques. Thus, processes must be finite state and those failing the automatic checks can be investigated in an interactive way using debugging facilities.

SPIN

SPIN is a tool for checking the logical consistency of distributed system specifications, based on CSP and extended with some new powerful constructs [67]. A formal model is built using the input language PROMELA (Protocol Meta Language). Both synchronous and asynchronous communication can be specified, and concurrent processes can be created and deleted dynamically. The correctness of distributed systems can be expressed in a general linear time temporal logic.

The Mobility Workbench

The Mobility Workbench [122] is a tool for manipulating and analysing mobile concurrent systems specified in the π-calculus. Its basic functionality is to decide the (strong and weak) open bisimulation equivalences for agents with finite control (similar to finite state CCS agents) which do not admit parallel composition within recursively defined agents. The bisimulation algorithm is implemented with an "on the fly" technique. In fact, due to problems of name instantiation in the π-calculus, the bisimulation relation is constructed while generating the transitions systems of the two agents to be verified. The Mobility Workbench also provides facilities for simulating the behaviour of agents in an interactive way and for finding deadlocks.

Symbolic Verification Tools

An automata based approach to process algebra verification is very efficient and permits automatic checking of properties. However, this approach has a few problems and limitations. One problem is the well-known *state explosion*: the number of states of a concurrent system potentially increases exponentially in the number of its parallel components. Techniques to control state explosion have been proposed which are able to reduce the problem. These range from applying the laws for behavioural equivalences to minimise the state space of parallel subsystems before composing them, to performing verification while generating the transition systems. However, the state

space of a system often contains a number of computations that are hidden at the global level and thus irrelevant to its external behaviour. Global context conditions do not permit the construction of the minimal process representation by minimising the single components. In [29] global properties of a composite system can be deduced from local properties of its components using additional interface processes to model the environment of a component. However, state explosion is significantly reduced only in the case of loosely coupled systems. A similar method is used in [52] where state explosion is controlled by requiring the user to supply additional information, called *interface specifications*, on the way subsystems interact. The approach is based on the successive construction of minimal partially defined transition systems, guided by "guesses" of the interface specifications. The correctness of the method does not depend on the correctness of these specifications, but its efficiency does.

Many of the tools described above have been recently extended by means of *symbolic* verification algorithms, in which finite state graphs are represented using BDD's. These algorithms sometimes allow one to circumvent the state explosion problem, and very large and complex systems can be checked efficiently and automatically. Nevertheless, BDD's do not prevent state explosion in all cases and new methods have been proposed to combine them with other symbolic manipulation techniques [31]. One approach is to combine model checking with theorem proving: model checking is applied as much as possible and semi-automatic or interactive proof techniques are used when automatic verification is not feasible [68]. This means that axioms and rules of the given logic are formalised in the theorem prover and proofs are constructed by applying such rules. In [79] model checking is used to verify local properties of the components of a system and then theorem proving is applied to show that these properties imply the correctness of the composite system. Another hybrid approach is taken in [73] where an efficient implementation of a form of model checking, called *symbolic trajectory evaluation*, is integrated with the HOL theorem prover. This methodology allows one to (almost) automatically solve many hardware verification problems which could be very difficult in a pure theorem proving environment.

A limitation of automata based approaches is that they can deal with only finite state specifications. In such a framework, there is no easy way to accommodate the verification of infinite state processes and value-passing specifications (unless the value domain is finite, because value-passing agents can, in this case, be translated into finite state agents, e.g. [19]). Furthermore, even in the area of finite state specifications there are restrictions on the kind of verification proofs that one can perform. Recursively defined parameterised processes which are finite state are very common. Let the following specification describe the behaviour of a buffer of capacity $n \geq 1$ [97]:

$$
\begin{aligned}
Buffer_n(0) &\stackrel{\text{def}}{=} in.\, Buffer_n(1) \\
Buffer_n(k) &\stackrel{\text{def}}{=} in.\, Buffer_n(k{+}1) \;+\; \overline{out}.\, Buffer_n(k{-}1) \qquad (0 < k < n) \\
Buffer_n(n) &\stackrel{\text{def}}{=} \overline{out}.\, Buffer_n(n{-}1)
\end{aligned}
$$

Such a specification is parameterised on the capacity n and the number k of the data presently stored in the buffer. For any fixed n, the specification $Buffer_n(k)$ is a finite state process, thus model checking and equivalence verification can be carried out using an automata based tool. Given one of these tools, a table is typically provided that displays, for increasing values of n, the number of states of the automata representing the buffer specification and the performance of the tool. However, properties of specifications like the one above can be naturally verified using mathematical induction. General and powerful proof techniques such as induction, case analysis, contradiction, are not available in an automata based tool. Hence, it is not possible to formally prove in a rigorous way that the above specification satisfies a certain property "for all $n \in \mathbb{N}$", even though the theory behind process calculi supports such a kind of reasoning. Some induction techniques have been encoded in automata based tools, such as the SMV system which combines induction methods with symbolic model checking [90]. These techniques allow one to prove properties of a system which are independent of the number of its components. Typically, the user must provide a *process* or *network invariant* to serve as the inductive hypothesis and then the proof is checked automatically. Further work on using model checking techniques for reasoning about inductively defined systems can be found in [124] and in various papers in [31].

Finally, the approaches based on a finite state machine representation lack any insight into the meaning of the specifications. When performing verification, these tools usually deliver a yes/no answer and very frequently the answer is no; in this situation the user is not provided with any suggestion about what went wrong and in which part of the specification the error was located. Some of the above verification systems provide debugging facilities and, in particular, the TAV system is able to return explanations in terms of modal properties and their proofs. Nevertheless, the user might prefer to interact with the verification system and control the verification phase, even when dealing with finite state processes and specially when specifications with data are considered and the interaction between the data component and the process behaviour can be tricky. Apart from being necessary to address problems which are, in the general case, undecidable, a system that leaves some crucial decisions to the user can, for efficiency reasons, also be useful when tackling decidable problems. This methodology also has the advantage of providing a better understanding of the specifications and of the correctness criteria one is trying to verify.

1.1.2 Theorem Proving Based Tools

In the past few years, several investigations into verification environments based on the algebraic nature of the concurrency calculi have been carried out. This approach allows one to manipulate the specifications in a homogeneous way by working only on their symbolic representation. Various proof tools have been developed based on

equational reasoning. They include:

- (semi-)automatic rewriting based tools, in which term rewriting techniques are exploited to apply the (in)equational laws that characterise the behavioural equivalences/preorders;
- specially designed axiomatic tools in which the syntax of a process calculus and its behavioural laws are defined and then used to construct proofs;
- proof tools developed by embedding a process calculus in general purpose theorem provers.

Term Rewriting Techniques

The various behavioural equivalences for process algebras are characterised by sets of equational laws. These axiomatisations can be analysed by means of term rewriting techniques to see if these laws can be transformed into rewriting rules (i.e. oriented equations), while maintaining the same deduction power as in the original axiomatisation. In other words, this means to "complete" the set of laws into a term rewriting system which is *canonical*, i.e. *terminating* and *confluent*. Termination means that all rewriting sequences are finite and terminate in a term which is a *normal form* of the term from which rewriting started. Confluence means that any term has a *unique* normal form, namely all terminating rewriting sequences from a given term will terminate in the same term. Thus, a canonical term rewriting system, if it exists, provides a decision procedure for the corresponding behavioural equivalence. Completion procedures [37] are applied to derive a canonical term rewriting system from equational presentations. In the case of process algebras, these procedures are modulo AC (Associativity and Commutativity), as these two axioms occur among the laws that characterise the operator of binary summation.

In [2] a canonical term rewriting system is found for a version of the process algebra ACP with the laws of the branching bisimulation semantics (taken from [8]). Also in the case of finite CCS (no recursion), the axiomatic presentation of branching bisimulation can be completed into a canonical term rewriting system [35].

In [76] the application of term rewriting techniques to the verification of LOTOS specifications has been studied. The behavioural laws for several subsets of the LOTOS algebra are defined in the RRL system and completion procedures are applied to these sets of laws. A canonical set of rules is obtained only for a restricted subset of Basic LOTOS. When moving to larger subsets of specifications, the rule set becomes incomplete and strategies are needed to avoid non-termination of the rewriting sequences. Moreover, recursion cannot be handled in this setting and some control over the verification process is desirable. Thus, the behavioural laws are defined in PAM (see below) where recursion can be treated and interactive proofs can be performed. This equational approach is then extended to full LOTOS (i.e. with abstract data

types). Proofs of correctness for some full LOTOS specifications are carried out by hand using a combination of equational reasoning and bisimulation techniques. Such proofs are not formalised in any verification tool.

In [35, 36] an interactive system for reasoning about CCS and LOTOS specifications is implemented in Quintus Prolog by following an equational approach. Both the operational semantics and the laws for behavioural equivalences are seen as Horn theories and logic-functional techniques are used. The behaviour of a process can be explored by executing the rules of the operational semantics. The behavioural laws can be applied as rewriting rules for reducing processes to their normal forms both automatically and interactively to perform single transformations.

As mentioned above, term rewriting techniques alone appear to be quite restrictive and therefore it is not always possible to rely on a fully automatic verification. In fact, even in the simple case of finite processes, there are behavioural semantics, such as observation congruence and testing equivalence, whose axiomatic presentations do not admit an equivalent canonical term rewriting system. This is due to the fact that the completion of the behavioural laws *diverges*, namely it generates infinitely many new rewriting rules [65]. Moreover, as soon as the recursion operator is added to the syntax of the process algebra, the unfolding rule makes rewriting non-terminating and more difficult to treat [38]. Thus, rewriting strategies have to be developed to guide the rewriting sequences towards the normal forms and prevent non-termination [69, 70, 71]. A rather similar approach is also adopted in [1] for deciding the strong early equivalence between finite terms of the π-calculus. Terms are first reduced to a prefixed form by means of a rewriting strategy which applies the basic behavioural laws. The equivalence is then decided using a proof system derived from its operational definition based on bisimulation. Such a strategy is implemented as a tactic in the HOL theorem prover.

Hence, a system with a meta-language to facilitate programming additional strategies is needed. Furthermore, proofs by reduction are not the only verification technique, as one would like to be able to use operational means, induction, case analysis, contradiction, etc. A framework which provides more general and powerful proof techniques is thus required. In order to achieve this, specially designed proof tools can be developed or general purpose theorem provers can be used as the infrastructure to encode the theory of process algebras.

PAM

The parameterised proof tool PAM (Process Algebra Manipulator) is based on the axiomatic presentation of behavioural semantics [86]. A meta-language allows users to define their own process calculus and carry out proofs in it in an interactive way. Several concurrency calculi have been mechanised in PAM including ACP, CCS, CSP and LOTOS. A calculus is defined by entering its signature and the set of axioms which

characterise the behavioural semantics under consideration, without resorting to the operational semantics. The core of PAM is a rewriting machine and the axioms are applied using term rewriting techniques. Recursive agents are dealt with by means of folding/unfolding rules and unique fixpoint induction. Tactics are provided for combining simple transformation steps into more powerful ones. Help facilities and a window based user interface make proofs easier to carry out.

PAM has been extended to VPAM [87], which deals with value-passing specifications according to the theory of *symbolic bisimulation* [61]. VPAM is based on a proof system in which data and boolean expressions are treated symbolically. This means that, when value-passing agents are analysed, boolean and value expressions are not evaluated, and input variables are not instantiated. In this way, reasoning about data is separated from reasoning about agents and is performed by extracting proof obligations, which can be verified by another theorem prover later or on-line with the main proof about the process behaviour (see Chapter 4 for further details).

PSF

The axiomatic tool PSF is specifically implemented for reasoning about the PSF process algebra (i.e. an extension of the ACP algebra with abstract data types) within the PSF Toolkit [89]. Typically, the user interacts with the proof assistant PSF by specifying the axioms and tactics to be applied during the verification of behavioural equivalences.

Instead of building a theorem proving environment from scratch and specialised in reasoning about process algebras, one can think of embedding the process algebra theory in general purpose theorem provers. In this way, the features of the particular theorem prover are inherited, such as the type theoretic approach, the logic, the proof environment and proof techniques.

LOTOS and the Boyer-Moore Theorem Prover

In [5] the Boyer-Moore theorem prover is used to formalise Basic LOTOS specifications. The syntax, the transition rules of the operational semantics and the laws for observation equivalence are asserted in the theorem prover, and simple transitions and equivalences are automatically proved. No inductive proof mechanisms can be used because of the restrictions of the Boyer-Moore shell data type.

μCRL and the Coq system

The specification language μCRL combines processes with abstract data types [53] in a way similar to the PSF algebra. The process component is still based on ACP with branching bisimulation semantics (taken from [8]) and is extended with data that are defined through many sorted equational specifications. A proof theory has

been given to μCRL in a natural deduction style [54]. It includes axioms and rules that express equivalences over data and processes and formalise the relationship between the two components. The proof theory of μCRL has been implemented in the Coq system [114] by extensively using its inductive mechanisms. According to the algebraic approach to defining semantics, both the equations about data and the laws for branching bisimulation are directly asserted in Coq. A version of unique fixpoint induction over recursive processes, called Recursive Specification Principle, is formalised by means of a rule that, given two processes and a recursive equation, checks that the processes "satisfy" the equation and that such an equation is guarded. The correctness of several protocols has been verified using μCRL and its embedding in Coq [77].

To conclude this brief overview of verification tools for process algebras, it is worth stressing that the described approaches are not to be considered as opposed to each other or contradictory, rather they are complementary. As already mentioned, there is a lot of ongoing research on combining (automatic) model checking with (interactive) theorem proving. Moreover, new tools have been derived by interfacing an algebraic tool with an automata based one, thus allowing different verification styles to be adopted in the resulting environment. This kind of interaction is restricted to those process algebra specifications for which the automata based tool can build finite state automata. Examples are the combination of PAM with the Concurrency Workbench and the ECRINS tool. For instance, the interface between PAM and the Concurrency Workbench allows PAM to pass parts of its proofs to the Concurrency Workbench, so that they can be checked automatically.

1.2 On Formalising a Process Calculus in the HOL Proof Assistant

In this thesis, higher order logic and the general purpose theorem prover HOL [51] are used to develop an interactive proof environment for supporting reasoning about process algebra theory and its applications. The aim is to build a verification system based on theorem proving which: (i) is logically sound, (ii) allows meta-reasoning to be carried out, (iii) is interactive but allows automation whenever possible, (iv) can be used in a practical and effective way.

Higher order logic is a good formalism for mechanising other mathematical languages because it is both powerful and general enough to allow sound and practical formulations. Several logics have been mechanised in higher order logic [50] and the theorem proving system HOL is used in these mechanisations. The HOL system was originally developed by Gordon [49] for reasoning about hardware systems, but in the past few years the range of its applications has widened considerably. The HOL proof

assistant is now used for mechanised theorem proving in many areas, including design and verification of critical and real-time systems, program refinement, program correctness, compiler verification and concurrency.

The HOL logic is a version of (classical) higher order logic based on Church's simple theory of types [26]. The interface to the logic is the functional programming language ML [32]. The approach to theorem proving in HOL is based on the LCF methodology [107] for interactive and secure theorem proving by mechanising the logic in a strongly typed language like ML. Theorems are represented by abstract data types and the user can interact with the theorem prover by ML procedures which operate on such data types. Theorem proving tools are ML functions and user-defined ML programs can only perform valid logical inferences. Typically, a purely definitional approach is taken when using the HOL logic. This means that new entities are only introduced by means of (primitive and derived) definition mechanisms which allow one to extend the logic in a sound way. Propositions and theorems are then derived from definitions and/or previously proved theorems by formal proof. This guarantees that inconsistencies are not introduced into the logic.

The HOL system can be used in two ways: (I) for proving theorems directly in the HOL logic whenever higher order logic is a suitable specification language, e.g. in hardware verification [48]; (II) as a theorem proving environment to support reasoning about other formalisms.

According to the former approach, a formal system (such as the theory for a process algebra) is mechanised in HOL by translating its syntactic objects into appropriate denotations in higher order logic. In this way, the embedded system inherits a certain amount of syntactic infrastructure from the HOL logic. For example, variable binding and substitution in a process algebra can be formalised using λ-abstraction and β-reduction in higher order logic. This approach allows one to build an environment suitable for reasoning about applications of the embedded system, but not for meta-reasoning about the embedded system itself. In fact, meta-theorems about the formalised system cannot be proved in the HOL logic.

The latter approach allows for both reasoning about applications and meta-reasoning about the mechanised system. The formalism to be represented in HOL is not translated into terms of higher order logic, but is encoded as a type in the logic. These types are objects within the logic which can be referred to, properties about them can be expressed and meta-theorems about the embedded language can be proved. In the case of a process algebra, this language is formalised in HOL as a type of agent expressions, to which meaning is then given by defining operational and behavioural semantics. This approach has been strictly followed in [93], where Milner's π-calculus is mechanised in the HOL system. In that formalisation all syntactic operations over agent expressions, such as substitution, are defined within the logic and then referred to explicitly in theorems and propositions involving such operations. The HOL mechanisation of the π-calculus in [1] uses the same methodology. Similar

approaches are adopted in [7] where notions of program refinement are formalised in HOL, and in [21, 22] where CSP trace and failure-divergence semantics are embedded in HOL and some of their laws are formally derived. However, in these works substitution and other syntactic operations are not defined explicitly, but inherited from higher order logic.

The approach taken in this dissertation is similar to the ones above, as it is mainly the type based one. However, some operators and syntactic operations of the embedded calculus will be mechanised by translating them into HOL terms and rules. In the initial stage of the described research, the pure subset of the CCS process calculus is considered. The formal theory for some behavioural semantics, namely strong equivalence, observation equivalence and congruence, and for a slight extension of Hennessy-Milner modal logic is embedded in HOL.[3] The resulting formalisation is the basis for the definition of verification strategies by mechanised formal proof. These include strategies that exhibit different degrees of user interaction depending on the subsets of CCS under consideration [23], proofs of correctness by mathematical induction for parameterised specifications [100] and verification of modal properties [101].

The formalisation of the pure calculus in HOL is given in two steps, hoping that this will make it easier to understand. First, pure CCS with binary summation, *rec*-notation and a string based encoding of actions is embedded in HOL. The labelled transitions, the operational and axiomatic characterisations of behavioural semantics and a modal logic are mechanised in HOL, and proof tools are defined to allow reasoning to be performed on agent specifications. Next, *rec*-notation is replaced by systems of recursive equations, actions can be of any type and the more general formulation of the CCS syntax with indexed summation is introduced. This will result in a polymorphic version of the type for pure agent expressions.

The process calculus is then extended to its value-passing version. The approach taken to formalising value-passing CCS is based on Milner's translation from the value-passing calculus to its pure subset [97]. A type for value-passing agent expressions is defined in the HOL logic, such that an input prefixed agent $a(x).E$ is encoded through a λ-abstraction that binds the variable x in the expression E. Thus, the HOL λ-abstraction is used to formalise variable binding and the HOL β-reduction implements the substitution of values for variables in value-passing agents. The translation is then defined and proved to be *correct* with respect to the semantics of value-passing CCS, in the sense that the translation *preserves* all the transitions that a value-passing expression can perform.

A proof environment for the value-passing calculus is derived using the translation and the HOL formalisation for pure CCS, thus avoiding redoing all the proofs of the properties of operators and of the behavioural laws. In fact, these are proved by simply translating the value-passing expressions into their pure versions, and then using

[3] In the following, the term 'HOL' will be used to denote both the version of higher order logic implemented in the HOL system and the theorem proving environment.

the corresponding results already proved for pure CCS. In the resulting environment, reasoning about value-passing specifications is performed without translating them into their pure versions. Moreover, the proofs about the data component can be neatly separated from the proofs about the process behaviour in the spirit of what advocated by Hennessy in [59].

One goal of the formalisation of a process calculus in HOL is to allow meta-reasoning to be performed on the embedded calculus. The translation from value-passing CCS to its pure subset is an example of the meta-reasoning that can be carried out in the HOL-CCS environment. The translation is used at *meta-level* for developing behavioural theories for the value-passing calculus, and properties about the relationship between pure and value-passing calculi can be formally proved.

The formalisation of the value-passing calculus is parameterised on the data component and is carried out by keeping such a language as general as possible. This means that no knowledge about the value domain V and its particular structure is assumed. The idea is to investigate what can be obtained by formalising V as a polymorphic set, that can then be tailored to the particular application by properly instantiating the type variable for the data. If the value domain is known through its constants and constructors, it can be defined as a type in higher order logic and substitution of value constants for value variables can be recursively defined on the structure of the value domain. When the value domain is instead given as a polymorphic set, HOL infrastructure such as λ-abstraction and β-reduction can be used to mechanise variable binding and substitution.

Finally, note that a parallel and independent formalisation of pure CCS in HOL has been carried out in [112], where CCS is used for describing and analysing the behaviour of hardware systems. In that preliminary work, the relabelling and recursion operators are not included in the CCS syntax and the parallel composition operator is replaced by other constructors inspired by the ACP algebra. Both operational and observation semantics are defined as inductively defined relations and then some simple laws are formally derived.

1.3 Outline of the Dissertation

The dissertation is organised as follows.

- Chapter 2 introduces the syntax and the operational semantics of pure CCS, followed by the theory of the strong and observation semantics, the axiomatic characterisation for observation congruence and the Hennessy-Milner logic. All these components are then formalised in HOL. The version of pure CCS under consideration includes binary summation and *rec*-notation, and the HOL mechanisation assumes that labels are represented by strings of characters. Proof tools and techniques are then developed to help reasoning about CCS specifi-

cations. The resulting proof environment is used to verify the correctness of a simple scheduler, namely a reader-writer system, in two ways. First, two different descriptions of the reader-writer system are proved to be observation congruent. Second, it is formally checked that the reader-writer system has a mutual exclusion property.

- Chapter 3 extends the syntax of pure CCS. Agent constants and defining equations are introduced to replace the *rec*-notation. This new notation allows (mutually recursive) infinite state systems to be specified in a very convenient and readable way. As a verification example, a specification of an infinite counter is proved to be observation congruent to a low level implementation and to satisfy a certain modal property. The main correctness results and most intermediate theorems in this example are proved by mathematical induction. The syntax of pure CCS is then extended to include an operator of (possibly infinite) indexed summation. This leads to the use of function types in the definition of the recursive type for CCS expressions. Pure actions are made polymorphic and the formalisation of the expansion law for the parallel composition operator is revisited and mechanised in terms of indexed summations.

- Chapter 4 extends the HOL formalisation to the value-passing calculus. The syntax of value-passing CCS is mechanised in HOL and then the translation from value-passing agents to pure agents is defined. The translation is proved to preserve the transitions that a value-passing agent expression can perform. Next, a proof environment is developed for value-passing CCS based on the translation and the proof environment for pure CCS. Reasoning about value-passing specifications in HOL is shown by proving the observation congruence between two different specifications of a simple communicating system.

- Chapter 5 provides a summary of the work described in this dissertation and its main contributions. Some directions and ideas for further research are also proposed.

- Appendix A contains a brief introduction to higher order logic and the HOL system. The overview is not meant to be complete. The aim of this appendix is to recall only the information essential for understanding the HOL formalisation presented in the earlier chapters and the HOL sessions given in the following appendices.

- Appendices B, C and D present the HOL code corresponding to the proofs of correctness which have been described in the previous chapters in a more informal way. Further comments about such proofs and their HOL mechanisation are also provided.

Chapter 2

Pure CCS in HOL

This chapter describes a pure process algebra and its mechanisation in the HOL system. In order to make the presentation of the HOL formalisation easier, a version of the pure CCS calculus is introduced which is slightly different from the one given in [97]. The more general formulation of the pure syntax will be mechanised in Chapter 3, thus obtaining in the end a HOL formalisation of Milner's calculus.

In the following sections, several definitions and theorems about CCS will be recalled. The HOL formalisation of some of them, namely HOL definitions and theorems, usually equipped with the name under which they are stored in the HOL theories, will be presented. Excerpts of HOL sessions will be enclosed in boxes. In order to help readability, HOL definitions, theorems and transcripts will be edited to show proper logical symbols instead of their ASCII representations, e.g. the usual symbol for universal quantification '$\forall$' will be used instead of the HOL symbol '!'. Moreover, CCS expressions will sometimes be parsed and pretty-printed (modulo ASCII syntax, e.g. $\overline{a}$ will be written $-a$ in boxes) so that the notation normally associated with CCS can be adopted instead of its HOL representation.[1]

2.1 Pure CCS

The essential information about the syntax, the operational and behavioural semantics and Hennessy-Milner logic for pure CCS is recalled in this section. The reader is referred to [64, 97, 98] for more details about the calculus.

2.1.1 The Syntax and Operational Semantics

Pure CCS is a subset of the CCS language which does not involve value passing. The communication between agents is simply synchronisation and no values, or more

[1]Note that a parser and a pretty-printer for the CCS notation have not yet been implemented in the HOL system. Parsing and pretty-printing are done manually, thus all misprints, if any, are only the author's fault.

generally data, are exchanged. The calculus consists of the inactive agent **nil** (called **0** in [97, 98]), a countable set $\mathcal{X}$ of *agent variables* ranged over by X, and the following operations on agent expressions: *prefix* (.), *binary summation* (+), *parallel composition* (|), *restriction* (\), *relabelling* ([]) and *recursion* (**rec**). The syntax of pure agent expressions $\mathcal{E}$, ranged over by E, E', ..., is as follows:[2]

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

where

- u ranges over actions, which are either labels (visible actions, also referred to as *ports*) or the silent (or invisible) action τ.
- L is a subset of labels ranged over by l. 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 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 $f(\tau) = \tau$.

In the expression $\mathbf{rec}\, X.\, E$ the agent variable X is *bound* by the operator **rec**. An occurrence of the agent variable X is *free* in E if it is not bound by any operator **rec**. For any expression E, $Fv(E)$ denotes the set of free variables in E. An agent expression E is *closed* if $Fv(E) = \emptyset$, otherwise E is an *open* agent expression. Closed agent expressions are called *agents*. The set of agents is denoted by $\mathcal{P}$ and ranged over by P, P', Q, etc.

An agent variable X is *guarded* in E if every free occurrence of X in E occurs within some subexpression $l.E'$ of E. An agent variable X is *sequential* in E if every subexpression of E which contains X, apart from X itself, is of the form $u.E'$ or $E_1 + E_2$. An agent expression E is *finite* if it contains no recursion and is *sequential* if it contains no parallel composition, restriction and relabelling.

Let $\tilde{E}$ denote the set of agent expressions $\{E_i : i \in I\}$ for some indexing set I. Then $E\{\tilde{E}/\tilde{X}\}$ denotes the simultaneous substitution of E_i for all free occurrences of agent variable X_i in the expression E (with variable renaming where necessary).

The meaning of the operators is the following.

- The agent **nil** cannot perform any action (*inaction*).
- The agent expression $u.E$ can perform the action u and then behaves like E.
- The agent expression $E + E'$ behaves like either E or E'.

[2] An equivalent notation for the recursion operator is **fix** and recursive expressions are then written $\mathbf{fix}(X = E)$ and, more generally, $\mathbf{fix}_j(\{X_i = E_i : i \in I\})$ where I is an indexing set and $j \in I$.

- The agent expression $E \mid E'$ can perform the actions of E and E' in parallel; moreover, E and E' can synchronise through the action τ whenever they are able to perform complementary actions.
- The agent expression $E\backslash L$ 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 expression E via the relabelling f.
- The expression $\mathbf{rec}\, X.\, E$ denotes a recursive agent which is a "solution" of the recursive agent equation $X = E$.

The operational semantics of the above operators is given via *labelled interleaving transitions* $\xrightarrow{u}$ over agent expressions based on their structure. This means that, in the inference rules defining the operational semantics, the transitions of a composite expression are defined in terms of the transitions of its component expression(s). There is no transition rule for the inactive agent, as it cannot perform any action. The meaning of the other operators is given through one or more transition rules, each one with zero or more hypotheses (and possibly side conditions) and one conclusion. The transition relation $E \xrightarrow{u} E'$ is inductively defined by the rules in Figure 2.1.

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

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

$$\text{PAR1:}\quad \frac{E \xrightarrow{u} E'}{E \mid F \xrightarrow{u} E' \mid F} \qquad \text{PAR2:}\quad \frac{E \xrightarrow{u} E'}{F \mid E \xrightarrow{u} F \mid E'} \qquad \text{PAR3:}\quad \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\overline{l}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\text{RESTR:}\quad \frac{E \xrightarrow{u} E'}{E\backslash L \xrightarrow{u} E'\backslash L} \quad u, \overline{u} \notin L \qquad \text{RELAB:}\quad \frac{E \xrightarrow{u} E'}{E[f] \xrightarrow{f(u)} E'[f]}$$

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

Figure 2.1: The transition rules for pure CCS.

The last rule asserts that the actions of a recursive expression $\mathbf{rec}\, X.\, E$ are all the actions of its *unfolding* $E\{\mathbf{rec}\, X.\, E/X\}$.

The above calculus differs from the one described in [97] in two ways. First, the *rec*-notation instead of agent constants and defining equations is used for denoting processes with infinite behaviour. Recursive agent definitions through agent constants and defining equations and their mechanisation in HOL will be illustrated in Section 3.1. Second, the inactive agent **nil** and the binary summation '+' are introduced as basic operators of the calculus. Actually, they are instances of the more gen-

eral form of the summation operator, namely indexed summation, presented in [97]. Indexed summation and its formalisation in HOL will be described in Section 3.3.2.

2.1.2 The Strong and Observation Semantics

In the literature several behavioural semantics have been defined for CCS. Examples are strong equivalence and observation equivalence/congruence [64, 97], trace and testing equivalences [34] and branching bisimulation [45]. Each of these semantics has been characterised in terms of axiomatisations, i.e. collections of equational laws for the CCS operators, 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, the strong and observation semantics are considered and the relevant definitions are briefly recalled.

Both strong and observation equivalence are defined in terms of a *bisimulation* relation between agents. A binary relation $\mathbf{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a *strong bisimulation* if for all P, Q, $P\,\mathbf{S}\,Q$ implies that for all actions u

(i) whenever $P \xrightarrow{u} P'$, then for some Q', $Q \xrightarrow{u} Q'$ and $P'\,\mathbf{S}\,Q'$;
(ii) whenever $Q \xrightarrow{u} Q'$, then for some P', $P \xrightarrow{u} P'$ and $P'\,\mathbf{S}\,Q'$.

Thus, a strong bisimulation contains those pairs of agents such that any action of the first agent can also be performed by the other agent and leads to agents which are strongly bisimilar as well, and vice versa. Note that the silent action τ does not have any special status. P and Q are defined to be *strongly equivalent*, written $P \sim Q$, if and only if $P\,\mathbf{S}\,Q$ for some strong bisimulation $\mathbf{S}$. In other words, strong equivalence $\sim$ is the union of all strong bisimulations.

Weaker equivalences in which (some of) the occurrences of the τ-action can be ignored are observation equivalence and congruence. In order to define them, the *weak transition* relation $\stackrel{s}{\Longrightarrow}$ for any sequence s of actions is first introduced. 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 $E\,(\xrightarrow{\tau})^*\,F$.

A binary relation $\mathbf{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a *weak bisimulation* if for all P, Q, $P\,\mathbf{S}\,Q$ implies that for all actions $u \neq \tau$

(i) whenever $P \xrightarrow{u} P'$, then for some Q', $Q \stackrel{u}{\Longrightarrow} Q'$ and $P'\,\mathbf{S}\,Q'$;
(ii) whenever $Q \xrightarrow{u} Q'$, then for some P', $P \stackrel{u}{\Longrightarrow} P'$ and $P'\,\mathbf{S}\,Q'$;

and

(i) whenever $P \xrightarrow{\tau} P'$, then for some Q', $Q \stackrel{\epsilon}{\Longrightarrow} Q'$ and $P'\,\mathbf{S}\,Q'$;
(ii) whenever $Q \xrightarrow{\tau} Q'$, then for some P', $P \stackrel{\epsilon}{\Longrightarrow} P'$ and $P'\,\mathbf{S}\,Q'$.

Note that any τ-action of an agent need not be matched by a τ-action of the other agent. P and Q are defined to be *observation equivalent*, written $P \approx Q$, if and only if $P \mathbf{S} Q$ for some weak bisimulation $\mathbf{S}$. In other words, observation equivalence $\approx$ is the union of all weak bisimulations.

Hence, in order to prove the strong (observation) equivalence of two agents P and Q, it is sufficient to show that there exists a strong (weak) bisimulation which contains the pair (P, Q). An alternative approach is to use equational reasoning: given a collection of equational laws which are known to hold of the behavioural semantics under consideration, the equivalence between P and Q can be proved by applying these laws using the principle of "substituting equals for equals". This is possible if the behavioural equivalence is a *congruence*, where a congruence over CCS agents is a relation which is preserved by all CCS operators. Strong equivalence is indeed a congruence, while observation equivalence turns out not to be a congruence relation as it is not preserved by summation contexts. Nevertheless, observation equivalence can be refined to a congruence relation by defining that two agents P and Q are *observation congruent*, written $P \approx_c Q$,[3] if for all actions u

(i) whenever $P \xrightarrow{u} P'$, then for some Q', $Q \stackrel{u}{\Longrightarrow} Q'$ and $P' \approx Q'$;
(ii) whenever $Q \xrightarrow{u} Q'$, then for some P', $P \stackrel{u}{\Longrightarrow} P'$ and $P' \approx Q'$.

This means that every initial 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. Observation congruence is preserved by all CCS operators and its axiomatic presentation allows one to perform equational reasoning on agents by substituting equals for equals. Note that observation equivalence is preserved by the summation operator under the hypothesis of stability. An agent P is *stable* if P has no τ-derivative, namely P cannot perform any silent action. If $P \approx Q$ and P and Q are both stable, then $P \approx_c Q$. Thus, under the hypothesis of stability the observation equivalence is not only preserved but even strengthened.

Strong congruence, observation equivalence and observation congruence are then extended to agent expressions. Let $\tilde{X}$ be the set of free agent variables which occur in agent expressions E and F. Then $E \approx_c F$ if and only if $E\{\tilde{P}/\tilde{X}\} \approx_c F\{\tilde{P}/\tilde{X}\}$ for all agents $\tilde{P}$. Strong congruence and observation equivalence over agent expressions are defined in a similar way.

2.1.3 The Axiomatic Characterisation of Observation Congruence

The axiomatic presentations which characterise the behavioural equivalences for CCS can be separated into two sets of algebraic laws: those common to all equivalences,

[3] In this dissertation, the usual notation '=' for observation congruence is replaced by '$\approx_c$' (as used in [64]) in order to avoid confusion with the HOL equality relation.

referred to as *basic laws*, and those concerning the silent action τ which distinguish the various equivalences, referred to as *τ-laws*.

By means of the basic laws, any finite agent can be proved equivalent to a sequential one, i.e. an agent containing only **nil**, prefix and summation operators. The basic laws (here given for observation congruence) are shown below:

$$P + (Q + R) \approx_c (P + Q) + R \quad \text{(A1)}$$
$$P + Q \approx_c Q + P \quad \text{(A2)}$$
$$P + P \approx_c P \quad \text{(A3)}$$
$$P + \mathbf{nil} \approx_c P \quad \text{(A4)}$$

$$\mathbf{nil}\backslash L \approx_c \mathbf{nil} \quad \text{(A5)}$$
$$(P + Q)\backslash L \approx_c P\backslash L + Q\backslash L \quad \text{(A6)}$$
$$(u.P)\backslash L \approx_c u.(P\backslash L) \qquad \text{if } u, \overline{u} \notin L, \mathbf{nil} \text{ otherwise} \quad \text{(A7)}$$

$$\mathbf{nil}[f] \approx_c \mathbf{nil} \quad \text{(A8)}$$
$$(P + Q)[f] \approx_c P[f] + Q[f] \quad \text{(A9)}$$
$$(u.P)[f] \approx_c f(u).P[f] \quad \text{(A10)}$$

$$\text{If } P \equiv \sum_{i=1}^{n} u_i.P_i \text{ and } Q \equiv \sum_{j=1}^{m} v_j.Q_j \text{ then} \quad \text{(A11)}$$
$$P \mid Q \approx_c \sum_{i=1}^{n} u_i.(P_i \mid Q) + \sum_{j=1}^{m} v_j.(P \mid Q_j) + \sum \{\tau.(P_i \mid Q_j) : u_i = \overline{v_j}\}$$

In the following, (A11) will be referred to as the *expansion law*. This law can be combined together with the laws for the relabelling and restriction operators to get what will be referred to as the *expansion theorem*.[4] The τ-laws for the observation congruence are the following:

$$u.\tau.P \approx_c u.P \quad \text{(T1)}$$
$$P + \tau.P \approx_c \tau.P \quad \text{(T2)}$$
$$u.(P + \tau.Q) + u.Q \approx_c u.(P + \tau.Q) \quad \text{(T3)}$$

The axiomatic theory of observation congruence for finite pure CCS is given by the laws (A1)–(A11) and (T1)–(T3). These laws have been proved correct and complete with respect to the definition of observation congruence in [64, 97].

The recursion operator is characterised by the following laws:

$$\mathbf{rec}\,X.E \approx_c E\{\mathbf{rec}\,X.E/X\} \quad \text{(A12)}$$
Let $Fv(E) \subseteq \{X\}$ and X be guarded and sequential in E. (A13)
If $P \approx_c E\{P/X\}$ then $P \approx_c \mathbf{rec}\,X.E$.

[4]Note that the expression "expansion law" is used in [97] to refer to what is called "expansion theorem" here (following the notation in [95]), in which relabelling and restriction may also occur, besides parallel composition. In this dissertation, the expansion law is just the law for the parallel operator, which can be derived as a particular instance of the expansion theorem. Furthermore, note that there exist different formulations of the law (A11) in the literature. Above, this law is recalled as presented in [95, 64]; another formulation, taken from [97], will be described and embedded in HOL in Section 3.4.

The unfolding law (A12) asserts that a recursive expression $\mathbf{rec}\, X.\, E$ is observation congruent to the expression $E\{\mathbf{rec}\, X.\, E/X\}$ obtained by "unwinding" $\mathbf{rec}\, X.\, E$ once. In other words, $\mathbf{rec}\, X.\, E$ is a solution (or a fixed point) of the recursive equation $X = E$ up to observation congruence. The law (A13) expresses the uniqueness of such a solution. In order to prove that an agent P is observation congruent to $\mathbf{rec}\, X.\, E$, it is sufficient to show that P "satisfies" the recursive definition, that is P is observation congruent to the agent obtained by replacing every free occurrence of X in E with P itself. The conditions of guardedness and sequentiality of X ensure the uniqueness of fixed points modulo observation congruence.

Given the subset of finite state sequential agents, the recursion laws (A12)–(A13) together with (A1)–(A4) and the τ-laws (T1)–(T3) have been proved sound and complete for observation congruence in [96]. In that paper, a further set of laws is also given for transforming unguarded recursive expressions to guarded ones in a sound and complete way.

2.1.4 A Modal Logic

The modal logic under consideration is a slight extension of Hennessy-Milner logic [64] as presented in [118]. Its formulas, ranged over by Φ, are defined by the following syntax:[5]

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

where A ranges over sets of actions. The meaning of the first three formulas is familiar: $\mathbf{tt}$ denotes the constant true formula, $\neg\Phi$ is a negated formula and $\Phi_1 \wedge \Phi_2$ is a conjunction of formulas. The modalised 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. This logic is a slight extension of Hennessy-Milner logic [64] in the sense that modalities are parameterised on a set of actions instead of a single action.

Modal logics are interpreted on labelled transition systems. For any formula Φ of the logic it is defined when an agent P has (or "satisfies") the property Φ. The notation $P \models \Phi$ is used to mean "P satisfies Φ", and $P \not\models \Phi$ to mean "P fails to have the property Φ". The *satisfaction* relation $\models$ is inductively defined on the structure of formulas:

$$\begin{array}{lll} P \models \mathbf{tt} & & \\ P \models \neg\Phi & \text{iff} & P \not\models \Phi \\ P \models \Phi_1 \wedge \Phi_2 & \text{iff} & P \models \Phi_1 \text{ and } P \models \Phi_2 \\ P \models [A]\Phi & \text{iff} & \text{for all } P' \text{ and } u \in A, \text{ if } P \xrightarrow{u} P' \text{ then } P' \models \Phi \end{array}$$

[5] The usual notation, also adopted in [118], for the modal operators, that is $\neg$, $\wedge$, $\vee$, $[A]$ and $\langle A\rangle$, is here replaced by a *bold* version $\neg$, $\wedge$, $\vee$, $[A]$ and $\langle A\rangle$ respectively. This is done to avoid confusion with the HOL logical connectives $\neg$, $\wedge$ and $\vee$, the CCS relabelling operator $[f]$ and the HOL list constructor $[e_1; \ldots; e_n]$.

Every agent has the property $\mathbf{tt}$. 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. 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:

$$\begin{array}{lcl} \mathbf{ff} & = & \neg\,\mathbf{tt} \\ \Phi_1 \vee \Phi_2 & = & \neg\,(\neg\,\Phi_1 \wedge \neg\,\Phi_2) \\ \langle A \rangle \Phi & = & \neg\,[A]\neg\,\Phi \end{array}$$

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

$$P \models \langle A \rangle \Phi \quad \text{iff} \quad \text{for some } P' \text{ and } u \in A,\ P \xrightarrow{u} P' \text{ and } P' \models \Phi$$

Properties such as *capacity* and *necessity* can be expressed within Hennessy-Milner logic. The modal formula $\langle A \rangle \mathbf{tt}$ expresses a capacity to perform an action in A, since:

$$P \models \langle A \rangle \mathbf{tt} \quad \text{iff} \quad \text{for some } P' \text{ and } u \in A,\ P \xrightarrow{u} P'$$

and $[A]\mathbf{ff}$ 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 P must perform a given action u (necessity) can be expressed as follows:

$$P \models \langle - \rangle \mathbf{tt} \wedge [-u]\mathbf{ff}$$

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

2.2 Mechanisation of Pure CCS in HOL

This section describes the formalisation in HOL of the various components of the CCS theory presented above.

2.2.1 The Syntax

The first step in the mechanisation of the calculus is the definition of the types for labels and actions. A label (or port) can be seen as the name of a visible action, so labels can be assumed to be strings of characters. The type for labels has then a certain structure based on whether labels are complemented (co-names) or not (names). Actions are either labels or the constant τ. The syntactic types *label* and *action* can thus be defined as the following concrete data types:

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

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

using the derived rule for (recursive) type definitions (Section A.4). Given the above specifications, this rule automatically derives a theorem of higher order logic for each type being defined, which characterises the type in a complete and abstract way. The theorems for the types *label* and *action* are the following:

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

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

As these types are not recursive, these theorems simply assert the admissibility of defining functions over labels and actions by cases.

The notion of complement is 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*, the property that $\bar{\bar{l}} = l$ for any l can be proved, thus obtaining the following theorem in HOL:

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

The type *relabelling* for relabelling functions is defined as the subset of functions of type $label \rightarrow label$ such that relabelling respects complements. This is obtained using the type definition mechanism in HOL (Section A.3), but the details of this formalisation are not presented here. A relabelling function is then extended to actions by defining a function relabel such that τ is renamed as τ:

$$\begin{array}{l}(\forall rf.\ \mathsf{relabel}\ rf\ \mathsf{tau} = \mathsf{tau})\ \wedge \\ (\forall rf\ l.\ \mathsf{relabel}\ rf\ (\mathsf{label}\ l) = \mathsf{label}\,(\mathsf{REP\_Relabelling}\ rf\ l))\end{array}$$

where REP_Relabelling is the representation function for the new type *relabelling*. The possibility of defining a relabelling function with the usual substitution-like notation $[l'_1/l_1, \ldots, l'_n/l_n]$ where l'_j renames l_j $(1 \leq j \leq n)$, is also provided. A constructor $\mathsf{RELAB} : (label \times label)list \rightarrow relabelling$ is defined which, given a list of pairs of labels, returns the associated relabelling function. For example, the relabelling $[u'/u, a'/a, d'/d]$ can be written in HOL as the term

$$\mathsf{RELAB}\ [\mathsf{name}\ u',\ \mathsf{name}\ u;\ \mathsf{name}\ a',\ \mathsf{name}\ a;\ \mathsf{name}\ d',\ \mathsf{name}\ d]$$

of type *relabelling*, where u', u, a', a, d', d are all strings.

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

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

Agent variables, whether bound or free, are formalised as strings denoting "placeholders" which can be replaced by agent expressions. Similarly to the types *label* and *action*, a complete characterisation of the type *CCS* is automatically derived:

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

This theorem of higher order logic is the basis for reasoning about the type *CCS*. For example, many syntactic notions can be defined by primitive recursion over the type *CCS*. As recalled in Section 2.1.2, weak bisimulation, observation equivalence and observation congruence are first defined over agents, i.e. agent expressions with no free agent variables, and then extended to agent expressions. A function $\mathsf{Fv}: \mathit{CCS} \to (\mathit{CCS})\mathit{set}$ that returns the set of free agent variables which occur in a given agent expression, is defined in HOL by primitive recursion over the type *CCS*. Agents are then characterised as those expressions whose set of free agent variables is empty, as asserted by the predicate Is_Agent:

$$\forall E.\ \mathsf{Is\_Agent}\ E\ =\ (\mathsf{Fv}\ E = \emptyset)$$

In particular, a recursive expression $\mathbf{rec}\,X.\ E$ is closed if its body E has (at most) the free variable X:

$$\text{Is\_Agent\_rec:}\ \vdash \forall X\ E.\ \mathsf{Is\_Agent}\,(\mathsf{rec}\ X\ E)\ =\ (\mathsf{Fv}\ E)\ \subseteq\ \{\mathsf{var}\ X\}$$

The conversion `Is_Agent_CONV` has been defined in HOL for deriving whether or not an agent expression is an agent.

Another example is the function CCS_Subst which, given two expressions E and E' and a string X denoting an agent variable, implements the substitution $E\{E'/X\}$ of E' for all free occurrences of the variable X in E. Such a function is defined through primitive recursion over the type *CCS*.[6] The following theorem (proved by structural induction over the type *CCS*) characterises the free variables of a substitution:

FVars_CCS_Subst:
$\vdash E\ E'\ X.\ (\mathsf{Fv}\,(\mathsf{CCS\_Subst}\ E\ E'\ X)) \subseteq (((\mathsf{Fv}\ E) - \{\mathsf{var}\ X\}) \cup (\mathsf{Fv}\ E'))$

From such a theorem a property can be derived, which will be useful when proving the unfolding law for the recursion operator:

Is_Agent_CCS_Subst_rec:
$\vdash \forall X\ E.\ \mathsf{Is\_Agent}\,(\mathsf{rec}\ X\ E) \supset \mathsf{Is\_Agent}\,(\mathsf{CCS\_Subst}\ E\,(\mathsf{rec}\ X\ E)\ X)$

Thus, given a closed recursive expression $\mathbf{rec}\,X.\,E$, the result of substituting such an expression for the free occurrences of X in its body E is an agent.

2.2.2 The Operational Semantics

Once the type *CCS* of agent expressions has been defined in HOL, the next step is the formalisation of the labelled transition relation which gives the operational meaning of the CCS operators. This relation can be embedded in HOL using the derived principle for inductively defined relations (Section A.4). A transition $E \xrightarrow{u} E'$ is represented by $\mathsf{Trans}\ E\ u\ E'$, where the relation $\mathsf{Trans}: \mathit{CCS} \to \mathit{action} \to \mathit{CCS} \to \mathit{bool}$ 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 Figure 2.1:

PREFIX: $\vdash \forall u\ E.\ \mathsf{Trans}\,(\mathsf{prefix}\ u\ E)\ u\ E$

SUM1: $\vdash \forall E\ u\ E1.\ \mathsf{Trans}\ E\ u\ E1 \supset (\forall E'.\ \mathsf{Trans}\,(\mathsf{sum}\ E\ E')\ u\ E1)$

SUM2: $\vdash \forall E\ u\ E1.\ \mathsf{Trans}\ E\ u\ E1 \supset (\forall E'.\ \mathsf{Trans}\,(\mathsf{sum}\ E'\ E)\ u\ E1)$

PAR1: $\vdash \forall E\ u\ E1.\ \mathsf{Trans}\ E\ u\ E1 \supset (\forall E'.\ \mathsf{Trans}\,(\mathsf{par}\ E\ E')\ u\ (\mathsf{par}\ E1\ E'))$

PAR2: $\vdash \forall E\ u\ E1.\ \mathsf{Trans}\ E\ u\ E1 \supset (\forall E'.\ \mathsf{Trans}\,(\mathsf{par}\ E'\ E)\ u\ (\mathsf{par}\ E'\ E1))$

PAR3: $\vdash \forall E\ E1\ E'\ E2.$
$(\exists l.\ \mathsf{Trans}\,E\,(\mathsf{label}\ l)\ E1 \wedge \mathsf{Trans}\,E'\,(\mathsf{label}(\mathsf{Compl}\ l))\ E2) \supset$
$\mathsf{Trans}\,(\mathsf{par}\ E\ E')\ \mathsf{tau}\,(\mathsf{par}\ E1\ E2)$

[6] Note that, in order to avoid capture of free variables, CCS_Subst works under the implicit (i.e. informal) assumption that no agent variable, which occurs bound in a recursive expression, also occurs free in any other agent expression (referred to as *Barendregt convention* in [120]). This aspect is instead treated formally in Melham's formalisation of the π-calculus [93], in which the possibility of free variable capture is dealt with explicitly.

RESTR: $\vdash \forall E\ u\ E'\ L.$
$(\exists l.\ \mathsf{Trans}\ E\ u\ E' \wedge$
$((u = \mathsf{tau}) \vee ((u = \mathsf{label}\ l) \wedge (l \notin L) \wedge (\mathsf{Compl}\ l \notin L)))) \supset$
$\mathsf{Trans}\ (\mathsf{restr}\ E\ L)\ u\ (\mathsf{restr}\ E'\ L)$

RELAB: $\vdash \forall E\ u\ E'.$
$\mathsf{Trans}\ E\ u\ E' \supset (\forall rf.\ \mathsf{Trans}\ (\mathsf{relab}\ E\ rf)\ (\mathsf{relabel}\ rf\ u)\ (\mathsf{relab}\ E'\ rf))$

REC: $\vdash \forall E\ X\ u\ E1.$
$\mathsf{Trans}\ (\mathsf{CCS\_Subst}\ E\ (\mathsf{rec}\ X\ E)\ X)\ u\ E1 \supset \mathsf{Trans}\ (\mathsf{rec}\ X\ E)\ u\ E1$

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 theorem

TRANS_Is_Agent: $\vdash \forall E\ u\ E'.\ \mathsf{Trans}\ E\ u\ E' \supset \mathsf{Is\_Agent}\ E \supset \mathsf{Is\_Agent}\ E'$

asserts that the derivatives of an agent are themselves agents. Its proof is an application of rule induction.[7] This theorem will be useful when proving properties about the behavioural semantics.

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 proving the goal $\Gamma \ ?\vdash\ \mathsf{Trans}\ (\mathsf{sum}\ E1\ E2)\ u\ E$ is reduced to proving $\Gamma \ ?\vdash\ \mathsf{Trans}\ E1\ u\ E$.

The theorem for performing exhaustive case analysis over the inductively defined relation allows one to derive 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 several properties about the relation Trans and the CCS operators can be derived. A few of them are presented below:

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

TRANS_PREFIX_EQ:
$\vdash\ \forall u\ E\ u'\ E'.\ \mathsf{Trans}\ (\mathsf{prefix}\ u\ E)\ u'\ E' = (u' = u) \wedge (E' = E)$

TRANS_SUM_EQ:
$\vdash\ \forall E\ E'\ u\ E''.\ \mathsf{Trans}\ (\mathsf{sum}\ E\ E')\ u\ E'' = \mathsf{Trans}\ E\ u\ E'' \vee \mathsf{Trans}\ E'\ u\ E''$

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

Many other theorems about Trans are proved, which will be used when deriving properties about the various behavioural semantics. Moreover, the theorems about Trans allow one to "animate" the operational semantics and thus simulate the behaviour of

[7] Note that the theorem is written in a form suitable for applying rule induction.

agent specifications. Given any agent expression E, the conversion `Run_CONV` returns a theorem stating the possible transitions of E in the following form:

$$\vdash\ \forall u\, E'.\ \mathsf{Trans}\ E\ u\ E' = ((u = u_1)\ \wedge\ (E' = E_1))\ \vee\ \ldots\ \vee\ ((u = u_n)\ \wedge\ (E' = E_n))$$

where $u_1, \ldots, u_n$ are the (immediate) actions that E can perform and E_j is the agent derivative into which E evolves after performing u_j, $1 \leq j \leq n$.

2.2.3 The Strong and Observation Semantics

The various behavioural semantics are defined based on the formalisation of the labelled transition relation and then the algebraic laws are derived by formal proof. The HOL-CCS environment includes the mechanisation of the strong semantics (bisimulation and equivalence) and the observation semantics (weak bisimulation, observation equivalence and congruence). In what follows, the HOL definition of strong bisimulation and equivalence is briefly presented and then the formalisation of the observation semantics is illustrated.

Using the HOL restricted quantification, strong bisimulation is formalised by the following constant definition (Sections A.2 and A.3):

$$\begin{array}{l}
\forall\, Bsm. \\
\quad \mathsf{Strong_Bisim}\ Bsm = \\
\quad (\forall E\ E' :: \mathsf{Is_Agent}. \\
\qquad Bsm\ E\ E'\ \supset \\
\qquad (\forall u. \\
\qquad\quad (\forall E1 :: \mathsf{Is_Agent}. \\
\qquad\qquad \mathsf{Trans}\ E\ u\ E1\ \supset\ (\exists E2 :: \mathsf{Is_Agent}.\ \mathsf{Trans}\ E'\ u\ E2\ \wedge\ Bsm\ E1\ E2))\ \wedge \\
\qquad\quad (\forall E2 :: \mathsf{Is_Agent}. \\
\qquad\qquad \mathsf{Trans}\ E'\ u\ E2\ \supset\ (\exists E1 :: \mathsf{Is_Agent}.\ \mathsf{Trans}\ E\ u\ E1\ \wedge\ Bsm\ E1\ E2))))
\end{array}$$

Restricted quantifications formalise the fact that the variables denote terms in the set $\mathcal{P}$ of agents. As recalled in Section 2.1.1, this is usually expressed by saying that $\mathcal{P}$ is ranged over by given symbols P, Q, ..., and then using such letters to denote the set of terms under consideration. Thus, these symbols also carry, implicitly, some kind of semantic meaning which must instead be explicitly encoded in HOL. In the HOL formalisation of CCS, two variables E and P of type *CCS* denote any agent expression. If agents are to be considered, then this has to be explicitly asserted using, for example, the predicate Is_Agent.[8]

Several properties about Strong_Bisim have been formally derived in HOL, such as proving that the identity relation is a strong bisimulation, the converse of a strong

[8]Nevertheless, it is worth noting that in the definition of Strong_Bisim (as in many of the following definitions), only the restricted universal quantification '$\forall E\, E' ::$ Is_Agent' is strictly necessary. Given such a condition on E and E', the fact that $E1$ and $E2$ are agents can be easily obtained by modus ponens with the theorem TRANS_Is_Agent.

bisimulation is a strong bisimulation and the union and the composition of two strong bisimulations are strong bisimulations.

The strong equivalence $E \sim E'$ over agents, represented in HOL by the relation Strong_Equiv $E\ E'$, is defined using the function `new_resq_definition` (Section A.3) for defining constants with restricted quantified arguments:

$$\forall E\ E' :: \mathsf{Is\_Agent}.\ \mathsf{Strong\_Equiv}\ E\ E' = (\exists\, Bsm.\ Bsm\ E\ E' \wedge \mathsf{Strong\_Bisim}\ Bsm)$$

Using the theorems about strong bisimulation, it is easy to show that strong equivalence is a congruence relation, namely an equivalence relation preserved by all CCS operators over agents.

As far as the observation semantics is concerned, the reflexive-transitive closure of the transition relation $E \xrightarrow{\tau} E'$, represented in HOL by Eps $E\ E'$, is first defined using the derived rule for inductive definitions. Based on the relation Eps, the weak transition relation $E \stackrel{u}{\Longrightarrow} E'$ is formalised in HOL by making a constant definition:

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

The following theorem can now be proved, it asserts that a transition Trans $E\ u\ E'$ is a particular weak transition Weak_Trans $E\ u\ E'$:

$$\text{TRANS\_IMP\_WEAK\_TRANS:}\ \vdash \forall E\ u\ E'.\ \mathsf{Trans}\ E\ u\ E' \supset \mathsf{Weak\_Trans}\ E\ u\ E'$$

The notion of weak bisimulation is then encoded by the following constant definition for Weak_Bisim : $CCS \to CCS \to bool$:

$$\begin{array}{l}
\forall Wbsm. \\
\quad \mathsf{Weak_Bisim}\ Wbsm = \\
\quad (\forall E\ E' :: \mathsf{Is_Agent}. \\
\qquad Wbsm\ E\ E' \supset \\
\qquad (\forall l. \\
\qquad\quad (\forall E1 :: \mathsf{Is_Agent}. \\
\qquad\qquad \mathsf{Trans}\ E\ (\mathsf{label}\ l)\ E1 \supset \\
\qquad\qquad (\exists E2 :: \mathsf{Is_Agent}.\ \mathsf{Weak_Trans}\ E'\ (\mathsf{label}\ l)\ E2 \wedge Wbsm\ E1\ E2)) \wedge \\
\qquad\quad (\forall E2 :: \mathsf{Is_Agent}. \\
\qquad\qquad \mathsf{Trans}\ E'\ (\mathsf{label}\ l)\ E2 \supset \\
\qquad\qquad (\exists E1 :: \mathsf{Is_Agent}.\ \mathsf{Weak_Trans}\ E\ (\mathsf{label}\ l)\ E1 \wedge Wbsm\ E1\ E2))) \wedge \\
\qquad (\forall E1 :: \mathsf{Is_Agent}. \\
\qquad\quad \mathsf{Trans}\ E\ \mathsf{tau}\ E1 \supset (\exists E2 :: \mathsf{Is_Agent}.\ \mathsf{Eps}\ E'\ E2 \wedge Wbsm\ E1\ E2)) \wedge \\
\qquad (\forall E2 :: \mathsf{Is_Agent}. \\
\qquad\quad \mathsf{Trans}\ E'\ \mathsf{tau}\ E2 \supset (\exists E1 :: \mathsf{Is_Agent}.\ \mathsf{Eps}\ E\ E1 \wedge Wbsm\ E1\ E2)))
\end{array}$$

Theorems similar to the ones for strong bisimulation are derived for weak bisimulation by making use of the above theorem TRANS_IMP_WEAK_TRANS.

The observation equivalence $E \approx E'$ over agents is defined as the following constant Obs_Equiv with restricted quantified arguments:

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

Using the theorems about weak bisimulation, it is shown that the observation equivalence is an equivalence relation and is preserved by all CCS operators over agents, except for summation. It is proved that observation equivalence is preserved by the summation operator under the hypothesis of stability of agents, where the predicate Stable is defined as follows:

$$\forall E :: \mathsf{Is\_Agent}.\ \mathsf{Stable}\ E = (\forall u.\ \forall E' :: \mathsf{Is\_Agent}.\ \mathsf{Trans}\ E\ u\ E' \supset \neg(u = \mathsf{tau}))$$

Based on the definition of observation equivalence, the observation congruence $E \approx_c E'$ over agents is formalised through a constant definition by defining the relation $\mathsf{Obs\_Congr} : CCS \rightarrow CCS \rightarrow bool$ as follows:

$\forall E\ E' ::$ Is_Agent.
 Obs_Congr $E\ E' =$
 $(\forall u.$
  $(\forall E1 ::$ Is_Agent.
   Trans $E\ u\ E1 \supset (\exists E2 ::$ Is_Agent. Weak_Trans $E'\ u\ E2 \wedge$ Obs_Equiv $E1\ E2)) \wedge$
  $(\forall E2 ::$ Is_Agent.
   Trans $E'\ u\ E2 \supset (\exists E1 ::$ Is_Agent. Weak_Trans $E\ u\ E1 \wedge$ Obs_Equiv $E1\ E2)))$

Theorems similar to those proved for the previous equivalences are derived, such as showing that observation congruence is an equivalence relation and is preserved by all CCS operators over agents.

The relationship between the different behavioural equivalences is also formally derived. A fact which will be useful when proving the behavioural laws in the next section, is that strong equivalence implies both observation equivalence and congruence, as the following theorem says as far as observation congruence is concerned:

STRONG_IMP_OBS_CONGR:
$\vdash \forall E\ E' ::$ Is_Agent. Strong_Equiv $E\ E' \supset$ Obs_Congr $E\ E'$

Moreover, it is proved that observation congruence implies observation equivalence, plus several other properties given in [97] that formalise meta-reasoning about the process calculus CCS in the HOL system.

2.2.4 The Laws for Observation Congruence

The laws for the CCS operators (Section 2.1.3) are usually shown to be correct by starting from the definition of observation congruence and then exhibiting an appropriate weak bisimulation. All of them have been formally derived in HOL, but

their mechanisation follows a different approach based on strong equivalence by exploiting the implication asserted by the theorem STRONG_IMP_OBS_CONGR. The basic laws are common to all the behavioural semantics and it is easier to deal with strong bisimulation than with the relation $\stackrel{u}{\Longrightarrow}$ which includes any possible sequence of silent actions. This means that the basic laws are proved for strong equivalence and then derived for the other equivalences by simple inference steps. Obviously, the τ-laws for any behavioural equivalence other than strong equivalence must be proved directly, unless they are derivable from other equivalences (e.g. the τ-laws for observation congruence hold for observation equivalence as well).

The formalisation of a few laws for observation congruence is given below to illustrate its similarity to the presentation in Section 2.1.3. They are the laws for the summation operator (A1)–(A4), the τ-laws (T1)–(T3) and the unfolding law for recursion (A12):

SUM_ASSOC_R: $\vdash \forall E\ E'\ E''$ :: Is_Agent.
Obs_Congr (sum (sum E E') E'') (sum E (sum E' E''))

SUM_COMM: $\vdash \forall E\ E'$ :: Is_Agent. Obs_Congr (sum E E') (sum E' E)

SUM_IDEMP: $\vdash \forall E$ :: Is_Agent. Obs_Congr (sum E E) E

SUM_IDENT_R: $\vdash \forall E$ :: Is_Agent. Obs_Congr (sum E nil) E

TAU_1: $\vdash \forall u.\ \forall E$ :: Is_Agent. Obs_Congr (prefix u (prefix tau E)) (prefix u E)

TAU_2: $\vdash \forall E$ :: Is_Agent. Obs_Congr (sum E (prefix tau E)) (prefix tau E)

TAU_3: $\vdash \forall u.\ \forall E\ E'$ :: Is_Agent.
Obs_Congr
(sum (prefix u (sum E (prefix tau E'))) (prefix u E'))
(prefix u (sum E (prefix tau E')))

UNFOLDING: $\vdash \forall X\ E$.
Is_Agent (rec X E) $\supset$
Obs_Congr (rec X E) (CCS_Subst E (rec X E) X)

Note that all the expressions on the left and right-hand sides of the observation congruences can be easily proved to be agents, under the assumption that the variables E, E', E'' of type *CCS* denote agents. In the unfolding law the assumption is not on the agent expression E, which typically is not an agent, but on the recursive expression rec X E. The right-hand side CCS_Subst E (rec X E) X is also an agent using the theorem Is_Agent_CCS_Subst_rec (Section 2.2.1).

The only law which requires some explanation is the expansion law (A11) of Section 2.1.3 for the parallel composition operator.[9] This is due to the notation for indexed summation which needs to be formalised in HOL. In particular, "finite" indexed summations of prefixed agents are used in (A11), where indexes range over

[9]The unique fixpoint law is not yet mechanised in HOL.

natural numbers. Finite summations of agent expressions can be formalised by means of a function SIGMA such that, given a function $f : num \rightarrow CCS$ and an index $n : num$, the expression SIGMA $f\ n$ denotes the summation of $n+1$ agent expressions $(\ldots(f\ 0 + f\ 1) + \ldots) + f\ n$. The function SIGMA is defined by primitive recursion:

$$(\forall f.\ \mathsf{SIGMA}\ f\ 0\ =\ f\ 0)\ \wedge$$
$$(\forall f\ n.\ \mathsf{SIGMA}\ f\ (n+1)\ =\ \mathsf{sum}\ (\mathsf{SIGMA}\ f\ n)\ (f\ (n+1)))$$

The transitions of the agent summation SIGMA $f\ n$ are given by the following theorem, which is proved by mathematical induction on n:

SIGMA_TRANS_THM_EQ:
$$\vdash \forall n\ f\ u\ E.\ \mathsf{Trans}\ (\mathsf{SIGMA}\ f\ n)\ u\ E\ =\ (\exists k.\ k \leq n\ \wedge\ \mathsf{Trans}\ (f\ k)\ u\ E)$$

The summands $f\ i$ in SIGMA $f\ n$ can be checked to be prefixed expressions using the predicate Is_Prefix:

$$\forall E.\ \mathsf{Is\_Prefix}\ E\ =\ (\exists u\ E'.\ E = \mathsf{prefix}\ u\ E')$$

and the action and process components of prefixed expressions are extracted by means of the projection functions PREF_ACT and PREF_PROC respectively:

$$\forall u\ E.\ \mathsf{PREF\_ACT}\ (\mathsf{prefix}\ u\ E)\ =\ u$$
$$\forall u\ E.\ \mathsf{PREF\_PROC}\ (\mathsf{prefix}\ u\ E)\ =\ E$$

A function ALL_SYNC is then defined by primitive recursion, which computes the summation of all possible synchronisations between two agent summations given through the functions f, $f' : num \rightarrow CCS$ of length n, m respectively. ALL_SYNC makes use of a function SYNC which computes the summation of all possible synchronisations between a single prefixed expression $u.E$ and a summation. Such a function is defined again by primitive recursion:

$$\begin{array}{l}
(\forall u\ E\ f. \\
\quad \mathsf{SYNC}\ u\ E\ f\ 0\ = \\
\quad (((u = \mathsf{tau}) \vee (\mathsf{PREF_ACT}\ (f\ 0) = \mathsf{tau}))\ \Rightarrow\ \mathsf{nil}\ \mid \\
\quad\ ((\mathsf{LABEL}\ u = \mathsf{Compl}\ (\mathsf{LABEL}\ (\mathsf{PREF_ACT}\ (f\ 0))))\ \Rightarrow \\
\quad\quad \mathsf{prefix}\ \mathsf{tau}\ (\mathsf{par}\ E\ (\mathsf{PREF_PROC}\ (f\ 0)))\ \mid\ \mathsf{nil}))) \wedge \\
(\forall u\ E\ f\ n. \\
\quad \mathsf{SYNC}\ u\ E\ f\ (n+1)\ = \\
\quad (((u = \mathsf{tau}) \vee (\mathsf{PREF_ACT}\ (f\ (n+1)) = \mathsf{tau}))\ \Rightarrow\ \mathsf{SYNC}\ u\ E\ f\ n\ \mid \\
\quad\ ((\mathsf{LABEL}\ u = \mathsf{Compl}\ (\mathsf{LABEL}\ (\mathsf{PREF_ACT}\ (f\ (n+1)))))\ \Rightarrow \\
\quad\quad \mathsf{sum}\ (\mathsf{prefix}\ \mathsf{tau}\ (\mathsf{par}\ E\ (\mathsf{PREF_PROC}\ (f\ (n+1)))))\ (\mathsf{SYNC}\ u\ E\ f\ n)\ \mid \\
\quad\quad \mathsf{SYNC}\ u\ E\ f\ n)))
\end{array}$$

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

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

The function ALL_SYNC is thus defined as follows:

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

The law (A11) is finally derived by proving the following theorem:

$$
\begin{array}{l}
\vdash \forall f\ n\ f'\ m. \\
\quad (\forall i.\ i \leq n\ \supset\ \mathsf{Is\_Agent}\ (f\ i)\ \wedge\ \mathsf{Is\_Prefix}\ (f\ i)) \wedge \\
\quad (\forall j.\ j \leq m\ \supset\ \mathsf{Is\_Agent}\ (f'\ j)\ \wedge\ \mathsf{Is\_Prefix}\ (f'\ j))\ \supset \\
\quad \mathsf{Obs\_Congr} \\
\quad (\mathsf{par}\ (\mathsf{SIGMA}\ f\ n)\ (\mathsf{SIGMA}\ f'\ m)) \\
\quad (\mathsf{sum} \\
\quad\ (\mathsf{sum} \\
\quad\ \ (\mathsf{SIGMA}\ (\lambda i.\ \mathsf{prefix}\ (\mathsf{PREF\_ACT}\ (f\ i))\ (\mathsf{par}\ (\mathsf{PREF\_PROC}\ (f\ i))\ (\mathsf{SIGMA}\ f'\ m)))\ n) \\
\quad\ \ (\mathsf{SIGMA}\ (\lambda j.\ \mathsf{prefix}\ (\mathsf{PREF\_ACT}\ (f'\ j))\ (\mathsf{par}\ (\mathsf{SIGMA}\ f\ n)\ (\mathsf{PREF\_PROC}\ (f'\ j))))\ m)) \\
\quad\ (\mathsf{ALL\_SYNC}\ f\ n\ f'\ m))
\end{array}
$$

The law is formulated under the assumption that the summands in the two indexed summations are prefixed agents. This is checked by the predicates Is_Agent and Is_Prefix for all summands. Using such predicates, the subtype of all functions $f : num \rightarrow CCS$ such that, for all i, $f\,i$ is a prefixed agent, could be defined through type abstraction (Section A.3). Thus, the function f in any agent summation SIGMA f n would be chosen as an element of such a type, rather than defining it as a raw function from numbers to agent expressions and then checking that each summand is a prefixed agent. Nevertheless, the formalisation of the above notation for indexed summations is completely hidden from the user and raw functions seem adequate for the given purpose. A conversion for the application of the expansion law (A11) is defined that includes the translation from a CCS summation to a SIGMA form and vice versa, and checks that all summands are prefixed agents. It is also very natural to use numbers as the indexing set for CCS summations and then define the necessary functions by primitive recursion. A further alternative to this formalisation is to adopt *set indexed operations* using a package developed in HOL by Newey [94], based on previous work by Chou [25]. This package enables the lifting of a binary, associative and commutative operator with identity, such as CCS binary summation, to operate on a collection of terms indexed by a set. This offers the possibility of explicitly writing (and reasoning about) "finite" summations of agents given through indexing sets by defining a new indexed summation operation. However, possibly infinite indexed summations of CCS agents will be needed when mechanising the translation of value-passing ex-

pressions into pure expressions in Chapter 4. Their HOL formalisation (Section 3.3.2) is just an extension of the above functional approach.

2.2.5 The Modal Logic

The first step in the formalisation in HOL of the modal logic is to represent its syntax. This is done by defining a concrete data type *eHML* of formulas of the extended Hennessy-Milner logic using the derived rule for recursive type definition:

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

Similarly to the definition of the types *label*, *action* and *CCS* (Section 2.2.1), a theorem which completely characterises the type *eHML* is automatically derived in HOL. Given this theorem, the satisfaction relation $\mathsf{Sat} : \mathit{CCS} \to \mathit{eHML} \to \mathit{bool}$ is defined by primitive recursion over the type *eHML*, thus obtaining the following theorems:[10]

SAT_tt: $\vdash \forall E :: \mathsf{Is\_Agent}.\ \mathsf{Sat}\ E\ \mathsf{tt}$

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

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

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

The derived operators of the modal logic are then defined:

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

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

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

and the related theorems for the relation Sat are easily proved by rewriting with the above definitions and the satisfaction rules for the basic modal operators:

SAT_ff: $\vdash\ \forall E :: \mathsf{Is\_Agent}.\ \neg\, \mathsf{Sat}\ E\ \mathsf{ff}$

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

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

[10] Actually, as the HOL rule for primitive recursive definitions (Section A.4) is not extended to deal with restricted quantification, the relation Sat is first defined using `new_recursive_definition` and the predicate Is_Agent, and then the above theorems with restricted quantification are derived.

A tactic that reduces a goal which matches the structure of formulas is defined for each case of the satisfaction relation. For example, the tactic `SAT_conj_TAC` is generated from the theorem SAT_conj so that proving a goal $\Gamma ?\vdash \mathsf{Sat}\, E\, (\mathsf{conj}\, Fm\, Fm')$ is reduced to proving that E is an agent plus the subgoals $\Gamma ?\vdash \mathsf{Sat}\, E\, Fm$ and $\Gamma ?\vdash \mathsf{Sat}\, E\, Fm'$.

2.3 Reasoning about CCS Specifications

Once the formal theory for the CCS behavioural semantics and modal logic has been embedded in the HOL system, a collection of proof tools can be developed to assist in verifying properties of agent specifications. These tools include inference rules, conversions and tactics, which can be used either interactively in a stepwise fashion or composed together to give automatic strategies whenever possible.

2.3.1 Rewriting modulo Behavioural Equivalences in HOL

Basic rules, conversions and tactics are first needed for rewriting CCS expressions with respect to behavioural relations. The built-in HOL rewriting tools for the equality relation cannot be used, as rewriting is performed modulo a behavioural equivalence. For each behavioural equivalence *Beh* and for each CCS operator *op*, a conversion has been defined in HOL that applies the algebraic laws for *op* with respect to *Beh*. Given a CCS expression E which contains one or more instances of the left-hand sides of the laws for *op*, the theorem returned by the corresponding conversion asserts the behavioural equivalence *Beh* E E', where E' is the result of applying the laws for *op* to E. Depending on the definition of the conversion, the expression E' can still contain subexpressions which can be rewritten with the same laws, so the same conversion can be used once more. Let `OC_RELAB_ELIM_CONV` be the conversion which applies the basic laws (A8)–(A10) for the relabelling operator with respect to observation congruence (Section 2.1.3). Consider the following interaction with the HOL system:

```
#let E = "(a.b.nil + c.b.nil) [b1/b]";;
E = "(a.b.nil + c.b.nil) [b1/b]" : term

#OC_RELAB_ELIM_CONV E;;
⊢ Obs_Congr
  ((a.b.nil + c.b.nil) [b1/b])
  ((a.b.nil [b1/b]) + (c.b.nil [b1/b]))

#(OC_TOP_DEPTH_CONV OC_RELAB_ELIM_CONV) E;;
⊢ Obs_Congr ((a.b.nil + c.b.nil) [b1/b]) (a.b1.nil + c.b1.nil)
```

The conversion OC_RELAB_ELIM_CONV applies only the law (A9) to the term E. Using the function OC_TOP_DEPTH_CONV which repeatedly applies a conversion for observation congruence until it fails, the laws (A8)–(A10) for relabelling are applied, thus renaming the occurrences of the action b with $b1$ and deleting those of the relabelling operator.[11] The conversions for applying the behavioural laws also check that the expressions being manipulated are actually agents using the conversion Is_Agent_CONV.

Starting from these conversions, the associated tactics have been defined which apply the algebraic laws for given operator *op* and behavioural equivalence *Beh* to a goal. For example, the tactic OC_RELAB_ELIM_TAC is defined by converting OC_RELAB_ELIM_CONV into a tactic. More complex conversions and tactics can be defined in terms of smaller conversions and tactics. The tactic OC_EXP_THM_TAC that applies the expansion theorem for observation congruence is defined by composing the tactics for the relabelling, parallel composition and restriction operators.

Sometimes, the application of a given theorem to a behavioural relation in the current goal is instead desired. In this case, a substitution tactic for the behavioural relation can be used, provided with the theorem to be applied. Let RELAB_SUM be the theorem which formalises the law (A9) for observation congruence in HOL:

$$\vdash \forall E\ E' :: \mathsf{Is\_Agent}.\ \forall rf.\ \mathsf{Obs\_Congr}\ (\mathsf{relab}\ (\mathsf{sum}\ E\ E')\ rf)\ (\mathsf{sum}\ (\mathsf{relab}\ E\ rf)\ (\mathsf{relab}\ E'\ rf))$$

This theorem can be applied if the terms which instantiate the variables E and E' are agents. In fact, the equational laws for behavioural equivalences are formalised in HOL as implicational theorems: restricted quantification like '$\forall E :: \mathsf{Is\_Agent}.\ P[E]$' is just a short-hand notation for '$\forall E : \mathit{CCS}.\ \mathsf{Is\_Agent}\, E \supset P[E]$'. Substitution tactics can be defined for conditional rewriting modulo behavioural equivalences by adapting some functions for conditional (equational) rewriting in HOL [125]. These tactics apply the given implicational theorem under the assumptions that the terms being manipulated are agents, and a subgoal for each of such assumptions (which need to be checked) is generated besides the main behavioural subgoal. Actually, it is not necessary to have proofs about behavioural equivalences cluttered with subgoals for checking whether a given expression is an agent. This *syntactic* check can be done automatically by the substitution tactics and the assumptions removed (if the check is successful, otherwise the substitution step is not valid).[12] The tactic OC_SUBST1_TAC does this and rewrites a goal by substituting the right-hand side of the given observational theorem for every occurrence of its left-hand side in the goal. Hence, given the goal

```
#g "Obs_Congr ((a.b.nil + c.b.nil) [b1/b]) E'";;
"Obs_Congr ((a.b.nil + c.b.nil) [b1/b]) E'"
```

[11] The reader is referred to [106] for further information on basic conversions, such as DEPTH_CONV and TOP_DEPTH_CONV, which are here extended to deal with behavioural equivalences.

[12] In Section 4.5 conditional rewriting modulo behavioural equivalences will be implemented to reason about data in value-passing agents.

the application of (a specialised version of) the theorem RELAB_SUM through the tactic OC_SUBST1_TAC would result in an invalid step, as no information about whether E' is an agent is given:[13]

```
#e (OC_SUBST1_TAC
#   (SPEC "[b1/b]"
#    (RESQ_SPECL ["a.b.nil"; "c.b.nil"] RELAB_SUM)));;
OK..
evaluation failed     Invalid tactic
```

By providing such information, the substitution is successfully applied:

```
#set_goal(["Is_Agent E'"],
          "Obs_Congr ((a.b.nil + c.b.nil) [b1/b]) E'";;
"Obs_Congr ((a.b.nil + c.b.nil) [b1/b]) E'"
    ["Is_Agent E'" ]

#e (OC_SUBST1_TAC
#   (SPEC "[b1/b]"
#    (RESQ_SPECL ["a.b.nil"; "c.b.nil"] RELAB_SUM)));;
OK..
"Obs_Congr ((a.b.nil [b1/b]) + (c.b.nil [b1/b])) E'"
    ["Is_Agent E'" ]
```

When dealing with behavioural relations, it is often the case that only one agent expression needs to be rewritten with a single theorem or a list of theorems. For this purpose, tactics such as OC_LHS_SUBST1_TAC and OC_RHS_SUBST1_TAC that rewrite, respectively, the left-hand side and the right-hand side of an observation congruence with a single theorem, have been defined. In the above example, OC_LHS_SUBST1_TAC can be used instead of OC_SUBST1_TAC. Rewriting with a list of theorems (modulo observation congruence) can instead be performed using tactics like OC_LHS_SUBST_TAC and OC_RHS_SUBST_TAC.

2.3.2 Verification Strategies

Several verification strategies can be defined based on the HOL mechanisation of the CCS calculus, depending on the subset of specifications under consideration, the kind of property to be proved, the proof techniques to be applied, the complexity of the specifications and the level of confidence one has in their correctness.

When dealing with finite CCS expressions, the observation congruence of two agents can be decided by means of an *automatic rewriting strategy*. In fact, the

[13]Note that this information is required by the transitivity rule of observation congruence, in order to justify (or validate) the HOL proof.

complete axiomatisation for observation congruence over finite CCS (Section 2.1.3) can be used for rewriting the two agents into their normal forms with respect to the algebraic laws. As mentioned in Section 1.1.2, a complete term rewriting system which is equivalent to the axiomatisation does not exist. However, it is possible to define a rewriting strategy that proves the observation congruence of two finite agents by first reducing them to their normal forms, and then checking their equivalence (modulo associativity and commutativity for the summation operator) [69]. This strategy adopts criteria for selecting the law to be applied, which depend on the state of the proof and on the state of the expression being manipulated. It is fully automatic and has been embedded in HOL [23].[14]

When verifying recursive agents, automatic verification strategies are still possible whenever agents are finite state. But even in this case one might prefer to be able to guide the verification process, in particular when one is not confident in the correctness of the specifications and when the outcome of the automatic verification is negative and no feedback is given about what is wrong. Moreover, partially interactive strategies are essential for reasoning about infinite state specifications. The steps of an interactive verification strategy, called *lazy expansion* in [23], are defined for dealing with the unfolding law (A12) for recursion and the expansion law (A11) for parallel composition, at the same time keeping the size of the expression being manipulated to a minimum. This strategy implements the usual verification method for recursive agents: it first expands the specifications with the definitions of the agents involved, then applies the behavioural laws (and possibly other theorems) by reducing the resulting expressions as much as possible, and finally folds back some subexpressions by rewriting with the definitions of the agents, thus controlling the termination of the application of the unfolding law.

Agent specifications are often parameterised and their definitions depend on the parameters' values. In this case the above verification strategies must be enriched with other (mathematical) proof techniques, which are typically available in a theorem proving framework. Parameters can be indexes, so that the verification process involves some form of induction. For example, the verification of specifications indexed over natural numbers is usually by mathematical induction. In other situations, case analysis can be sufficient to carry out a proof for indexed specifications. Case analysis and contradiction are also useful when dealing with parameterised specifications and value-passing expressions (as it will be shown in Sections 3.2 and 4.6).

All these strategies and proof techniques are used to verify several properties of CCS agents. Behavioural equivalences can be verified with either an "operational" or an "equational" approach. When the behavioural semantics under consideration is defined in terms of bisimulation, two different descriptions, say *Spec* and *Impl*,

[14]Note that the formalisation in [23] is based on a non-definitional version of CCS in HOL. However, given the HOL mechanisation of CCS presented in the previous sections, the rewriting strategy can now be formalised in a purely definitional way.

of the same system can be proved to be equivalent by exhibiting a bisimulation which contains the pair (*Spec*, *Impl*). This can be done in HOL in two steps. In the case of strong bisimulation, an ML function `build_strong_bisim` is defined which, given two CCS agents, attempts to compute a strong bisimulation containing the two agents. This function returns a triple *(s,bsml,thml)*, where *s* is a string saying whether or not there exists a bisimulation, *bsml* is a list of pairs of agents, and *thml* is a list of pairs of theorems corresponding to the agents in *bsml* and giving their possible transitions (using the conversion `Run_CONV` presented at the end of Section 2.2.2). If the function fails to find a bisimulation, namely *s* says that there is no bisimulation, then the list *bsml* contains those pairs of agents which have been tested for bisimilarity before the detection of failure. In this list there is at least one pair of agents for which the definition of bisimulation fails, namely an action of one of the agents is not matched by an equal action of the other. In this way some information is provided about non-bisimilar subexpressions and their location in the given specifications. If the function `build_strong_bisim` is successful, then the resulting list *bsml* is transformed into a binary relation *bsm* and, using the theorems in *thml*, the conversion `CHECK_STRONG_BISIM` is applied to prove that *bsm* is indeed a strong bisimulation. For example, this operational approach can be used to prove that the two recursive terms **rec** $X.\, a.\, a.\, X$ and **rec** $X.\, a.\, X$ are strongly equivalent. In HOL this is checked as follows:

```
#build_strong_bisim "rec X (a.a.X)" "rec X (a.X)";;
(`there exists a strong bisimulation`,
 [("rec X (a.a.X)", "rec X (a.X)");
  ("a.rec X (a.a.X)", "rec X (a.X)")],
 [(⊢ ∀u E.
      Trans (rec X (a.a.X)) u E = (u = a) ∧ (E = a.rec X (a.a.X)),
   ⊢ ∀u E.
      Trans (rec X (a.X)) u E = (u = a) ∧ (E = rec X (a.X)));
  (⊢ ∀u E.
      Trans (a.rec X (a.a.X)) u E = (u = a) ∧ (E = rec X (a.a.X)),
   ⊢ ∀u E.
      Trans (rec X (a.X)) u E = (u = a) ∧ (E = rec X (a.X)))])
: (string # (term # term) list # (thm # thm) list)

#CHECK_STRONG_BISIM it;;
⊢ Strong_Bisim
   (λx y.
     (x = rec X (a.a.X)) ∧ (y = rec X (a.X)) ∨
     (x = a.rec X (a.a.X)) ∧ (y = rec X (a.X)))
```

Moreover, model checking is given in terms of transition systems, thus an approach based on the operational semantics of the process calculus is also used when checking

that a specification has a given modal/temporal property. Examples of verification of modal properties will be presented in Sections 2.4.2 and 3.2.2.

Alternative to this operational way of verification is an approach based on equational reasoning, in which agent specifications are manipulated by applying the above verification strategies, such as lazy expansion, which involve the use of equational laws. Examples of this kind of reasoning about pure and value-passing specifications will be shown in Sections 2.4.1, 3.2.1 and 4.6. Furthermore, a proof (by mathematical induction) of the correctness of an implementation for the parameterised specification of the buffer given in Section 1.1.1 has been mechanised in [100].

2.4 A Verification Example

This section shows how the HOL formalisation of the CCS calculus and the associated proof tools can be used to verify properties of pure agents by considering a simple example, namely a reader-writer system. In this simplified version of a scheduling problem, a reader and a writer contest via a semaphore to access a shared resource. Let the reading and writing tasks to be performed on the resource be simply specified by begin actions br and bw and by end actions er and ew. The abstract behaviour of this scheduler can be described as a completely non-deterministic alternation of reading and writing tasks, as specified by the following recursive agent *Spec*:

$$\mathbf{rec}\, X.\, (\tau.\, br.\, er.\, X \;+\; \tau.\, bw.\, ew.\, X)$$

At a less abstract level, the scheduler can be seen as the composition of three agents, i.e. a reader, a writer and a semaphore, running in parallel and synchronising between each other. Reader and writer make their requests to access the resource by performing the action $\overline{p}$. If the resource is available, the semaphore can synchronise non-deterministically with either of the two by executing the action p. If the reader has obtained access to the resource, it can perform its reading task by executing the actions br and er. The reader signals the release of the resource through another synchronisation with the semaphore by executing the action $\overline{v}$. The writer behaves similarly but with writing actions bw and ew replacing the reading ones. Thus, the behaviour of the two agents *Reader* and *Writer* is specified by the following terms:

$$\mathbf{rec}\, X.\, \overline{p}.\, br.\, er.\, \overline{v}.\, X$$

$$\mathbf{rec}\, X.\, \overline{p}.\, bw.\, ew.\, \overline{v}.\, X$$

The semaphore *Sem* simply controls the access to the resource by performing the actions p and v indefinitely:

$$\mathbf{rec}\, X.\, p.\, v.\, X$$

An implementation *Impl* of the reader-writer system is obtained by composing in parallel the agents *Reader*, *Writer* and *Sem* and by restricting over the synchronising

actions p and v:

$$(\mathit{Reader} \mid \mathit{Sem} \mid \mathit{Writer})\backslash\{p, v\} \ .$$

The following subsections give two verification proofs for the correctness of the reader-writer system.

2.4.1 Proving Behavioural Equivalences

The correctness of the implementation of the reader-writer system with respect to its specification can be shown by proving that *Impl* and *Spec* are observation congruent.[15] The agent expressions in this example are all agents, thus the laws for observation congruence can always be applied. The agent variable X is guarded and sequential in the body of the recursive expression for *Spec*. Thus, the unique fixed point law (A13) (Section 2.1.3) can be used and, by taking P as *Impl*, verifying $\mathit{Impl} \approx_c \mathit{Spec}$ reduces to proving that *Impl* "satisfies" the recursive definition of *Spec*, i.e. that the following observation congruence holds:

$$\mathit{Impl} \ \approx_c \ \tau.\, br.\, er.\, \mathit{Impl} \ + \ \tau.\, bw.\, ew.\, \mathit{Impl}$$

In what follows, two HOL (backward) proofs for this observation congruence are presented. In order to make the proofs more readable, agents are written in the usual CCS notation, while using the HOL predicate Obs_Congr (defined in Section 2.2.3) to denote the observation congruence. Moreover, a goal given by an assumption list $\Gamma \equiv \{A_1; \ldots; A_n\}$ and a term t is written $\Gamma \vdash t$ (if Γ is empty, the goal is simply $\vdash t$). The HOL sessions with the ML code for these proofs can be found in Appendix B.1.

The goal to be proved is the following:

$$\vdash \ \mathsf{Obs\_Congr}\ \mathit{Impl}\ (\tau.\, br.\, er.\, \mathit{Impl} \ + \ \tau.\, bw.\, ew.\, \mathit{Impl})$$

The lazy expansion strategy is able to solve this goal. The idea is to rewrite the left-hand side of the behavioural goal so to transform it into an expression provably congruent to the right-hand side. Thus, only the left-hand side is expanded using the definitions of the agents:[16]

$$\begin{array}{l} \vdash \ \mathsf{Obs\_Congr} \\ \quad ((\mathbf{rec}\, X.\, \overline{p}.\, br.\, er.\, \overline{v}.\, X \ \mid \ (\mathbf{rec}\, X.\, p.\, v.\, X \ \mid \ \mathbf{rec}\, X.\, \overline{p}.\, bw.\, ew.\, \overline{v}.\, X))\backslash\{p, v\}) \\ \quad (\tau.\, br.\, er.\, \mathit{Impl} \ + \ \tau.\, bw.\, ew.\, \mathit{Impl}) \end{array}$$

[15] A forward proof of the same behavioural relation is presented in [23], but such a proof is based on a non-definitional formalisation of pure CCS in HOL. The observation congruence of a Basic LOTOS version of the reader-writer system is also verified in [76] using the tool PAM.

[16] Note that parallel composition is formalised in HOL as a binary operator. The agent *Impl* is thus defined as either $((\mathit{Reader} \mid \mathit{Sem}) \mid \mathit{Writer})\backslash\{p, v\}$ or $(\mathit{Reader} \mid (\mathit{Sem} \mid \mathit{Writer}))\backslash\{p, v\}$. The two agents are equivalent as parallel composition is associative with respect to observation congruence, as formally derived in HOL. The second agent definition is chosen in this example.

By unfolding the recursive agents once and using the agent definitions again, the following goal is obtained:

$$\vdash \mathsf{Obs\_Congr}\ ((\overline{p}.\, br.\, er.\, \overline{v}.\, Reader \mid (p.\, v.\, Sem \mid \overline{p}.\, bw.\, ew.\, \overline{v}.\, Writer))\backslash\{p, v\})\ (\tau.\, br.\, er.\, Impl + \tau.\, bw.\, ew.\, Impl)$$

The expansion theorem for observation congruence is applied and this means, in particular, using the expansion law (A11) and the restriction laws (A6)–(A7) (Section 2.1.3) for deleting those summands obtained by expansion that are prefixed by some action in the restriction set. Only the synchronisations are left:

$$\vdash \mathsf{Obs\_Congr}\ (\tau.\, ((\overline{p}.\, br.\, er.\, \overline{v}.\, Reader \mid (v.\, Sem \mid bw.\, ew.\, \overline{v}.\, Writer))\backslash\{p, v\}) + \tau.\, ((br.\, er.\, \overline{v}.\, Reader \mid (v.\, Sem \mid \overline{p}.\, bw.\, ew.\, \overline{v}.\, Writer))\backslash\{p, v\}))\ (\tau.\, br.\, er.\, Impl + \tau.\, bw.\, ew.\, Impl)$$

Another expansion step pulls the only possible actions bw and br out of the parallel and restricted subexpressions:

$$\vdash \mathsf{Obs\_Congr}\ (\tau.\, bw.\, ((\overline{p}.\, br.\, er.\, \overline{v}.\, Reader \mid (v.\, Sem \mid ew.\, \overline{v}.\, Writer))\backslash\{p, v\}) + \tau.\, br.\, ((er.\, \overline{v}.\, Reader \mid (v.\, Sem \mid \overline{p}.\, bw.\, ew.\, \overline{v}.\, Writer))\backslash\{p, v\}))\ (\tau.\, br.\, er.\, Impl + \tau.\, bw.\, ew.\, Impl) \tag{2.1}$$

At this point the proof can proceed in two different ways. The first one consists of manipulating only the left-hand side of the behavioural goal using the lazy expansion strategy, as done so far. Two further expansion steps lead to the following goal:

$$\vdash \mathsf{Obs\_Congr}\ (\tau.\, bw.\, ew.\, \tau.\, ((\overline{p}.\, br.\, er.\, \overline{v}.\, Reader \mid (Sem \mid Writer))\backslash\{p, v\}) + \tau.\, br.\, er.\, \tau.\, ((Reader \mid (Sem \mid \overline{p}.\, bw.\, ew.\, \overline{v}.\, Writer))\backslash\{p, v\}))\ (\tau.\, br.\, er.\, Impl + \tau.\, bw.\, ew.\, Impl)$$

Either the writer or the reader has terminated its task and then released the resource by synchronising again with the semaphore. This generates the new occurrences of the internal action τ. They can be removed using the τ-law (T1) (Section 2.1.3):

$$\vdash \mathsf{Obs\_Congr}\ (\tau.\, bw.\, ew.\, ((\overline{p}.\, br.\, er.\, \overline{v}.\, Reader \mid (Sem \mid Writer))\backslash\{p, v\}) + \tau.\, br.\, er.\, ((Reader \mid (Sem \mid \overline{p}.\, bw.\, ew.\, \overline{v}.\, Writer))\backslash\{p, v\}))\ (\tau.\, br.\, er.\, Impl + \tau.\, bw.\, ew.\, Impl)$$

By folding back the subexpressions for *Reader* and *Writer* using the behavioural theorems

$$\text{Reader\_lemma}: \quad \vdash \mathsf{Obs\_Congr}\ (\overline{p}.\, br.\, er.\, \overline{v}.\, Reader)\ Reader$$

$$\text{Writer\_lemma}: \quad \vdash \mathsf{Obs\_Congr}\ (\overline{p}.\, bw.\, ew.\, \overline{v}.\, Writer)\ Writer$$

the following goal is obtained:

$$\begin{array}{l} ?\vdash \mathsf{Obs\_Congr} \\ \quad (\tau.\, bw.\, ew.\, ((Reader \mid (Sem \mid Writer))\backslash\{p, v\}) + \\ \quad \tau.\, br.\, er.\, ((Reader \mid (Sem \mid Writer))\backslash\{p, v\})) \\ \quad (\tau.\, br.\, er.\, Impl + \tau.\, bw.\, ew.\, Impl) \end{array}$$

This goal is simply solved by rewriting with the definition of *Impl* and the commutativity law (A2) for binary summation with respect to observation congruence (Section 2.1.3).

After the first two expansion steps, a different proof technique could be used which manipulates both sides of the behavioural goal (2.1). The substitutivity property of the binary summation operator with respect to observation congruence, which in HOL is the theorem

$$\begin{array}{l} \mathrm{OBS\_CONGR\_PRESD\_BY\_SUM}: \\ \vdash \forall E1\ E1'\ E2\ E2' :: \mathsf{Is\_Agent}. \\ \quad \mathsf{Obs\_Congr}\ E1\ E1' \wedge \mathsf{Obs\_Congr}\ E2\ E2' \supset \\ \quad \mathsf{Obs\_Congr}\ (\mathsf{sum}\ E1\ E2)\ (\mathsf{sum}\ E1'\ E2') \end{array}$$

can be used to split the verification in smaller proofs about the components (represented by the premisses of the above theorem) of a summation agent. Given the goal (2.1), the inference rule OBS_CONGR_PRESD_BY_SUM is applied backward, after using the commutativity law (A2) once, in order to get the subgoals in the right order. The two following subgoals are obtained:

$$\begin{array}{l} ?\vdash \mathsf{Obs\_Congr} \\ \quad (\tau.\, bw.\, ((\overline{p}.\, br.\, er.\, \overline{v}.\, Reader \mid (v.\, Sem \mid ew.\, \overline{v}.\, Writer))\backslash\{p, v\})) \\ \quad (\tau.\, bw.\, ew.\, Impl) \end{array}$$

$$\begin{array}{l} ?\vdash \mathsf{Obs\_Congr} \\ \quad (\tau.\, br.\, ((er.\, \overline{v}.\, Reader \mid (v.\, Sem \mid \overline{p}.\, bw.\, ew.\, \overline{v}.\, Writer))\backslash\{p, v\})) \\ \quad (\tau.\, br.\, er.\, Impl) \end{array}$$

Once more, it is possible to transform both subgoals before using the lazy expansion technique. The substitutivity rule of the prefix operator with respect to observation congruence, formalised in HOL by the theorem

$$\begin{array}{l} \mathrm{SUBST\_PREFIX}: \\ \vdash \forall E\ E' :: \mathsf{Is\_Agent}.\ \mathsf{Obs\_Congr}\ E\ E' \supset (\forall u.\ \mathsf{Obs\_Congr}\ (\mathsf{prefix}\ u\ E)\ (\mathsf{prefix}\ u\ E')) \end{array}$$

can be applied backward twice; for example, the first subgoal above becomes:

$$?\vdash \mathsf{Obs\_Congr}\ ((\overline{p}.\, br.\, er.\, \overline{v}.\, Reader \mid (v.\, Sem \mid ew.\, \overline{v}.\, Writer))\backslash\{p, v\})\ (ew.\, Impl)$$

and is then solved using the expansion theorem, the τ-law (T1) and the definition of *Impl*. Transformations similar to the ones already shown are produced with the

difference that reasoning is carried out separately on the (smaller) components of the original (larger) system. This is the advantage of dealing with a behavioural congruence, as it allows the analysis of an entire composite system to be reduced to that of its (simpler) components. Moreover, proofs become more readable because expressions of smaller size are considered at each step. From the HOL implementation point of view, this can also have effects on the performance of tactics like `OC_EXP_THM_TAC`, which implements the expansion theorem for observation congruence by looking for subterms to be rewritten starting from the deepest ones.

2.4.2 Checking Modal Properties

The correctness of the implementation *Impl* of the reader-writer system can also be checked against a modal formula, which provides a partial specification of the system by expressing properties that *Impl* must have. The basic safety property that the reader-writer system must satisfy is *mutual exclusion*, namely the reading and writing tasks cannot be carried out on the common resource at the same time. This means that, when either the reader or the writer is working on the resource, the other agent cannot access it. One way to express this mutual exclusion is through the following modal formula Φ, where the notation given at the end of Section 2.1.4 is adopted to express the notion of necessity of an action:[17]

$$[\tau]\,([br]\,(\langle - \rangle \mathrm{tt} \wedge [-er]\,\mathrm{ff}) \wedge [bw]\,(\langle - \rangle \mathrm{tt} \wedge [-ew]\,\mathrm{ff}))$$

This formula says that, after a synchronisation has taken place (i.e. the access to the resource has been given to either reader or writer), whenever the action br is performed (i.e. the reader has obtained access to the resource), then the next action must be er and whenever the action bw is executed, then the next action must be ew.[18] Another modal formula that expresses mutual exclusion by stating which actions cannot occur, rather than which actions must occur, is $[\tau]\,([br]\,[bw]\,\mathrm{ff} \wedge [bw]\,[br]\,\mathrm{ff})$. This formula says that an action br cannot be followed by an action bw and vice versa.

Let us prove that the agent *Impl* has the property Φ, namely $Impl \models \Phi$. In order to check this in HOL, the derived modal operator of necessity is first defined:

$$\forall u.\ \mathsf{nec}\ u\ =\ \mathsf{conj}\ (\mathsf{dmd}\ \mathsf{Univ}\ \mathsf{tt})\ (\mathsf{box}\ (\mathsf{Univ} - \{u\})\ \mathsf{ff})$$

where $\mathsf{Univ} : (action)set$ denotes the universe of actions. The satisfaction relation for the necessity operator, $E \models \mathsf{nec}\ u$, is derived from the ones for the modal operators

[17] From now on, braces of singleton sets of actions inside the box and diamond operators will be omitted.

[18] Note that a weaker formulation $[\![A]\!]$ and $\langle\langle A \rangle\rangle$ of the modal operators could be used, which is interpreted on the transition relation $\Longrightarrow$, thus allowing τ-actions to be ignored. Moreover, the above safety property can be expressed to hold forever, namely over the infinite recursive behaviour of the reader-writer system. This requires an extension to the modal logic, i.e. the μ-calculus [118], which is not treated here.

in the definition of nec and is given by the following theorem directly in terms of the transitions of the agent E:

$$\begin{aligned}\text{SAT\_nec:}\ \vdash\ & \forall E :: \mathsf{Is\_Agent}.\ \forall u.\\ & \mathsf{Sat}\ E\ (\mathsf{nec}\ u) =\\ & (\exists E' :: \mathsf{Is\_Agent}.\ \mathsf{Trans}\ E\ u\ E') \wedge\\ & \neg(\exists u'.\ \exists E' :: \mathsf{Is\_Agent}.\ \neg(u' = u)\ \wedge\ \mathsf{Trans}\ E\ u'\ E')\end{aligned}$$

This theorem asserts that an agent E necessarily performs an action u if and only if E can perform u and cannot perform any other action different from u.

The HOL proof of the goal $\vdash\ Impl \models \Phi$ is outlined below. Once more, the usual CCS notation is used for the agents and the satisfaction relation, while the HOL predicate Trans is used to denote the transition relation. Assumptions of the form 'Is_Agent E' and subgoals for checking that a given specification is an agent are left out. The HOL code for this proof is reported in Appendix B.2.

Because the modal logic is interpreted on transition systems, the proof technique for verifying that an agent has a given property mostly involves computing the agent transitions and showing that they are (or are not) allowed by the modal formula. In HOL this often means proving that there is an inconsistency between transitions. The proof of $Impl \models \Phi$ starts by rewriting the goal with the definition of Φ:

$$\vdash\ Impl \models [\tau]([br](\langle - \rangle \mathbf{tt} \wedge [-er]\mathbf{ff}) \wedge [bw](\langle - \rangle \mathbf{tt} \wedge [-ew]\mathbf{ff}))$$

By applying the satisfaction relation for the box operator (Section 2.1.4), the new goal is:

$$\begin{aligned}\vdash\ & \forall u.\ \forall E' :: \mathsf{Is\_Agent}.\\ & u \in \{\tau\}\ \wedge\ \mathsf{Trans}\ Impl\ u\ E'\ \supset\\ & E' \models [br](\langle - \rangle \mathbf{tt} \wedge [-er]\mathbf{ff}) \wedge [bw](\langle - \rangle \mathbf{tt} \wedge [-ew]\mathbf{ff})\end{aligned}$$

The condition $u \in \{\tau\}$ is true if and only if $u = \tau$ and the transitions of *Impl* can be derived using the conversion `Run_CONV` (Section 2.2.2):

$$\begin{aligned}\vdash\ & \forall u.\ \forall E' :: \mathsf{Is\_Agent}.\\ & (u = \tau) \wedge\\ & ((u = \tau)\ \wedge\ (E' = (Reader\ |\ (v.\ Sem\ |\ bw.\ ew.\ \overline{v}.\ Writer))\backslash\{p, v\})\ \vee\\ & (u = \tau)\ \wedge\ (E' = (br.\ er.\ \overline{v}.\ Reader\ |\ (v.\ Sem\ |\ Writer))\backslash\{p, v\}))\ \supset\\ & E' \models [br](\langle - \rangle \mathbf{tt} \wedge [-er]\mathbf{ff}) \wedge [bw](\langle - \rangle \mathbf{tt} \wedge [-ew]\mathbf{ff})\end{aligned}$$

Given any action u and agent E', let us assume the antecedent of the implicational goal to be true and try to prove its conclusion. In the HOL jargon, this means that the antecedent of the implication is moved into the assumption list of the goal by stripping the universal quantifiers and the implication. Moreover, the conjunctions in

the antecedent are split and, as the antecedent contains a disjunction, two subgoals are generated:

$$\{\, u = \tau \;;\; E' = (\mathit{Reader} \mid (v.\,\mathit{Sem} \mid bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\,\} \qquad (2.2)$$
$$\vdash\; E' \models [br]\,(\langle - \rangle \mathbf{tt} \wedge [-er]\,\mathbf{ff}) \wedge [bw]\,(\langle - \rangle \mathbf{tt} \wedge [-ew]\,\mathbf{ff})$$

$$\{\, u = \tau \;;\; E' = (br.\,er.\,\overline{v}.\,\mathit{Reader} \mid (v.\,\mathit{Sem} \mid \mathit{Writer}))\backslash\{p, v\}\,\} \qquad (2.3)$$
$$\vdash\; E' \models [br]\,(\langle - \rangle \mathbf{tt} \wedge [-er]\,\mathbf{ff}) \wedge [bw]\,(\langle - \rangle \mathbf{tt} \wedge [-ew]\,\mathbf{ff})$$

Let us prove (2.2). The satisfaction relation for conjunctive formulas can be applied, thus obtaining two subgoals:

$$\{\, u = \tau \;;\; E' = (\mathit{Reader} \mid (v.\,\mathit{Sem} \mid bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\,\} \qquad (2.4)$$
$$\vdash\; E' \models [br]\,(\langle - \rangle \mathbf{tt} \wedge [-er]\,\mathbf{ff})$$

$$\{\, u = \tau \;;\; E' = (\mathit{Reader} \mid (v.\,\mathit{Sem} \mid bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\,\} \qquad (2.5)$$
$$\vdash\; E' \models [bw]\,(\langle - \rangle \mathbf{tt} \wedge [-ew]\,\mathbf{ff})$$

The proof of subgoal (2.4) makes use of the satisfaction relation for the box operator:

$$\{\, u = \tau \;;\; E' = (\mathit{Reader} \mid (v.\,\mathit{Sem} \mid bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\,\}$$
$$\vdash\; \forall u.\, \forall E'' :: \mathsf{Is\_Agent}.\; u \in \{br\} \wedge \mathsf{Trans}\; E'\; u\; E'' \supset E'' \models \langle - \rangle \mathbf{tt} \wedge [-er]\,\mathbf{ff}$$

As before, let us rewrite the condition $u \in \{br\}$, assume the antecedent of the implicational goal and try to prove its conclusion. Using the conversion `Run_CONV`, the transitions of agent E' can be computed and added to the assumption list. This yields the following goal:

$$\{\, u = \tau \;;\; E' = (\mathit{Reader} \mid (v.\,\mathit{Sem} \mid bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\} \;;$$
$$u' = br \;;\; \mathsf{Trans}\; E'\; u'\; E'' \;;$$
$$u' = bw \;;\; E'' = (\mathit{Reader} \mid (v.\,\mathit{Sem} \mid ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\,\}$$
$$\vdash\; E'' \models \langle - \rangle \mathbf{tt} \wedge [-er]\,\mathbf{ff}$$

The assumptions of this goal are inconsistent, as the action u' has been derived to be equal to the two different constants br and bw. The conversion `Action_EQ_CONV` that decides equality of pure actions is used to derive a theorem whose conclusion is false, thus solving subgoal (2.4).

The proof of subgoal (2.5) is slightly different as E' can execute the action bw in the modal formula that E' must satisfy. After applying the satisfaction relation for the box operator, the modal formula $\langle - \rangle \mathbf{tt} \wedge [-er]\,\mathbf{ff}$ (expressing the necessity of the action er) is rewritten using the theorem SAT_nec. The outermost connectives

are then stripped, thus producing two subgoals:

$$\{\ u = \tau\ ;\ E' = (\mathit{Reader}\ \mid\ (v.\,\mathit{Sem}\ \mid\ bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\ ; \qquad (2.6)$$
$$u' = bw\ ;\ \mathsf{Trans}\ E'\ u'\ E''\ \}$$
$$\vdash\ \exists E' :: \mathsf{Is\_Agent}.\ \mathsf{Trans}\ E''\ ew\ E'$$

$$\{\ u = \tau\ ;\ E' = (\mathit{Reader}\ \mid\ (v.\,\mathit{Sem}\ \mid\ bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\ ; \qquad (2.7)$$
$$u' = bw\ ;\ \mathsf{Trans}\ E'\ u'\ E''\ ;$$
$$\neg(u'' = ew)\ ;\ \mathsf{Trans}\ E''\ u''\ E'''\ \}$$
$$\vdash\ \mathsf{F}$$

where the last two assumptions and the conclusion F of subgoal (2.7) are obtained by stripping the outermost connectives of the second conjunct in the right-hand side of the theorem SAT_nec and instantiating its existentially quantified variables. The agent E' can execute bw and evolve into $(\mathit{Reader} \mid (v.\,\mathit{Sem} \mid ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}$. Thus, the existentially quantified variable in subgoal (2.6) can be instantiated with $(\mathit{Reader} \mid (v.\,\mathit{Sem} \mid \overline{v}.\,\mathit{Writer}))\backslash\{p, v\}$:

$$\{\ u = \tau\ ;\ E' = (\mathit{Reader}\ \mid\ (v.\,\mathit{Sem}\ \mid\ bw.\,ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\ ;$$
$$u' = bw\ ;\ \mathsf{Trans}\ E'\ u'\ E''\ ;$$
$$E'' = (\mathit{Reader}\ \mid\ (v.\,\mathit{Sem}\ \mid\ ew.\,\overline{v}.\,\mathit{Writer}))\backslash\{p, v\}\ \}$$
$$\vdash\ \mathsf{Trans}\ E''\ ew\ (\mathit{Reader}\ \mid\ (v.\,\mathit{Sem}\ \mid\ \overline{v}.\,\mathit{Writer}))\backslash\{p, v\}$$

The goal is solved by applying the transition rules for the restriction, parallel and prefix operators. Subgoal (2.7) is solved by deriving a contradiction about the value of action u'', thus concluding the proof of subgoals (2.5) and (2.2). Subgoal (2.3) is proved with proof steps similar to the ones for subgoal (2.2).

A concluding remark concerns the predicate Is_Agent. As it can be seen in Appendix B.2, when applying theorems/rules with restricted quantification either in backward or in forward manner, it must be verified that the specifications under consideration are agents. Although there exist HOL conversions and tactics to deal with restricted quantified variables, using restricted quantification leads to longer and less readable proofs. The verification of such a predicate can be automated to a certain extent, like in the substitution tactics/conversions for the behavioural equivalences (as discussed in Section 2.3.1). More practically, whenever applications specified in CCS are considered, it can be assumed that all specifications are actually agents. The situation might be different when performing meta-reasoning on the process calculus. In the rest of the dissertation, unless where necessary, HOL proofs will be abstracted from the issue of checking the predicate Is_Agent for the given specifications.

2.5 Summary

This chapter has presented the HOL formalisation of some components of the theory for the CCS process calculus. The syntax, various semantics, the laws for observation congruence and a modal logic have been embedded in higher order logic. This is done using the primitive and derived rules of definition provided by the HOL system and by deriving all other properties and theorems by formal proof. Actions, agent expressions and modal formulas are mechanised as concrete recursive data types, and then several functions are defined by primitive recursion over these types. New types, such as the one for relabelling functions, are encoded by means of the type abstraction mechanism. The derived rule for inductive definitions is used to define the transition relation over agent expressions, and then bisimulation and behavioural equivalences are introduced into the HOL logic by making constant definitions and using restricted quantification over agents. Properties about the various types, transition relations, bisimulations, behavioural equivalences and their laws, are formally derived starting from the mechanisation of the syntax and the various semantics.

This methodology allows users to take advantage of all components of the formal theory for process calculi in a unified framework and to define their own verification strategies. As an example, two different proof methods are adopted when showing the correctness of the reader-writer system (Section 2.4 and Appendix B). The first proof is based on equational reasoning, and the axiomatic characterisation of observation congruence is used by applying the behavioural laws with a lazy expansion strategy. The second proof is about checking modal properties, thus an operational technique based on the labelled transition relation is mostly used.

Various 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 the HOL mechanisation of CCS. Different degrees of automation can be chosen, in the sense that conversions and tactics can be defined to perform only one or more computation steps. The HOL proofs shown in Appendix B are fairly interactive, but "bigger" tactics could be defined by composing the smaller ones, such as those for the expansion theorem, unfolding, resolution, etc. In the case of finite state specifications, these tactics would be able to automatically solve the original goal.

It is fair to note that the reader-writer example used to show how reasoning about pure agents can be performed in HOL is just a toy system. The verification of both the observation congruence between implementation and specification and the mutual exclusion property can be trivially checked by any finite state machine tool. The automaton associated to the reader-writer system is really small, thus verification is very fast. Nevertheless, in its simplicity this verification example introduces the kind of formal proofs which can be carried out in the HOL-CCS environment and shows how different proof styles (such as the two proofs in Section 2.4.1) can be used

when considering a behavioural congruence. The example and, more generally, the HOL formalisation of pure CCS also give evidence of the various technical details that need to be formalised when embedding a calculus in a theorem prover, such as the predicate Is_Agent and the notation for finite indexed agent summations.

The reader-writer system can be easily extended to the general case of n processes competing for access to a common resource. Particular access orderings, e.g. round robin access, can also be specified, thus yielding typical scheduling systems such as Milner's scheduler [97]. In general, the correctness of inductively defined systems, both finite and infinite state, is shown using induction. Proofs by mathematical induction of the correctness of specifications will be presented in Chapter 3.

The law (A13) of unique fixpoint induction has not yet been derived in HOL, while this is provided by axiomatic tools such as PAM and the embedding of μCRL in Coq. Therefore, the HOL proofs of behavioural congruences start from a goal to which unique fixpoint induction has already been applied by hand. The formal theory for deriving the law (A13) is currently being mechanised in HOL.

Extensions to the Hennessy-Milner logic and the subset of CCS can be embedded in the HOL system. For example, more expressive temporal logics can be represented in higher order logic and proof tools, such as the *tableau system* for the μ-calculus [118, 16], can be soundly mechanised. Related work about embedding temporal logics in higher order logic includes various mechanisations of TLA in HOL [24, 126] and in Isabelle [75], and a formalisation of ACTL in HOL [46, 80]. As far as extensions to the CCS calculus are concerned, in the next chapter the pure syntax is extended to include agent constants and (possibly infinite) indexed summations.

Chapter 3

Polymorphic Versions of Pure CCS in HOL

This chapter presents two new HOL formalisations of the pure CCS calculus. The aim is twofold: (i) to mechanise the pure calculus as given in [97] and (ii) to mechanise it in such a way that value-passing agent expressions, which will be considered in Chapter 4, can be translated into pure agent expressions.

The first objective is achieved in two steps. First, agent constants and defining equations are formalised in HOL, thus replacing the *rec*-notation in the CCS syntax. Defining agents with infinite behaviour through systems of recursive equations also gives the possibility of specifying (pure) parameterised agents easily. Pure agent constants A are explicitly allowed to have arity $n \geq 0$, thus agent constants of the form $A(e_1, \ldots, e_n)$ can be defined in HOL. This leads to a polymorphic type for pure expressions which is parameterised on the type of the parameters of agent constants. Based on this new formalisation of the pure calculus, the correctness of an infinite counter is verified in HOL by proving that two different descriptions of the counter are equivalent modulo observation congruence and that the specification of the counter has a given modal property.

Second, an operator of indexed summation is included in the HOL type for pure agent expressions, thus replacing the binary summation operator. This achieves aim (i) above and leads to a new formalisation of the expansion law based on the new summation operator. Moreover, it is a step towards aim (ii), due to the translation of value-passing input expressions into (possibly infinite) indexed summations. Finally, pure labels are not assumed to be strings any more but can be of any type, namely the HOL types for pure labels and actions also become polymorphic.

All these transformations on the HOL formalisation of the pure syntax result in a polymorphic type for pure agent expressions, which is parameterised on the types for the indexing domain of indexed summations, the parameters in parameterised agent constants and the labels (or ports). This new formalisation allows value-passing agent expressions to be translated into pure expressions as it will be shown in Chapter 4.

3.1 Recursive Agent Definitions

So far, the only way of defining agents with infinite behaviour has been through the *rec*-notation (Section 2.1.1). This differs from the notation based on agent constants and (recursive) defining equations used by Milner when introducing his calculus [97, 98]. In this section a brief comparison between the two notations is given by means of a few examples, and the formalisation in HOL of agent constants and recursive definitions is then presented.

3.1.1 A Few Examples

As Taubner says in [120], there are two common notations for recursion in process algebras. One defines recursive agents through a system of recursive equations. This notation is legible, concise and very convenient from a practical point of view. The other is the *rec*-notation which, from a theoretical point of view, is closer to λ-calculus and has the advantage that all information is encoded in the *rec*-terms. This means that one clause in the syntax is sufficient for recursion and no system of defining equations is needed. However, this notation can easily become illegible.

Let us consider the following CCS agents which will also be used when defining an infinite counter in Section 3.2.1. An implementation of the counter can be built by "linking" together (several copies of) a cell C and an agent B defined as follows [97]:

$$\begin{aligned} C &\stackrel{\text{def}}{=} up.(C^\frown C) + down.D \\ D &\stackrel{\text{def}}{=} \overline{d}.C + \overline{a}.B \\ B &\stackrel{\text{def}}{=} up.(C^\frown B) + around.B \end{aligned}$$

where the linking operator '$^\frown$' is a derived CCS operator defined in terms of relabelling, parallel composition and restriction (its definition is not relevant here and is given in Section 3.2.1). The agents C, D, B are mutually recursively defined through a (finite) system of recursive equations. However, these agents can also be easily expressed as the following closed *rec*-terms:

$$\begin{aligned} C &\stackrel{\text{def}}{=} \mathbf{rec}\, X.\,(up.(X^\frown X) + down.(\overline{d}.X + \overline{a}.(\mathbf{rec}\, Y.\,(up.(X^\frown Y) + around.Y)))) \\ B &\stackrel{\text{def}}{=} \mathbf{rec}\, Y.\,(up.(C^\frown Y) + around.Y) \\ D &\stackrel{\text{def}}{=} \overline{d}.C + \overline{a}.B \end{aligned}$$

These agents can be simply introduced in HOL through constant definitions (Section A.3). The definition of D is the same as the one given initially. As far as the agents C and B are concerned, as it is more convenient to rewrite them using their definitions as recursive equations, these can be derived by formal proof up to the behavioural equivalence under consideration. In the case of observation congruence,

this means proving the following theorems in HOL:

$$\vdash \mathsf{Obs\_Congr}\ C\ (up.\,(C^\frown C) + down.\,D)$$
$$\vdash \mathsf{Obs\_Congr}\ B\ (up.\,(C^\frown B) + around.\,B)$$

and then using such theorems for rewriting C and B up to observation congruence.

Unfortunately, it is not always so simple to write the *rec*-terms corresponding to a system of recursive equations. Let us consider the following system which defines the agent constants A, B and C:

$$A \stackrel{\text{def}}{=} a'.\,A + b.\,B + c.\,C$$
$$B \stackrel{\text{def}}{=} a.\,A + b'.\,B + c.\,C$$
$$C \stackrel{\text{def}}{=} a.\,A + b.\,B + c'.\,C$$

A *rec*-term that denotes the agent A is the following:

$$\begin{array}{l}\mathbf{rec}\,A.\\ \quad a'.\,A +\\ \quad b.\,(\mathbf{rec}\,B.\,a.\,A + b'.\,B + c.\,(\mathbf{rec}\,C.\,a.\,A + b.\,B + c'.\,C)) +\\ \quad c.\,(\mathbf{rec}\,C.\,a.\,A + b.\,(\mathbf{rec}\,B.\,a.\,A + b'.\,B + c.\,C) + c'.\,C)\end{array}$$

where A, B, C are now denoting the agent variables bound by the **rec** operators. In this term some recursive subexpressions need to be duplicated due to the scoping rules of *rec*-terms. For instance, in the last summand of the body of the outermost **rec** operator, the recursive term denoting the agent B must be given explicitly again, because it is not in the "right" scope and thus cannot be simply referred to with an occurrence of the corresponding bound variable.[1] The *rec*-term above is just one of several *rec*-terms that denote A and it is clear that the transformation from a system of recursive equations to the *rec*-notation can easily lead to very cumbersome *rec*-terms.

In [120] Taubner has shown that the two notations for recursion in process algebras are equivalent. In fact, he gives a (back and forth) translation between the two notations and proves that they are strongly equivalent. Hence, one can define agent constants through systems of (mutually) recursive equations in a more convenient way than using the *rec*-notation. Given a finite set $\mathcal{K}$ of agent identifiers (or constants) and the set *terms* of terms generated by a CCS-like syntax, a mapping $\Delta : \mathcal{K} \to terms$ formalises the defining equations of the agent constants. Both the operational semantics of the agents in *terms* and the translation are parameterised on the function Δ, that can be seen as the environment containing agent identifiers and corresponding agents, in which a process algebra term is evaluated, executed and verified.

[1] This phenomenon is called *horizontal sharing* and it is well known that the μ-calculus ($\mu X.\,E$ is just another notation for $\mathbf{rec}\,X.\,E$) is not able to cope with it [70].

However, one would also like to be able to deal with parameterised agent constants. Taubner's translation considers "parameterless recursive processes" and the set $\mathcal{K}$ of agent identifiers is assumed to be finite. His translation and the strong equivalence between the two notations for recursion hold in the framework of finite state agents, i.e. agents whose labelled transition systems have a finite number of states. In the presence of parameterised recursive agents, a set of defining equations may well define an infinite family of agents depending on the domain of the parameters, thus leading to an infinite set $\mathcal{K}$ of agent constants.

Let us recall the defining equations that specify the behaviour of a buffer of capacity $n \geq 1$ (Section 1.1.1):

$$\begin{aligned} \mathit{Buffer}_n(0) &\stackrel{\text{def}}{=} \mathit{in}.\,\mathit{Buffer}_n(1) \\ \mathit{Buffer}_n(k) &\stackrel{\text{def}}{=} \mathit{in}.\,\mathit{Buffer}_n(k{+}1) \;+\; \overline{\mathit{out}}.\,\mathit{Buffer}_n(k{-}1) \qquad (0 < k < n) \\ \mathit{Buffer}_n(n) &\stackrel{\text{def}}{=} \overline{\mathit{out}}.\,\mathit{Buffer}_n(n{-}1) \end{aligned}$$

Such a specification is parameterised on the capacity n of the buffer and the number k of the values presently stored in the buffer. Whenever the capacity n is fixed, the above specification can also be given through a *rec*-term. In fact, Buffer_n defines a finite state agent for any fixed n, as the other parameter k can only range between 0 and the given n. For example, if $n = 3$ a *rec*-term for $\mathit{Buffer}_3(0)$ is as follows:

$$\mathbf{rec}\, B_{30}.\, \mathit{in}.\, (\mathbf{rec}\, B_{31}.\, \mathit{in}.\, (\mathbf{rec}\, B_{32}.\, \mathit{in}.\, \overline{\mathit{out}}.\, B_{32} \;+\; \overline{\mathit{out}}.\, B_{31}) \;+\; \overline{\mathit{out}}.\, B_{30})$$

where B_{3k} $(0 \leq k \leq 3)$ denotes an agent variable bound in a *rec*-term.

Let the behaviour of a counter be specified by means of the following defining equations [97, 118] (this specification will be analysed in Section 3.2):

$$\begin{aligned} \mathit{Counter}_0 &\stackrel{\text{def}}{=} \mathit{up}.\,\mathit{Counter}_1 \;+\; \mathit{around}.\,\mathit{Counter}_0 \\ \mathit{Counter}_n &\stackrel{\text{def}}{=} \mathit{up}.\,\mathit{Counter}_{n+1} \;+\; \mathit{down}.\,\mathit{Counter}_{n-1} \qquad (n > 0) \end{aligned}$$

The agent expression $\mathit{Counter}_n$ is parameterised on $n \in \mathbb{N}$, which represents the "state" of the counter, i.e. how many items are currently being counted. Actually, $\mathit{Counter}_n$ represents an infinite family of agents, one agent for each number n, which are mutually recursively defined and different from each other. The counter is an example of systems with evolving structure; it is an infinite state system and there is no *rec*-term that denotes $\mathit{Counter}_n$ which is "finite", in the sense that it has a finite number of subterms.

In Chapter 4 the value-passing version of the CCS calculus will be considered. In the value-passing framework, specifications can also be parameterised on data which are exchanged during communication. Whenever the value domain is infinite, value-passing specifications denote infinite state agents. For example, a storage register holding a value y is defined in [97] as follows:

$$\mathit{Reg}(y) \;\stackrel{\text{def}}{=}\; \mathit{put}(x).\,\mathit{Reg}(x) \;+\; \overline{\mathit{get}}(y).\,\mathit{Reg}(y)$$

The register is parameterised on a value denoted by the variable y, and the value domain over which the variables x and y range may be infinite, thus resulting in an infinite state agent associated to $Reg(y)$.

The following section illustrates how parameterised agent constants and their defining equations can be embedded in the HOL system.

3.1.2 Formalising Recursive Agent Definitions in HOL

When moving from the *rec*-notation to systems of recursive equations, the syntax of the pure calculus (Section 2.1.1) becomes the following:[2]

$$E ::= \mathbf{nil} \mid X \mid A(e_1, \ldots, e_n) \mid u.E \mid E + E \mid E \mid E \mid E \backslash L \mid E[f]$$

where A ranges over a set of (possibly parameterised) agent constants $\mathcal{K}$. In a way slightly different from the presentation of the pure calculus in [97], agent constants are explicitly allowed to have parameters. Thus, the more general expression $A(e_1, \ldots, e_n)$ is used to denote an agent constant rather than just A or $A_{e_1,\ldots,e_n} \in \mathcal{K}$. Each agent constant A with arity $n \geq 0$ (i.e. the number of its parameters) has a defining equation $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$ that gives the agent P (containing no free variables except $x_1, \ldots, x_n$) associated to A.

Every parameterised agent is identified through its name, which is assumed to be a string of characters, and its parameters. Parameters can be of any type and the definition of the type of pure agent expressions thus becomes polymorphic, as it is parameterised on a type variable β that represents the parameters' type. The type $(\beta)CCS$ is defined in HOL by giving the following signature to the recursive type definition mechanism:

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

The theorem of higher order logic that characterises the type $(\beta)CCS$ is similar to the one in Section 2.2.1, apart from the clause for the constructor rec which is now replaced by that for conp. A term conp s x denotes a parameterised agent whose name is given by the string s and whose parameters are denoted by x. The type variable β can be instantiated to represent any structure for the parameters. They

[2]Note that the recursion operator is replaced by agent constants, while agent variables (which in the previous version of CCS were either free or bound by a recursion operator), are maintained in the definition. This still gives the possibility of writing open agent expressions in the new signature.

can be a single natural number like in the counter specification, a pair of natural numbers like in the buffer example, pairs or lists of any objects and any other data type (possibly polymorphic and recursive).

A parameterless agent constant is identified simply through its name. A derived constructor con is introduced for defining parameterless constants in terms of the operator conp:

$$\forall s.\ \mathsf{con}\ s = \mathsf{conp}\ s\ \mathsf{ARB}$$

where the type of the constant $\mathsf{ARB} = \varepsilon x{:}\,\alpha.\mathsf{T}$ (from the HOL theory `bool`) is instantiated to the parameters' type β.

The rule for the transition relation for agent constants is as follows [97]:

$$\frac{P \xrightarrow{u} P'}{A \xrightarrow{u} P'} \qquad A \stackrel{\text{def}}{=} P$$

This rule says that the transitions of a constant A are the transitions of the agent P, provided there exists a defining equation $A \stackrel{\text{def}}{=} P$. The defining equations for (possibly parameterised) agent constants are introduced in HOL as a new (polymorphic) type using the rule of type definition (Section A.3). The abstract type $(\beta)Def\_Fun$ is defined as the subset of the type of all functions $Df : string \to \beta \to (\beta)CCS$ such that $Df\ s\ x$ is an agent for every constant identified through name s and parameters x. The characteristic function of such a subset is given by the predicate Is_Def_Fun:

$$\forall\, Df.\ \mathsf{Is\_Def\_Fun}\ Df\ =\ (\forall s\ x.\ \mathsf{Is\_Agent}\ (Df\ s\ x))$$

The type definition mechanism derives a bijection between the new type $(\beta)Def\_Fun$ and the subset of the function type $string \to \beta \to (\beta)CCS$, whose elements satisfy Is_Def_Fun. In what follows, defining equations will be simply given through raw functions $Df : string \to \beta \to (\beta)CCS$ by defining a constructor Def_Fun which creates a defining equations function by simply abstracting the given raw function:

$$\forall\, Df.\ \mathsf{Def\_Fun}\ Df = \mathsf{ABS\_Def\_Fun}\ Df$$

where ABS_Def_Fun is the abstraction function for the type $(\beta)Def\_Fun$.[3] Given a defining equations function $Df : (\beta)Def\_Fun$, a function Lookup which returns the agent associated to a given constant is defined as follows:

$$\forall\, Df\ s\ x.\ \mathsf{Lookup}\ Df\ s\ x = (\mathsf{REP\_Def\_Fun}\ Df)\ s\ x$$

where REP_Def_Fun is the representation function for the type $(\beta)Def\_Fun$. Thus, the function Lookup simply takes the function that implements Df and applies it to

[3]Other ways of representing defining equations are obviously possible. For example, lists of equations $A = P$ which better resemble the usual CCS notation could be used and the constructor Def_Fun would take a list of equations and return a function in $(\beta)Def\_Fun$.

the name and parameters of the given agent constant. It can then be proved that the agent expression associated to any constant conp $s\ x$ is indeed an agent:

$$\text{Is\_Agent\_Lookup:}\ \vdash\ \forall\, Df\ s\ x.\ \mathsf{Is\_Agent}\,(\mathsf{Lookup}\ Df\ s\ x)$$

A defining equations function Df is then given as a parameter to the definition of the transition relation, so that the labelled transitions $\xrightarrow{u}_{Df}$ are now computed with respect to a particular Df. In this way, the transition relation Trans actually defines a class of transition systems parameterised on the defining equations. Behavioural equivalences and modal logics are consequently parameterised on the mapping Df as well. The rule of the transition relation for agent constants in HOL is as follows:

$$\text{CONP:}\ \vdash \forall Df\ s\ x\ u\ E.\ \mathsf{Trans}\ Df\ (\mathsf{Lookup}\ Df\ s\ x)\ u\ E\ \supset\ \mathsf{Trans}\ Df\ (\mathsf{conp}\ s\ x)\ u\ E$$

This theorem says that the transitions of an agent constant conp $s\ x$ are the transitions of the agent associated to the name s and the parameters x through the defining equations function Df. The rule for parameterless agent constants is derived from the rule CONP and the definition of the constructor con:

$$\text{CON:}\ \vdash \forall Df\ s\ u\ E.\ \mathsf{Trans}\ Df\ (\mathsf{Lookup}\ Df\ s\ \mathsf{ARB})\ u\ E\ \supset\ \mathsf{Trans}\ Df\ (\mathsf{con}\ s)\ u\ E$$

The equational versions TRANS_CONP_EQ and TRANS_CON_EQ of CONP and CON respectively, in which implication is replaced by equality, are derived as well.

In what follows, the mechanisation in HOL of the defining equations for the agents A, B, C and for the buffer (given in the previous section) is described. The defining equations of the specification and implementation of the counter will be formalised in HOL in Section 3.2.1.

Parameterless agents

The agent constants A, B, C of Section 3.1.1 are parameterless and defined through mutual recursion. They can be introduced in HOL by defining the following abbreviations by constant definition:

$$\mathsf{A}\ =\ \mathsf{con}\ A \qquad\qquad \mathsf{B}\ =\ \mathsf{con}\ B \qquad\qquad \mathsf{C}\ =\ \mathsf{con}\ C$$

where A, B and C are constants of type *string*. The function $\mathsf{Df} : string \to \beta \to (\beta)CCS$ that represents the defining equations for A, B and C is then defined:

$$\begin{aligned}\mathsf{Df} = (\lambda\, s\ (x{:}\,\beta).\\ &((s = A) \Rightarrow (a'.\,\mathsf{A}\ +\ b.\,\mathsf{B})\ +\ c.\,\mathsf{C}\ \mid\\ &\ ((s = B) \Rightarrow (a.\,\mathsf{A}\ +\ b'.\,\mathsf{B})\ +\ c.\,\mathsf{C}\ \mid\\ &\ \ ((s = C) \Rightarrow (a.\,\mathsf{A}\ +\ b.\,\mathsf{B}) + c'.\,\mathsf{C}\ \mid\ \mathsf{ARBag}))))\end{aligned}$$

where a, a', b, b', c and c' are constants of type *string* and $\mathsf{ARBag} = \varepsilon P.\ \mathsf{Is\_Agent}\ P$ denotes an arbitrary agent. Df is automatically checked to be a defining equations

function using the function PROVE_IS_DEF_FUN that, given the definition of Df and the definitions A_DEF, B_DEF and C_DEF for A, B and C respectively, proves that Df indeed represents a defining equations function:

```
#let Def_thm = PROVE_IS_DEF_FUN Df [A_DEF; B_DEF; C_DEF];;
Def_thm = ⊢ Is_Def_Fun Df
```

In order to compute the transitions of the various agent constants, the agent Lookup (Def_Fun Df) s ARB needs to be evaluated for a given string s. This is done using the conversion CON_EVAL_CONV that, when provided with the function Df, the theorem Def_thm and the string identifying an agent constant, returns the agent associated to it. The following theorem is returned for the constant B:

$$\text{B\_THM: } \vdash \mathsf{Lookup}\,(\mathsf{Def\_Fun\ Df})\,B\ \mathsf{ARB} \;=\; (a.\,\mathsf{A} \;+\; b'.\,\mathsf{B}) \;+\; c.\,\mathsf{C}$$

By rewriting with the definition of B, the theorems TRANS_CON_EQ and B_THM, and using the conversion Run_CONV (Section 2.2.2), the following theorem is obtained:

$$\begin{array}{l}\vdash\ \forall u\,E.\\ \quad \mathsf{Trans}\,(\mathsf{Def\_Fun\ Df})\,\mathsf{B}\,u\,E\ =\\ \quad ((u=a)\ \wedge\ (E=\mathsf{A}))\ \vee\ ((u=b')\ \wedge\ (E=\mathsf{B}))\ \vee\ ((u=c)\ \wedge\ (E=\mathsf{C}))\end{array}$$

It asserts that the agent constant B can either perform the action a and then behave like A or perform the action b' and still behave like B or perform the action c and then behave like C.

The buffer

The defining equations define several agent constants parameterised on the capacity $n>0$ of the buffer and the number k of the items currently in the buffer ($0 \le k \le n$). A function $\mathsf{Buffer} : num \to num \to (num \times num)CCS$ that returns a parameterised agent constant whose name is *Buffer* and whose parameter is a pair of natural numbers is defined as follows:

$$\forall n\ k.\ \mathsf{Buffer}\ n\ k\ =\ \mathsf{conp}\ \mathit{Buffer}\ (n,k)$$

The following function represents the defining equations for the buffer specification:

```
Buffer_Df =
(λs x.
  ((s = Buffer) ⇒
   let (n,k) = (Fst x, Snd x) in
   (0<n ⇒
    ((k = 0) ⇒ in.(Buffer n 1) |
     ((0<k ∧ k<n) ⇒ in.(Buffer n (k+1)) + out.(Buffer n (k−1)) |
      ((k = n) ⇒ out.(Buffer n (n−1)) | ARBag))) |
    ARBag) |
   ARBag))
```

Buffer_Df is checked to be a defining equations function by means of a parameterised version of PROVE_IS_DEF_FUN, thus yielding a theorem `Buffer_Def_thm` similar to `Def_thm` above. Analogously, the conversion `CONP_EVAL_CONV` is defined to compute the agent associated to a parameterised agent constant. Differently from the parameterless counterparts which only deal with the name of the agent constant, the parameterised versions must also be provided with the actual parameters and a conversion for deciding equality between elements of the parameters' type. Moreover, further rewritings on the theorem returned by `CONP_EVAL_CONV` might be needed in order to get a reduced expression. For example, the agent denoting a buffer with capacity $n = 2$ and containing one item is given by the following theorem:

```
#REWRITE_RULE [LESS_0; NOT_SUC; LESS_SUC_REFL; INV_SUC_EQ; PRE]
# (CONP_EVAL_CONV Buffer_Df Buffer_Def_thm num_EQ_CONV
#  "‘Buffer‘" "(SUC(SUC 0), SUC 0)");;
⊢ Lookup(Def_Fun Buffer_Df)‘Buffer‘(SUC(SUC 0),SUC 0) =
   sum
   (prefix(label(name ‘in‘))(Buffer(SUC(SUC 0))(SUC(SUC 0))))
   (prefix(label(coname ‘out‘))(Buffer(SUC(SUC 0))0))
```

where `num_EQ_CONV` is a built-in conversion for proving (in)equality of constants in $\mathbb{N}$. Some rewritings with arithmetic theorems are necessary to reduce the theorem resulting from the evaluation performed by `CONP_EVAL_CONV`. Note that this evaluation step can provide information about incorrect actual parameters. For instance, if one attempts to assign a value greater than n to k, only arbitrary agents are returned when reducing the outcome of the evaluation with `CONP_EVAL_CONV`:

```
#REWRITE_RULE [NOT_SUC; NOT_SUC_LESS; SUC_ID]
# (CONP_EVAL_CONV Buffer_Df Buffer_Def_thm num_EQ_CONV
#  "‘Buffer‘" "(n,SUC n)");;
⊢ Lookup(Def_Fun Buffer_Df)‘Buffer‘(n,SUC n) =
   (0 < n ⇒ ARBag | ARBag)
```

The transitions for parameterised agents are computed in a way similar to the ones for parameterless constants, and then the laws for the behavioural equivalences are derived. The only relevant changes concern the laws for the recursion operator. For example, in the case of parameterised agents, the law UNFOLDING for observation congruence (Section 2.2.4) is replaced by the following law (Proposition 4(1) on page 65 in [97]):

OBSCONGR_CONP:

$\vdash \forall Df\ s\ x\ E.\ (\mathsf{Lookup}\ Df\ s\ x = E) \supset \mathsf{Obs\_Congr}\ Df\ (\mathsf{conp}\ s\ x)\ E$

By applying modus ponens between the parameterless version of the above theorem and B_THM, and then rewriting with the definition of B, the following observation

congruence is obtained:

$$\vdash \mathsf{Obs\_Congr}\ (\mathsf{Def\_Fun}\ \mathsf{Df})\ \mathsf{B}\ ((a.\,\mathsf{A} + b'.\,\mathsf{B}) + c.\,\mathsf{C})$$

This allows the constant B to be replaced by the agent associated to it modulo observation congruence.

The new version of the recursion law (A13), i.e. unique fixpoint induction, is as follows. Given an indexing domain I, let the expressions E_i and variables X_i $(i \in I)$ be such that $Fv(E_i) \subseteq \{X_i : i \in I\}$ and X_i be guarded and sequential in each E_i. Then, if $\tilde{P} \approx_c \tilde{E}\{\tilde{P}/\tilde{X}\}$ and $\tilde{Q} \approx_c \tilde{E}\{\tilde{Q}/\tilde{X}\}$, then $\tilde{P} \approx_c \tilde{Q}$.

Summing up, in this section the HOL formalisation of indexed agent constants P_i and parameterised agent constants $P(i)$ has been presented. The introduction of a mapping Df which associates an agent to each constant identified through its name and parameters enables one to deal with (mutually recursive) indexed or parameterised specifications using a convenient and concise notation. The formalisation of the CCS theory developed in Chapter 2 remains the same, except for the introduction of the mapping $Df : (\beta)Def\_Fun$ and for replacing the laws for rec with the laws for conp and con. Both finite state and infinite state (pure) agents can be soundly introduced in HOL and reasoning about them can be performed as shown in the following section.

3.2 Proving the Correctness of an Infinite Counter

This section illustrates how properties of infinite state agents can be analysed and verified in the HOL-CCS environment by presenting two proofs of correctness for the counter introduced in Section 3.1.1. The following family of agents $Counter_n$ $(n \in \mathbb{N})$ defines the behaviour of the counter:

$$\begin{aligned} Counter_0 &\stackrel{\text{def}}{=} up.\,Counter_1 + around.\,Counter_0 \\ Counter_n &\stackrel{\text{def}}{=} up.\,Counter_{n+1} + down.\,Counter_{n-1} \qquad (n > 0) \end{aligned}$$

Whenever the counter is in the initial state of a counting process $(n = 0)$, it can either count once by performing an action *up* and evolve to $Counter_1$, or execute an action *around* and still behave like the agent $Counter_0$. Whenever the counter is in a state $n > 0$, i.e. something has already been counted, it can either perform a further counting action *up* and evolve to the counter in the state $n+1$, or count down by executing an action *down* and move back to the agent $Counter_{n-1}$.

Two proofs of correctness of the counter are described in the following subsections. In the first proof a more detailed description of the counter is proved to meet the above specification by showing that the two descriptions are observation congruent. The second proof checks that the specification of the counter has a given modal property. The counter is a specification inductively defined over natural numbers and the proofs will make extensive use of mathematical induction.

3.2.1 Verifying Behavioural Equivalences

As mentioned in Section 3.1.1, an implementation of the counter can be built by "linking" together several copies of a cell C and one agent B defined as follows:

$$\begin{aligned} C &\stackrel{\text{def}}{=} up.(C^\frown C) + down.D \\ D &\stackrel{\text{def}}{=} \overline{d}.C + \overline{a}.B \\ B &\stackrel{\text{def}}{=} up.(C^\frown B) + around.B \end{aligned}$$

where the linking operator '$^\frown$' is a derived CCS operator defined in terms of relabelling, parallel composition and restriction. For any agent expressions E, E', the appropriate linking operation $E^\frown E'$ is defined by

$$(E\,[u'/u, a'/a, d'/d] \mid E'\,[u'/up, a'/around, d'/down])\backslash\{u', a', d'\}$$

such that $u', a', d' \notin Sort(E) \cup Sort(E')$, where *Sort* denotes the *syntactic sort* of agent expressions [97]. Given two agent expressions in parallel, linking them means renaming the labels of some of their actions with fresh names (hence the non-membership condition) and then making them private to the agent expressions by restricting over such new labels.[4]

The agent B cannot perform any action *down*, while a cell C can always perform the actions *up* and *down*. Thus, an implementation $Impl(n)$ of the counter in the state n can be defined by the following chain:

$$Impl(n) \stackrel{\text{def}}{=} \overbrace{C^\frown \ldots {}^\frown C^\frown}^{n\ times} B$$

The agent $Impl(n)$ is shown to be a correct implementation of the counter by proving that $Impl(n)$ and $Counter_n$ are observation congruent. By unique fixpoint induction this means proving that $Impl(n)$ "satisfies" the defining equations of $Counter_n$, thus resulting in the following observation congruences:

$$\begin{aligned} Impl(0) &\approx_c up.Impl(1) + around.Impl(0) \\ Impl(n) &\approx_c up.Impl(n+1) + down.Impl(n-1) \qquad (n>0) \end{aligned}$$

The proof is by induction on the parameter n and is sketched below in the way it has been formalised and carried out in HOL, following the guideline in [97]. The HOL code for this proof is reported in Appendix C.1.

The two descriptions of the counter are first introduced in HOL. The above linking

[4] In this particular application the linking combinator is treated as a binary operator defined over agent expressions only, given specific relabelling functions and restriction set. However, the general formulation of the linking operator could be defined for any agent expressions, relabelling functions and restriction set, thus resulting in an operator with five arguments.

operator Link is defined as a binary infix operator as follows:[5]

$$\vdash \forall E\ E'.$$
$$E\ \mathsf{Link}\ E' = ((E\ [u'/u, a'/a, d'/d])\ |\ (E'\ [u'/up, a'/around, d'/down]))\backslash\{u', a', d'\}$$

The agents C, D and B are introduced in HOL by the following definitions of agent constants of type $(num)CCS$:

$$\mathsf{C} = \mathsf{con}\,C \qquad \mathsf{D} = \mathsf{con}\,D \qquad \mathsf{B} = \mathsf{con}\,B$$

The families of agents $Counter_n$ and $Impl(n)$ are embedded in HOL through the functions Counter and Impl of type $num \to (num)CCS$:

$$\forall n.\ \mathsf{Counter}\ n = \mathsf{conp}\ Counter\ n$$
$$\forall n.\ \mathsf{Impl}\ n = \mathsf{conp}\ Impl\ n$$

The expressions involved in the above definitions are all agents. Their defining equations are given as a system of mutually recursive equations, which is represented in HOL by the following function CDfr:

$$\begin{array}{l}
\mathsf{CDfr} = \\
(\lambda\, s\ n. \\
\quad ((s = Counter) \Rightarrow \\
\quad ((n = 0) \Rightarrow \\
\quad up.(\mathsf{Counter}\ 1) + around.(\mathsf{Counter}\ 0)\ | \\
\quad up.(\mathsf{Counter}\ (n+1)) + down.(\mathsf{Counter}\ (n-1))\ | \\
\quad ((s = C) \Rightarrow up.(\mathsf{C\ Link\ C}) + down.\mathsf{D}\ | \\
\quad ((s = D) \Rightarrow \overline{d}.\mathsf{C} + \overline{a}.\mathsf{B}\ | \\
\quad ((s = B) \Rightarrow up.(\mathsf{C\ Link\ B}) + around.\mathsf{B}\ | \\
\quad ((s = Impl) \Rightarrow ((n = 0) \Rightarrow \mathsf{B}\ |\ \mathsf{C\ Link\ (Impl}\ (n-1)))\,|\,\mathsf{ARBag})))))
\end{array}$$

CDfr is proved to be a defining equations function using PROVE_IS_DEF_FUN. Let CDf be defined as the abstraction of CDfr, namely CDf = Def_Fun CDfr. Using the conversions CON_EVAL_CONV and CONP_EVAL_CONV, the agent associated to given constant name and parameter in the environment defined by CDf can be computed. The resulting theorems together with the unfolding law for the behavioural equivalences are used to derive behavioural theorems that assert the equivalence between a constant and the associated agent. Such theorems are needed for rewriting agents modulo behavioural equivalences. For observation equivalence and constants C, D and B, these theorems are:

$$\text{C\_OE\_THM:} \quad \vdash \mathsf{Obs\_Equiv\ CDf\ C}\ (up.(\mathsf{C\ Link\ C}) + down.\mathsf{D})$$
$$\text{D\_OE\_THM:} \quad \vdash \mathsf{Obs\_Equiv\ CDf\ D}\ (\overline{d}.\mathsf{C} + \overline{a}.\mathsf{B})$$
$$\text{B\_OE\_THM:} \quad \vdash \mathsf{Obs\_Equiv\ CDf\ B}\ (up.(\mathsf{C\ Link\ B}) + around.\mathsf{B})$$

[5]Note that the condition on whether the labels in the restriction set are not in the sort of the given agents is not formalised. For the moment it is implicitly assumed to be true. It will be possible to express it explicitly as soon as the notion of sort is mechanised in HOL.

The corresponding theorems for observation congruence are referred to as C_OC_THM, D_OC_THM and B_OC_THM. Some observation equivalences/congruences involving the linking operator are needed for proving the inductive case of the main proof. In HOL they are the following theorems:

OBS_EQUIV_D_C: $\vdash$ Obs_Equiv CDf (D Link C) (C Link D)

OBS_EQUIV_D_B: $\vdash$ Obs_Equiv CDf (D Link B) (B Link B)

OBS_CONGR_B_B: $\vdash$ Obs_Congr CDf (B Link B) B

OBS_EQUIV_D_Impl: $\vdash \forall n.$ Obs_Equiv CDf (D Link (Impl n)) (Impl n)

The proofs of OBS_EQUIV_D_C and OBS_EQUIV_D_B are both an application of the lazy expansion strategy to the left-hand side of the behavioural goal plus the τ-laws for observation equivalence. The theorem OBS_CONGR_B_B is instead proved using the expansion theorem only. The proof of OBS_EQUIV_D_Impl is by induction on the parameter n and makes use of the associativity of the linking operator and the first three theorems above.

In order to verify that the implementation of the counter meets the specification, by unique fixpoint induction the family of agents denoted by the function Impl must be observation congruent to the one denoted by Counter, whenever Counter has been replaced by Impl. This is formalised in HOL by the following goal:

$$?\vdash \forall n.\ \text{Obs\_Congr CDf (Impl } n) \text{ ((CDf } \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})$$

By applying mathematical induction the two cases to be proved are:

$$?\vdash \text{Obs\_Congr CDf (Impl 0) ((CDf } \mathit{Counter}\ 0)\{\mathit{Impl}/\mathit{Counter}\}) \tag{3.1}$$

$$\{\ \text{Obs\_Congr CDf (Impl } n) \text{ ((CDf } \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ \} \tag{3.2}$$

$$?\vdash \text{Obs\_Congr CDf (Impl } (n+1)) \text{ ((CDf } \mathit{Counter}\ (n+1))\{\mathit{Impl}/\mathit{Counter}\})$$

Rewriting with the definitions and the behavioural theorems of the agents involved is sufficient to solve the basis case (3.1). First, the definitions of Counter and Impl are used, thus obtaining the goal:

$$?\vdash \text{Obs\_Congr CDf (Impl 0) } (\mathit{up}.\text{(Impl 1)} + \mathit{around}.\text{(Impl 0)})$$

The agent constant Impl is then rewritten up to observation congruence for $n=1$ and $n=0$:

$$?\vdash \text{Obs\_Congr CDf B } (\mathit{up}.\text{(C Link B)} + \mathit{around}.\text{B})$$

The theorem B_OC_THM solves this subgoal. The proof of the inductive case (3.2) begins in a way similar to the basis case by rewriting with the definitions of the agents

and functions involved, namely Impl, Counter, and Link:

$$\{\ \mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ \}$$
$$\vdash\ \mathsf{Obs\_Congr\ CDf}$$
$$((\mathsf{C}\ [u'/u, a'/a, d'/d]\ |\ (\mathsf{Impl}\ n)\ [u'/up, a'/around, d'/down])\backslash\{u', a', d'\})$$
$$(up.\ (\mathsf{Impl}\ (n+2)) + down.\ (\mathsf{Impl}\ n))$$

The inductive hypothesis also needs to be rewritten using the definition of Counter, so that it can be applied to the goal. Note that the agent associated to name *Counter* and parameter n in CDf depends on the value of n. The aim of the inductive step in a proof (by mathematical induction) of a given property P is to show that $P\ n \supset P(n+1)$ for all $n \in \mathbb{N}$. This means proving $P\ 0 \supset P\ 1$, $P\ 1 \supset P\ 2$, and so on. Given the inductive definition of the specification of the counter, the proof of $P\ 0 \supset P\ 1$ involves agents which are different from the ones occurring in the proof of $P\ n \supset P(n+1)$ for all $n > 0$. Therefore, by performing case analysis on the value of the parameter n, the proof of the inductive case splits into two subproofs:

$$\{\ \mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ; \quad n=0\ \} \tag{3.3}$$
$$\vdash\ \mathsf{Obs\_Congr\ CDf}$$
$$((\mathsf{C}\ [u'/u, a'/a, d'/d]\ |\ (\mathsf{Impl}\ n)\ [u'/up, a'/around, d'/down])\backslash\{u', a', d'\})$$
$$(up.\ (\mathsf{Impl}\ (n+2)) + down.\ (\mathsf{Impl}\ n))$$

and

$$\{\ \mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ; \quad n=n'+1\ \} \tag{3.4}$$
$$\vdash\ \mathsf{Obs\_Congr\ CDf}$$
$$((\mathsf{C}\ [u'/u, a'/a, d'/d]\ |\ (\mathsf{Impl}\ n)\ [u'/up, a'/around, d'/down])\backslash\{u', a', d'\})$$
$$(up.\ (\mathsf{Impl}\ (n+2)) + down.\ (\mathsf{Impl}\ n))$$

Subgoal (3.3) is rewritten using the new assumption $n=0$ and the definition of C:

$$\{\ \mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ; \quad n=0\ \}$$
$$\vdash\ \mathsf{Obs\_Congr\ CDf}$$
$$(((up.\ (\mathsf{C\ Link\ C}) + down.\ \mathsf{D})\ [u'/u, a'/a, d'/d]\ |$$
$$(\mathsf{Impl}\ 0)\ [u'/up, a'/around, d'/down])\backslash\{u', a', d'\})$$
$$(up.\ (\mathsf{Impl}\ 2) + down.\ (\mathsf{Impl}\ 0))$$

The inductive hypothesis is rewritten with the assumption $n=0$ and the definitions of Counter and Impl, added to the assumption list and then applied to the left-hand

side of the goal:

$$\{\ \mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ;$$
$$n=0\ ;$$
$$\mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ 0)\ (\mathit{up}.\ (\mathsf{Impl}\ 1)+\mathit{around}.\ (\mathsf{Impl}\ 0))\ \}$$
$$\vdash\ \mathsf{Obs\_Congr\ CDf}$$
$$(((\mathit{up}.\ (\mathsf{C\ Link\ C})+\mathit{down}.\ \mathsf{D})\ [u'/u, a'/a, d'/d]\ \mid$$
$$(\mathit{up}.\ (\mathsf{Impl}\ 1)+\mathit{around}.\ (\mathsf{Impl}\ 0))\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}])\backslash\{u', a', d'\})$$
$$(\mathit{up}.\ (\mathsf{Impl}\ 2)+\mathit{down}.\ (\mathsf{Impl}\ 0))$$

The expansion theorem is used to transform the left-hand subterm by applying the laws for relabelling, parallel composition and restriction:

$$\{\ \mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ;$$
$$n=0\ ;$$
$$\mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ 0)\ (\mathit{up}.\ (\mathsf{Impl}\ 1)+\mathit{around}.\ (\mathsf{Impl}\ 0))\ \}$$
$$\vdash\ \mathsf{Obs\_Congr\ CDf}$$
$$(\mathit{up}.$$
$$((\mathsf{C\ Link\ C})\ [u'/u, a'/a, d'/d]\ \mid$$
$$(u'.\ ((\mathsf{Impl}\ 1)\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}])+$$
$$a'.\ ((\mathsf{Impl}\ 0)\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}]))))\backslash\{u', a', d'\})+$$
$$\mathit{down}.$$
$$((\mathsf{D}\ [u'/u, a'/a, d'/d]\ \mid$$
$$(u'.\ ((\mathsf{Impl}\ 1)\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}])+$$
$$a'.\ ((\mathsf{Impl}\ 0)\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}]))))\backslash\{u', a', d'\}))$$
$$(\mathit{up}.\ (\mathsf{Impl}\ 2)+\mathit{down}.\ (\mathsf{Impl}\ 0))$$

The above expansion shows that the actions of the second component in the parallel compositions are hidden by restriction, as they are renamed with labels occurring in the restriction set. Thus, this agent component does not contribute to the behaviour of the left-hand side of the goal and can be folded back by replacing it with the agent $(\mathsf{Impl}\ 0)\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}]$, because these two agents are observation congruent by the inductive hypothesis and the expansion theorem. The following goal is obtained:

$$\{\ \mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ;$$
$$n=0\ ;$$
$$\mathsf{Obs\_Congr\ CDf}\ (\mathsf{Impl}\ 0)\ (\mathit{up}.\ (\mathsf{Impl}\ 1)+\mathit{around}.\ (\mathsf{Impl}\ 0))\ \}$$
$$\vdash\ \mathsf{Obs\_Congr\ CDf}$$
$$(\mathit{up}.$$
$$(((\mathsf{C\ Link\ C})\ [u'/u, a'/a, d'/d]\ \mid$$
$$(\mathsf{Impl}\ 0)\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}])\backslash\{u', a', d'\})+$$
$$\mathit{down}.$$
$$((\mathsf{D}\ [u'/u, a'/a, d'/d]\ \mid$$
$$(\mathsf{Impl}\ 0)\ [u'/\mathit{up}, a'/\mathit{around}, d'/\mathit{down}])\backslash\{u', a', d'\}))$$
$$(\mathit{up}.\ (\mathsf{Impl}\ 2)+\mathit{down}.\ (\mathsf{Impl}\ 0))$$

The goal is then folded back using the definition of the linking operator:

$$\begin{array}{l}\{\ \mathsf{Obs\_Congr\ CDf\ (Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ;\\ \quad n=0\ ;\\ \quad \mathsf{Obs\_Congr\ CDf\ (Impl\ 0)}\ (\mathit{up}.\ \mathsf{(Impl\ 1)} + \mathit{around}.\ \mathsf{(Impl\ 0)})\ \}\\ \vdash\ \mathsf{Obs\_Congr\ CDf}\\ \quad (\mathit{up}.\ ((\mathsf{C\ Link\ C})\ \mathsf{Link\ (Impl\ 0)}) + \mathit{down}.\ \mathsf{(D\ Link\ (Impl\ 0))})\\ \quad (\mathit{up}.\ \mathsf{(Impl\ 2)} + \mathit{down}.\ \mathsf{(Impl\ 0)})\end{array}$$

The last steps of the proof apply the associativity of the linking operator and the previously proved theorem OBS_EQUIV_D_Impl by means of the theorem PROP6:

$$\begin{array}{l}\vdash \forall\, \mathit{Df}.\ \forall E\ E' :: \mathsf{Is\_Agent}.\\ \quad \mathsf{Obs\_Equiv}\ \mathit{Df}\ E\ E' \supset\ (\forall u.\ \mathsf{Obs\_Congr}\ \mathit{Df}\ (\mathsf{prefix}\ u\ E)\ (\mathsf{prefix}\ u\ E'))\end{array}$$

to strengthen observation equivalence in the presence of the prefix operator. The new goal is:

$$\begin{array}{l}\{\ \mathsf{Obs\_Congr\ CDf\ (Impl}\ n)\ ((\mathsf{CDf}\ \mathit{Counter}\ n)\{\mathit{Impl}/\mathit{Counter}\})\ ;\\ \quad n=0\ ;\\ \quad \mathsf{Obs\_Congr\ CDf\ (Impl\ 0)}\ (\mathit{up}.\ \mathsf{(Impl\ 1)}\ +\ \mathit{around}.\ \mathsf{(Impl\ 0)})\ \}\\ \vdash\ \mathsf{Obs\_Congr\ CDf}\\ \quad (\mathit{up}.\ \mathsf{(C\ Link\ (C\ Link\ (Impl\ 0)))}\ +\ \mathit{down}.\ \mathsf{(Impl\ 0)})\\ \quad (\mathit{up}.\ \mathsf{(Impl\ 2)}\ +\ \mathit{down}.\ \mathsf{(Impl\ 0)})\end{array}$$

This goal is simply solved using the definition of Impl. The remaining subgoal (3.4) is proved with steps similar to the ones for subgoal (3.3). The correctness of $Impl(n)$ with respect to $Counter_n$ modulo observation congruence is thus formally verified.

3.2.2 Checking Modal Properties

The parameterised specification of the infinite counter is now checked to possess a certain modal property. This property, given in [118], can be informally stated as "whatever goes up may come down in equal proportions". The (parameterised) formula that expresses such a property in Hennessy-Milner logic is the following. Let $[\![up]\!]^m\,\Phi$ be the formula Φ when $m=0$ and $[\![up]\!]\,[\![up]\!]^{m-1}\Phi$ when $m>0$, and similarly for $\langle\!\langle down\rangle\!\rangle^m$. The aim is to show that the following relation holds for all $m,\ n \in \mathbb{N}$:

$$Counter_n \ \models\ [\![up]\!]^m \langle\!\langle down\rangle\!\rangle^m\, \mathbf{tt} \tag{3.5}$$

This result is proved by induction on the parameter m. The proof is sketched below in the way it has been formalised and carried out in HOL.

The specification of the counter has been encoded in HOL in Section 3.2.1. In order to define the parameterised formula $\Phi = [\![up]\!]^m \langle\!\langle down\rangle\!\rangle^m\, \mathbf{tt}$, the m-times application

of a modal operator $f : (action)set \rightarrow eHML \rightarrow eHML$ to a set of actions A and a modal formula Fm is defined as the primitive recursive function Raise:

$$(\forall f\ A\ Fm.\ \mathsf{Raise}\ f\ A\ 0\ Fm\ =\ Fm) \wedge$$
$$(\forall f\ A\ m\ Fm.\ \mathsf{Raise}\ f\ A\ (m+1)\ Fm\ =\ f\ A\ (\mathsf{Raise}\ f\ A\ m\ Fm))$$

The intuition at the base of the proof of the relation (3.5) is that the proof of the inductive case can be done in a simple way if it is reduced to showing (as a separate result) that for all m, the counter in any state n satisfies the above formula Φ for $m+1$ if and only if it satisfies the same formula for m. This allows us to reduce the subgoal for the inductive case to the inductive hypothesis and prove (3.5) easily. Actually, a stronger result will be proved as it holds for *all* formulas Fm and not only for $\mathbf{tt}$, which is the particular formula in the property Φ. The key result to be proved is thus the following:

$$\forall m\ n\ Fm.$$
$$Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm \quad iff \quad Counter_n \models [up]^{m}\langle down\rangle^{m} Fm$$

This relation, referred to as Key_Lemma, is itself proved by induction on m. Its proof makes use of the following HOL theorems:

Box_Dmd_THM:

$$\vdash \forall n\ Fm.$$
$$\mathsf{Sat\ CDf}\ (\mathsf{Counter}\ n)\ (\mathsf{box}\ \{up\}\ (\mathsf{dmd}\ \{down\}\ Fm)) = \mathsf{Sat\ CDf}\ (\mathsf{Counter}\ n)\ Fm$$

Box_THM:

$$\vdash \forall n\ Fm.\ \mathsf{Sat\ CDf}\ (\mathsf{Counter}\ n)\ (\mathsf{box}\ \{up\}\ Fm) = \mathsf{Sat\ CDf}\ (\mathsf{Counter}\ (n+1))\ Fm$$

These theorems establish relations between the specification of the counter and some modal formulas. In particular, Box_Dmd_THM considers the counter in any state n and a formula modalised with the box and diamond operators. Box_THM deals with a formula containing the box operator only and shows that $Counter_n \models [up]Fm$ for all n and Fm, if and only if $Counter_{n+1} \models Fm$. Both theorems are proved by case analysis on the parameter n and then applying a tactic consisting of rewriting with the definition of the counter, the satisfaction relation Sat and the properties for the transition relation Trans, so that a contradiction is derived (in a way similar to the proof technique used in Section 2.4.2). A parametric tactic is defined in HOL which is then invoked in the proofs of the above theorems with appropriate arguments. The only difference is that Box_Dmd_THM also involves the diamond operator, so in its proof it is necessary to rewrite with the theorem

Dmd_THM:

$$\vdash \forall n\ Fm.\ \mathsf{Sat\ CDf}\ (\mathsf{Counter}\ (n+1))\ (\mathsf{dmd}\ \{down\}\ Fm) = \mathsf{Sat\ CDf}\ (\mathsf{Counter}\ n)\ Fm$$

stating that for all $n \in \mathbb{N}$ and formula Fm, $Counter_{n+1} \models \langle down\rangle Fm$ if and only if $Counter_n \models Fm$.

Moreover, when proving the inductive case of Key_Lemma, some manipulations of the modal formula are needed to transform it into a suitable form to which other rewritings can be applied. A transformation is given by the theorem Raise_Perm expressing the equality $f^{m+1} = f(f^m) = f^m(f)$ for all $m \in \mathbb{N}$ and modal operator f:

$$\text{Raise\_Perm: } \vdash \forall m\, f\, A\, Fm.\ \mathsf{Raise}\ f\ A\ (m+1)\ Fm\ =\ \mathsf{Raise}\ f\ A\ m\ (f\ A\ Fm)$$

The HOL proof of Key_Lemma is sketched below, where the usual CCS notation for agents and satisfaction relation is adopted. Note that all the reasoning holds in the environment defined by CDf, even though this is not made explicit in this informal presentation of the proof. Given the goal,

$$\begin{array}{l} ?\ \forall m\, n\, Fm. \\ \quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm) = \\ \quad (Counter_n \models [up]^{m}\langle down\rangle^{m} Fm) \end{array}$$

applying mathematical induction on the variable m produces the two subgoals:

$$\begin{array}{l} ?\ \forall n\, Fm. \\ \quad (Counter_n \models [up]^{1}\langle down\rangle^{1} Fm) = \\ \quad (Counter_n \models [up]^{0}\langle down\rangle^{0} Fm) \end{array} \tag{3.6}$$

$$\begin{array}{l} \{\ \forall n\, Fm. \\ \quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm) = \\ \quad (Counter_n \models [up]^{m}\langle down\rangle^{m} Fm)\ \} \\ ?\ \forall n\, Fm. \\ \quad (Counter_n \models [up]^{m+2}\langle down\rangle^{m+2} Fm) = \\ \quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm) \end{array} \tag{3.7}$$

Rewriting with the definition of Raise and the theorem Box_Dmd_THM solves the basis case (3.6). As far as the inductive case (3.7) is concerned, given any number n and formula Fm, the left-hand side of the induction subgoal is rewritten using the definition of Raise and the theorem Box_THM, thus getting the new subgoal:

$$\begin{array}{l} \{\ \forall n\, Fm. \\ \quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm) = \\ \quad (Counter_n \models [up]^{m}\langle down\rangle^{m} Fm)\ \} \\ ?\ (Counter_{n+1} \models [up]^{m+1}\langle down\rangle^{m+2} Fm) = \\ \quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm) \end{array}$$

The theorem Raise_Perm is then applied to the $(m+2)$-times application of the dia-

mond operator in left-hand side of the goal:

$$
\begin{array}{l}
\{\ \forall n\ Fm. \\
\quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm) = \\
\quad (Counter_n \models [up]^{m}\langle down\rangle^{m} Fm)\ \} \\
?\ (Counter_{n+1} \models [up]^{m+1}\langle down\rangle^{m+1}\langle down\rangle Fm) = \\
\quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm)
\end{array}
$$

The inductive hypothesis can now be applied on the left-hand side:

$$
\begin{array}{l}
\{\ \forall n\ Fm. \\
\quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm) = \\
\quad (Counter_n \models [up]^{m}\langle down\rangle^{m} Fm)\ \} \\
?\ (Counter_{n+1} \models [up]^{m}\langle down\rangle^{m}\langle down\rangle Fm) = \\
\quad (Counter_n \models [up]^{m+1}\langle down\rangle^{m+1} Fm)
\end{array}
$$

The induction subgoal (3.7) is finally solved by rewriting with the symmetric forms of the theorems Raise_Perm and Box_THM and the definition of Raise. As mentioned before, Key_Lemma is a stronger relation than the one actually needed for the proof of the relation (3.5) in two ways: it holds for every formula Fm and not only for **tt** and, once a given number m has been fixed, it is true for any state n of the counter.

Finally, the main property (3.5) can be checked in HOL by induction on m. The basis case is solved by rewriting with the definitions of the Raise operator and the satisfaction relation for the true formula. The induction step is proved by applying Key_Lemma and then rewriting with the inductive hypothesis. The detailed ML code for the proofs of Key_Lemma and the main property (3.5) are reported in Appendix C.2.

3.3 Extending the Syntax for Pure CCS

In Chapter 4 the value-passing calculus will be formalised in HOL based on Milner's translation into pure CCS. In order to be able to translate value-passing expressions into pure ones, the present mechanisation of the syntax for pure CCS needs to be extended to cope with the translation of value-passing actions and input prefixed agent expressions. This will lead to polymorphic pure actions and to the use of function types in recursive type definitions for representing (possibly infinite) indexed summations of agent expressions. This section presents the revised version of the HOL theory for pure CCS which will be the base for the formalisation of value-passing CCS. Milner's translation will be recalled and defined in HOL in Section 4.2. Here, it is enough to anticipate that an input prefixed expression $a(x).E$ is translated into an indexed summation of prefixed expressions. Such a summation ranges over the value

domain V under consideration, which may be infinite. Moreover, an input action $a(v)$, where a is an input port and v is the value being received, is translated into a pure action a_v for each $v \in V$.

3.3.1 Polymorphic Actions

Independently of the HOL formalisation of value-passing actions (which will be given in Section 4.1), the pure action a_v resulting from the translation of a value-passing action $a(v)$, would be an action label (name s) for some string s which is the result of a "combination" of the port a and the value v. This combination can be easily obtained if a polymorphic version of the pure actions is defined, in which ports can be of any type α. The polymorphic versions $(\alpha)label$ and $(\alpha)action$ for the syntactic types *label* and *action* are thus the following:

$$label = \mathsf{name}\ \alpha \mid \mathsf{coname}\ \alpha$$

$$action = \mathsf{tau} \mid \mathsf{label}\ (\alpha)label$$

The type variable α is appropriately instantiated depending on the particular application. For example, label (coname 7) is an action of type $(num)action$ and, if *bye* is a constant of type *string*, then the action label (name *bye*) is of type $(string)action$.

The operations of complement and relabelling of labels are defined on the polymorphic type $(\alpha)label$, and then extended to the type $(\alpha)action$, similarly to their definitions for the string-based version of pure CCS (Section 2.2.1).

When moving from the formalisation of pure CCS in which ports are strings, to the more general one in which ports can be of any type α, no relevant changes occur in the HOL code. All the proofs of properties and laws for the CCS operators go through unchanged, apart from adding some type information. The only interesting (and expected) change occurs in the definition of the conversions and tactics for applying the laws for the restriction, relabelling and parallel operators. In order to apply these laws, it is necessary to decide whether two labels are equal or not, or whether a label is in a given set of labels. In the polymorphic formalisation, these conversions and tactics have a procedure as a parameter to decide equality over the type which instantiates the type variable α. In the string-based formalisation such a decision procedure was defined in a straightforward way using the built-in HOL conversion `string_EQ_CONV` for equality of strings.

3.3.2 An Operator of Indexed Summation

The agent summation into which an agent expression $a(x).E$ is translated may be infinite depending on the value domain V. In Section 2.2.1 the syntax for pure CCS with the inactive agent and the binary summation operator has been mechanised in HOL. In what follows, the formalisation of an operator of indexed summation within a type for recursively defined agent expressions is presented.

Given the pure syntax in Section 3.1.2, the introduction of an operator of indexed summation leads to the following new version:

$$E ::= X \mid A(e_1, \ldots, e_n) \mid u.E \mid \sum_{i \in I} E_i \mid E \mid E \mid E \backslash L \mid E[f]$$

The meaning of the new operator is as follows. The agent expression $\sum_{i \in I} E_i$ is the (possibly infinite) summation of all expressions E_i as i ranges over the indexing set I. This agent expression may behave like any of its summands E_i. The inference rule for the transitions of a summation agent expression is:

$$\text{SUMM:} \quad \frac{E_j \xrightarrow{u} E'_j}{\sum_{i \in I} E_i \xrightarrow{u} E'_j} \qquad j \in I$$

Special cases of the summation operator are the inactive agent **nil** (when I is the empty set) and the binary summation $E_1 + E_2$ (when $I = \{1, 2\}$).

The summation operator $\sum_{i \in I} E_i$ can be formalised in HOL by means of a function f, defined on a set I of indexes, which returns an expression E_i for each $i \in I$. The specification for the type $(\gamma, \beta, \alpha)CCS$ of pure CCS expressions, where the type variables γ, β and α represent the types of indexes, parameters and ports respectively, is the following:

$$\begin{aligned} CCS = {} & \mathsf{var}\ string \mid \\ & \mathsf{conp}\ string\ \beta \mid \\ & \mathsf{prefix}\ (\alpha)action\ CCS \mid \\ & \mathsf{summ}\ (\gamma \to CCS)\ (\gamma)set \mid \\ & \mathsf{par}\ CCS\ CCS \mid \\ & \mathsf{restr}\ CCS\ ((\alpha)label)set \mid \\ & \mathsf{relab}\ CCS\ (\alpha)relabelling \end{aligned}$$

Using the extended version of the recursive type definition package (Section A.4), the theorem of higher order logic for the type $(\gamma, \beta, \alpha)CCS$ is derived:

$$\begin{aligned} \vdash\ & \forall f_1\ f_2\ f_3\ f_4\ f_5\ f_6\ f_7. \\ & \exists!\ fn. \\ & (\forall s1.\ fn(\mathsf{var}\ s1) = f_1\ s1) \wedge \\ & (\forall s1\ x2.\ fn(\mathsf{conp}\ s1\ x2) = f_2\ s1\ x2) \wedge \\ & (\forall A1\ C2.\ fn(\mathsf{prefix}\ A1\ C2) = f_3\ (fn\ C2)\ A1\ C2) \wedge \\ & (\forall f1\ s2.\ fn(\mathsf{summ}\ f1\ s2) = f_4\ (fn \circ f1)\ f1\ s2) \wedge \\ & (\forall C1\ C2.\ fn(\mathsf{par}\ C1\ C2) = f_5\ (fn\ C1)\ (fn\ C2)\ C1\ C2) \wedge \\ & (\forall C1\ s2.\ fn(\mathsf{restr}\ C1\ s2) = f_6\ (fn\ C1)\ C1\ s2) \wedge \\ & (\forall C1\ R2.\ fn(\mathsf{relab}\ C1\ R2) = f_7\ (fn\ C1)\ C1\ R2) \end{aligned}$$

where $\circ$ denotes composition of functions. Given an agent expression $\mathsf{summ}\ f1\ s2$, the clause for the indexed summation operator asserts that the unique function fn, which satisfies the recursive definition, is recursively applied to the agents defined through the function $f1$ and the indexing set $s2$.

The inactive agent and the binary summation operator can then be defined as instantiations of indexed summation. The inactive agent is formalised in HOL as a summation over an empty indexing set:

$$\mathsf{nil} = \mathsf{summ}\ \mathsf{ARB}\ \{\}$$

where ARB is instantiated to an arbitrary function from the indexing domain γ to $(\gamma, \beta, \alpha)CCS$. The operator '+' is defined as a summation over an indexing set of two distinct elements as follows:

$$\begin{array}{l}\forall E\ E' : (\gamma, \beta, \alpha)CCS. \\ \quad \mathsf{sum}\ E\ E' \ = \\ \quad \mathtt{let}\ x = \varepsilon x : \gamma.\mathsf{T}\ \mathtt{in} \\ \quad \mathtt{let}\ y = \varepsilon y : \gamma.\neg(y = x)\ \mathtt{in} \\ \quad \mathsf{summ}\ (\lambda i.\ (i = x) \Rightarrow E \mid ((i = y) \Rightarrow E' \mid \mathsf{nil}))\ \{x, y\}\end{array}$$

where $\varepsilon x : \gamma.\mathsf{T}$ denotes some element, say x, of type γ and $\varepsilon y : \gamma.\neg(y = x)$ denotes an element, say y, of type γ which is distinct from x.

The constructor con for parameterless agent constants is defined similarly to its definition in Section 3.1.2. Note that the formalisation of parameterised and parameterless agent constants through conp, con and defining equations, plus related functions and conversions, remains unchanged when going from the agent type $(\beta)CCS$ to $(\gamma, \beta, \alpha)CCS$.

The derived operators are easily proved to be distinct and one-to-one starting from their definitions and the corresponding properties for the basic operators. For example, the following theorem asserts that the operator sum is one-to-one:

sum_One_One:
$\exists\, a\, b : \gamma.\ \neg(a = b)$
$\vdash\ \forall E\ E'\ E''\ E'''.\ (\mathsf{sum}\ E\ E' = \mathsf{sum}\ E''\ E''') = (E = E'') \wedge (E' = E''')$

Note that binary summation is defined under the assumption that there exist two distinct elements in the indexing domain. From now on, every proposition involving the binary summation operator will have such an assumption. As soon as the type variable γ is instantiated to an indexing type that contains (at least) two distinct elements, this assumption can be removed by discharging it wherever it appears. Unless specified otherwise, the assumption $\exists\, a\, b : \gamma.\ \neg(a = b)$ will be denoted by a dot '.' before the symbol '$\vdash$'.

Note also that the operator of indexed summation does not introduce any difficulties when defining the transition relation and proving the algebraic laws of the various behavioural equivalences. The following theorem gives the operational semantics of the indexed summation operator by asserting that the transitions of a summation agent expression are all (and only) the transitions that any of the expressions in the

summation can perform:

TRANS_SUMM_EQ:
$\vdash \forall Df\ f\ I\ u\ E.\ \mathsf{Trans}\ Df\ (\mathsf{summ}\ f\ I)\ u\ E = (\exists i.\ i \in I\ \wedge\ \mathsf{Trans}\ Df\ (f\ i)\ u\ E)$

Properties about the transitions of any indexed summation can be derived from this theorem, e.g. that the inactive agent has no transitions. Behavioural congruences are preserved by the indexed summation operator. In the case of observation equivalence, this holds if for every index in the indexing set the corresponding summand is stable:

OBS_EQUIV_PRESD_BY_SUMM:
$\vdash \forall Df\ f\ f'\ I.$
$\quad (\forall i.\ i \in I\ \supset$
$\quad\quad \mathsf{Is\_Agent}\ (f\ i)\ \wedge\ \mathsf{Is\_Agent}\ (f'\ i)\ \wedge$
$\quad\quad \mathsf{Obs\_Equiv}\ Df\ (f\ i)\ (f'\ i)\ \wedge\ \mathsf{Stable}\ Df\ (f\ i)\ \wedge\ \mathsf{Stable}\ Df\ (f'\ i))\ \supset$
$\quad \mathsf{Obs\_Equiv}\ Df\ (\mathsf{summ}\ f\ I)\ (\mathsf{summ}\ f'\ I)$

Properties for the inactive agent and binary summation can be derived as particular instances of the ones for indexed summation.

3.4 Another Mechanisation of the Expansion Law

The introduction of the indexed summation operator in the HOL formalisation of pure CCS has no relevant effects on the proofs of the properties and algebraic laws for the other operators. They are unchanged, apart from updating some type information as the agent type has more type variables. However, given the presence of the indexed summation operator, one might want to reconsider the mechanisation of the expansion law (A11) in Section 2.1.3. The HOL formalisation of this law described in Section 2.2.4 can still be adopted, but it is obvious that a finite indexed summation defined through the function SIGMA is just a special case of an indexed summation defined through the constructor summ and applied to some function f over a finite indexing set I.

Let us recall that, given a function $f : num \rightarrow (\gamma, \beta, \alpha)CCS$ and an index $n : num$, the term SIGMA $f\ n$ encodes the summation $((f\ 0\ +\ f\ 1)\ +\ \ldots)\ +\ f\ n$ of agent expressions, where '+' is the binary summation operator. The constructor summ has two parameters, a function $f : \gamma \rightarrow (\gamma, \beta, \alpha)CCS$ and an indexing set $I : (\gamma)set$ which represents the domain of f. Thus, in the formulation of the expansion law based on SIGMA, the type variable γ is instantiated to the type num and the indexing set I is given by $\{0, \ldots, n\}$. The law (A11) can be mechanised using indexed summations ranging over natural numbers and it is reasonable to assume that, whenever one wants to write CCS expressions containing indexed summations, natural numbers are chosen as the indexing domain. Hence, when formalising the expansion law based

on summ, the type variable γ is instantiated to the type *num* and the expansion law is derived for agents of type $(num, \beta, \alpha)CCS$. By replacing SIGMA with summ, the expansion law PAR_LAW for observation congruence is the following HOL theorem:

$\vdash \forall Df\ f\ n\ f'\ m.$
  let $I1 = \{i : i \leq n\}$ and $I2 = \{j : j \leq m\}$ in
  $((\forall i.\ i \in I1 \supset \mathsf{Is\_Agent}\ (f\ i)\ \wedge\ \mathsf{Is\_Prefix}\ (f\ i))\ \wedge$
  $(\forall j.\ j \in I2 \supset \mathsf{Is\_Agent}\ (f'\ j)\ \wedge\ \mathsf{Is\_Prefix}\ (f'\ j))\ \supset$
  Obs_Congr Df
  (par (summ $f\ I1$) (summ $f'\ I2$))
  (sum
   (sum
    (summ ($\lambda i.$ prefix (PREF_ACT ($f\ i$)) (par (PREF_PROC ($f\ i$)) (summ $f'\ I2$))) $I1$)
    (summ ($\lambda j.$ prefix (PREF_ACT ($f'\ j$)) (par (summ $f\ I1$) (PREF_PROC ($f'\ j$)))) $I2$))
   (ALL_SYNC $f\ n\ f'\ m$)))

Note that the assumption about the existence of (at least) two distinct elements in the indexing domain has been discharged, as it is trivially true for the domain $\mathbb{N}$. The summations in parallel are given through two functions f, f' over the indexing sets $I1, I2$ respectively. The `let`-construct defines $I1$ and $I2$ so that they can be referred to in the theorem instead of writing down the actual sets. The predicates Is_Agent and Is_Prefix and the functions PREF_ACT, PREF_PROC and ALL_SYNC are the polymorphic versions of the ones in Section 2.2.4. Apart from replacing SIGMA $f\ n$ with summ $f\ \{0, \ldots, n\}$ and using the theorem TRANS_SUMM_EQ (Section 3.3.2) rather than SIGMA_TRANS_THM_EQ (Section 2.2.4), the new expansion law is exactly the same as the previous one. Moreover, the following theorem states that any indexed summation over a finite indexing set is observation congruent to the recursive application of binary summation through SIGMA:

SUMM_SIGMA: $\vdash \forall n\ f.$
  $(\forall i.\ i \leq n\ \supset\ \mathsf{Is\_Agent}\ (f\ i))\ \supset$
  $(\forall\ Df.\ \mathsf{Obs\_Congr}\ Df\ (\mathsf{summ}\ f\ \{i : i \leq n\})\ (\mathsf{SIGMA}\ f\ n))$

This theorem allows one to define the conversions for the application of the new formalisation of the expansion law by *reusing* the ones for the previous version. In fact, if two agents in parallel are both given through the constructor summ, then there is no transformation to be performed and the above theorem PAR_LAW can be applied directly. However, the binary summation operator (and possibly parentheses) will be typically used in applications instead of indexed summation. In this case, it is necessary to convert the parallel agents into indexed summations of the form summ $f\ \{0, \ldots, n\}$ such that the new expansion law can be applied. This is achieved through the theorem SUMM_SIGMA.

Nevertheless, one might not like to make the assumption that γ is the type *num* and prefer to keep it as a type variable. Actually, this is what is needed when the

value-passing version of the CCS calculus is translated into the pure one (Section 4.2). In fact, the translation function will instantiate the type variable γ for the indexing domain of the pure agent expressions to the disjoint sum of the indexing and value domains of the value-passing expressions. A simple solution to the problem of keeping γ as a type variable is to go back to the first formalisation of the expansion law through the function SIGMA. Contrary to the above mechanisation based on the constructor summ, one does not care about defining the expansion law in terms of the basic operator summ, but defines the function SIGMA on top of the CCS syntax for representing finite summations built from the repeated application of the binary summation operator. In this way, the expansion law can be defined in general over agents of type $(\gamma, \beta, \alpha)CCS$, where γ is not instantiated to any particular type. From the user's point of view, this means that one is not constrained to the type *num* for the type variable γ when defining his/her own agents, which is instead the case in the above formalisation of the expansion law based on summ. This is particularly convenient in the common situation in which there are no indexed summations in the user's applications: the user needs not know that a specific type must be provided for the indexing domain and can leave the type for such a domain unspecified. Thus, even though instantiating γ with the type *num* is reasonable, the solution based on SIGMA seems to be preferable.

In spite of this, one might be interested in investigating the HOL mechanisation of the expansion law with the constructor summ *and* the type variable γ. This raises the issue of how to enumerate the elements of a set over any type. In the above formalisation based on the indexing domain $\mathbb{N}$, it is easy to enumerate the objects by starting from 0 and then proceeding using the other constructor of the type, i.e. the successor operator SUC. But what about enumerating the elements in a set over any type γ? Given an agent $E_1 + \ldots + E_k$ (with various arrangements of parentheses for binary summation possible), the operator summ must be provided with a function $f : \gamma \rightarrow (\gamma, \beta, \alpha)CCS$ and a set $I = \{e_1, \ldots, e_k\} : (\gamma)set$, such that, for example, $f\, e_1 = E_1, \ldots, f\, e_k = E_k$. This means that an *enumeration function* is needed: given any type γ' and a set $s : (\gamma')set$, Enum s returns some function $\varepsilon f : num \rightarrow \gamma'$ that enumerates the elements in the set s:

$$\forall s.\ \mathsf{Enum}\ s\ =\ (\varepsilon f.\ s = \{f\, n : n < \mathsf{Card}\ s\})$$

where Card is the cardinality function over sets (from the HOL theory `sets`).

A mechanisation of the parallel law for pure CCS based on enumerating elements of any type is not only interesting from the theoretical point of view, as the formalisations presented above give reasonable solutions to this problem, but is also motivated by the following issue. As it will be discussed more extensively in Chapter 4, one of the aims of embedding the translation from value-passing CCS to the pure calculus in HOL, is to be able to derive the properties and laws for the value-passing calculus from the ones for the pure subset without having to redo their proofs. The derivation

of the expansion law for value-passing agents raises the following problems. First, the translation of an input prefixed (value-passing) agent is a possibly infinite summation. The above formalisations of the expansion law for pure CCS deal with agents in parallel which are finite summations, and thus are unable to cope with the result of translating parallel components which contain input prefixed agents. Second, the expansion law (A11) assumes that the two expressions is parallel are summations of prefixed agents. These prefixed summands are all denoted by means of the same operator prefix in the HOL formalisation of pure CCS. As it will be shown in Section 4.1, the prefix operator in value-passing CCS cannot be mechanised through a single constructor. This is due to the fact that prefixing an agent expression E with an input action $a(x)$ is semantically different from prefixing it with an output action $\overline{a}(e)$, as a binding for the variable x is introduced in $a(x).E$. Thus, three different operators for input, output and τ-prefixed agent expressions will be used when mechanising the value-passing syntax in HOL. When translating a summation of agents, each of which can be either of these three kinds of agents, output and τ-prefixed expressions will be translated into prefixed pure expressions, while input prefixed expressions will be translated into possibly infinite pure summations. This means that the result of the translation is not in the expected syntactic form to apply the expansion law for pure agents, but some transformations must be carried out to get the desired structure of the parallel expressions.

The syntactic differences between the formalisations of pure and value-passing expressions, mainly concerning the prefix operator, can be abstracted from if a *semantic* formulation of the expansion law, based on the transition relation, is adopted. This formulation is the one given in [97] and is recalled below. Let $n \geq 1$, then

$$\begin{aligned} & P_1 \mid \ldots \mid P_n \approx_c \\ & \sum \{u.\,(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_n) : P_i \xrightarrow{u} P_i'\} + \\ & \sum \{\tau.\,(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_j' \mid \ldots \mid P_n) : P_i \xrightarrow{l} P_i',\ P_j \xrightarrow{\bar{l}} P_j',\ i<j\} \end{aligned} \tag{3.8}$$

In this law there is a bit of abuse of the CCS notation. In fact, the indexed summation operator is not used as defined in the syntax: no indexing set is provided to the summation operator, which is instead applied to a set of agents. The original definition of the indexed summation operator $\sum_{i \in I} E_i$ can also be written as $\sum \{E_i : i \in I\}$, thus resulting in a particular instance of the more general specification $\sum \{E[x] : Pred[x]\}$ used in the law (3.8). Moreover, in this notation there is no information about any indexing or enumeration of the agents in the given sets.[6] Once more, there is the

[6]It would be very convenient to have the possibility of defining a type

$$CCS = \ldots \mid \mathsf{summ}\ (CCS)set \mid \ldots$$

in HOL, but unfortunately this is not feasible because the cardinality of the powerset of the type *CCS* is strictly greater than that of the type *CCS* being defined. Thus, the above type specification

problem of how to identify or enumerate the elements in a (finite or infinite) set of any type for which no indexing is provided.

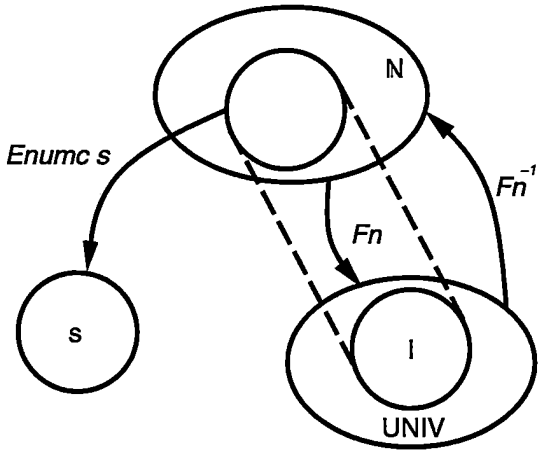

Figure 3.1: Constructing the arguments for summ.

Given a summation $\sum\{E[x]:Pred[x]\}$, let us discuss how such an agent expression can be formalised in terms of summ. Let Enumc s be an enumeration function for a (possibly infinite) set $s:(\gamma')set$ defined in HOL as follows:

$$\forall s:(\gamma')set.\ \mathsf{Enumc}\ s\ =\ (\varepsilon f{:}\,num\to\gamma'.\ s=\{x:\exists n.\ x=f\ n\})$$

Given a set $s:((\gamma,\beta,\alpha)CCS)set$ of pure agent expressions and an enumeration function for s, namely Enumc $s:num\to(\gamma,\beta,\alpha)CCS$, a function $Fn':\gamma\to num$ can be composed with Enumc s so that a function $f:\gamma\to(\gamma,\beta,\alpha)CCS$ is obtained. The function Fn' can be taken as the inverse of the one-to-one function $fn:num\to\gamma$ which exists if the universal set over the type γ is infinite, as stated by the following HOL theorem:

$$\text{fn\_num\_index:}\quad\vdash \mathsf{Infinite\ Univ}\ \supset\ (\exists fn.\ \mathsf{One\_One}\ fn)$$

where Univ $:(\gamma)set$ denotes the universal set over the type γ. This existing function is given a name, say Fn, by making a constant specification:

$$\text{Fn\_THM:}\quad\vdash \mathsf{Infinite\ Univ}\ \supset\ \mathsf{One\_One}\ Fn$$

This means that under the hypothesis that the indexing domain is infinite, for any set of CCS expressions $s:((\gamma,\beta,\alpha)CCS)set$, the function $f:\gamma\to(\gamma,\beta,\alpha)CCS$ defined as the composition

$$(\mathsf{Enumc}\ s)\ \circ\ (\mathsf{Inv}\ Fn)$$

is the function argument to the operator summ. The indexing set $I:(\gamma)set$ which represents the domain of such a function f is defined as the image of the function Fn

does not have a solution in HOL, as there is no one-to-one mapping of the powerset of the type into the type itself [55].

over the set of numbers which is the domain of the enumeration function. Thus, I is the set given by

$$\mathsf{Image}\ Fn\ \{n : (\mathsf{Enumc}\ s\ n) \in s\}$$

The situation is depicted by the drawing in Figure 3.1.

Let us now consider the summations over sets in the expansion law (3.8). If $n = 1$, there is no parallel composition and no synchronisation, thus the observation congruence reduces to:

$$P \ \approx_c \ \sum \{u.\, P' : P \xrightarrow{u} P'\}$$

which states that any agent P is observation congruent to the (possibly infinite) summation of all those agents obtained by prefixing each action u of P to the agent P' into which P evolves when doing u. This set will be referred to as the *transition set* of an agent expression E and is defined in HOL as follows:

$$\forall\, Df\ E.\ \mathsf{Trans\_Set}\ Df\ E = \{\mathsf{prefix}\ u\ E' : \mathsf{Trans}\ Df\ E\ u\ E'\}$$

The transition set can be either finite (finitely branching transition system) or infinite (infinitely branching transition system). An enumeration function for such a set will exist if the set is *countable*, namely its elements can be enumerated, according to the following definition:[7]

$$\forall s : (\gamma')set.\ \mathsf{Countable}\ s \ = \ (\exists f : num \to \gamma'.\ s = \{x : \exists n.\ x = f\ n\})$$

If a set s is countable, then the function $\mathsf{Enumc}\ s$ is just the enumeration function for s. This is stated by the following theorem:

$$\text{Enumc\_THM:} \quad \forall s : (\gamma')set.\ \mathsf{Countable}\ s \ \supset \ (s = \{x : \exists n.\ x = \mathsf{Enumc}\ s\ n\})$$

At this point it is possible to prove that any agent is behaviourally equivalent to its countably infinite branching tree (the finite case is just a special instance of the countable case). For observation congruence the theorem is as follows:

TRANS_SET_THM:
Infinite Univ
$\vdash\ \forall\, Df.\ \forall E ::$ Is_Agent.
  let $s =$ Trans_Set $Df\ E$ in
  (Countable $s\ \supset$
   Obs_Congr Df
   E
   (summ ((Enumc s) $\circ$ (Inv Fn)) (Image Fn $\{n : (\mathsf{Enumc}\ s\ n) \in s\}$)))

Let us now derive the expansion law (3.8) for $n = 2$. The functions Trans_Set_Par_L and Trans_Set_Par_R are first defined which compute the sets of agent expressions into

[7] A HOL formalisation of countable sets by Harrison can be found in [57].

which a parallel composition evolves as the result of the transitions of either of the two parallel components. Both sets are defined as the image of an appropriate function on the transition set of either component:

$\forall\, Df\ E\ E'.$
Trans_Set_Par_L $Df\ E\ E' =$
Image $(\lambda x.$ prefix (PREF_ACT x) (par (PREF_PROC x) E')) (Trans_Set $Df\ E$)

$\forall\, Df\ E\ E'.$
Trans_Set_Par_R $Df\ E\ E' =$
Image $(\lambda x.$ prefix (PREF_ACT x) (par E (PREF_PROC x))) (Trans_Set $Df\ E'$)

Then, the function Sync_Set encodes the set of agent expressions resulting from the synchronisation between the two parallel components:

$\forall\, Df\ E\ E'.$
Sync_Set $Df\ E\ E' =$
{prefix tau (par $E''\ E'''$) :
$\exists\, l.$ Trans $Df\ E$ (label l) $E''\ \wedge$ Trans $Df\ E'$ (label (Compl l)) E'''}

Various properties about such sets are proved in HOL which are then used in the proof of the expansion law. For example, under the assumption that the transition set of an agent expression is countable, it is easy to derive that the sets computed by Trans_Set_Par_L and Trans_Set_Par_R are countable. This is just a consequence of a more general theorem asserting that the image of a function on a countable set is countable:

IMAGE_COUNTABLE: $\forall s : (\gamma')set.$ Countable $s\ \supset\ (\forall f.$ Countable (Image $f\ s$))

The expansion law for observation congruence over agents with possibly infinite, yet countable, transition sets is finally derived:

COUNTABLE_PAR_LAW:
Infinite Univ
$\vdash \forall Df.\ \forall E\ E'$ :: Is_Agent.
let sl = Trans_Set_Par_L $Df\ E\ E'$
and sr = Trans_Set_Par_R $Df\ E\ E'$
and ss = Sync_Set $Df\ E\ E'$ in
(Countable (Trans_Set $Df\ E$) $\wedge$ Countable (Trans_Set $Df\ E'$) $\supset$
Obs_Congr Df
(par $E\ E'$)
(sum
(sum
(summ ((Enumc sl) $\circ$ (Inv Fn)) (Image Fn $\{n :$ (Enumc $sl\ n$) $\in sl\}$))
(summ ((Enumc sr) $\circ$ (Inv Fn)) (Image Fn $\{n :$ (Enumc $sr\ n$) $\in sr\}$)))
(summ ((Enumc ss) $\circ$ (Inv Fn)) (Image Fn $\{n :$ (Enumc $ss\ n$) $\in ss\}$)))))

Once more, note that the assumption about the existence of (at least) two distinct elements in the indexing domain (because binary summation occurs in the above term) is discharged, as it can trivially be derived from the infinity hypothesis for such a domain. Similarly to the other laws for observation congruence (Section 2.2.4), the above theorem is derived by some forward inference from the same result with respect to strong equivalence. In that proof, in order to apply the definition of strong equivalence, the parallel composition and the summation term must be proved to be agents. This is trivial for the parallel term '$\mathsf{par}\ E\ E'$' given that E and E' are agents, while a few proof steps are needed for the summation term. In this part of the proof, properties of the predicate Is_Agent and of the sets defined by Trans_Set_Par_L, Trans_Set_Par_R and Sync_Set are used together with the theorems Fn_THM and (the polymorphic version of) TRANS_Is_Agent (Section 2.2.2). The parallel law for strong equivalence is then proved in the usual manner, namely by providing the appropriate strong bisimulation which contains the two given agents.

3.5 Summary

In this chapter, agent constants and their defining equations have been introduced into the HOL formalisation of the pure calculus. They replace the *rec*-notation used in Chapter 2 and allow processes with infinite behaviour (and possibly mutually recursive) to be specified in a simple and convenient way. Agent constants may be parameterised, i.e. defined in terms of the values of some parameters. This is different from the presentation in [97], where pure agent constants are actually parameterless (nullary combinators) and parameterised constants $A(e_1, \ldots, e_n)$ are considered as parameterless constants $A_{e_1,\ldots,e_n}$, given an infinite set of agent constants $\mathcal{K}$. Thus, parameterised agents of type $(\beta)CCS$ are identified not only by their name (the constant) but also by their parameters. Moreover, the variable β for the parameters' type can be instantiated with any concrete (recursive) data type. This introduces the *data* ingredient in the pure CCS framework: data are not yet exchanged during communication, like in the value-passing setting, but the behaviour of pure agents can be defined based on data and this can lead to infinite state specifications.

Parameterised agents have also been formalised in HOL in [15]. There, process names are given through *function invocations* represented by a type variable *invoc*, and a function $invocval : invoc \rightarrow process$ mapping process names to processes, is a parameter to the transition relation, in a way similar to a defining equations function Df. Both parameterless and parameterised processes are considered by refining the type of process names to be the disjoint sum $invoc + (par \rightarrow invoc)$, where *par* is the type variable for the parameters' type.

The new polymorphic formalisation of pure CCS expressions has a few advantages. First, it gives the possibility of specifying pure (finite or infinite state) parameterised

agents, such as $Buffer_n$ and $Counter_n$, in a very convenient and readable form. Simple raw functions are used for defining constants. This allows one to add new constants and/or modify the agents associated to given constants very easily. Furthermore, conversions are provided for checking that the given function actually represents a defining equations function. Second, there is no difficulty in extending agent constants to constants with agent parameters (see Chapter 9 in [97]). The definition of the type $(\beta)CCS$ remains the same and the type variable β for parameters is instantiated to the agent type $(\beta)CCS$ itself (possibly in disjoint union with the types of other parameters). In this case, the combinator conp would take a string s and an agent P as its arguments, and return the agent constant with name s and parameterised over agent P. Third, value-passing parameterised agent constants and their defining equations will be translated into the pure calculus in a simple and natural way (Section 4.2).

The resulting HOL formalisation has been used to verify properties of parameterised specifications. The correctness of an implementation for the (finite state) specification $Buffer_n$ has been proved in [100, 102]. The correctness of a simple counter has been verified in Section 3.2 and Appendix C by showing how reasoning about infinite state systems can be carried out in the HOL-CCS environment.[8] In a theorem proving framework, inductively defined systems can be naturally analysed and verified by manipulating the process algebra specifications by means of powerful proof techniques, such as equational reasoning, various forms of induction (mathematical, structural, etc.), case analysis, etc. This kind of reasoning is instead more difficult to accommodate in an automata based framework, even in the case of finite state specifications.

The second part of this chapter has refined the CCS syntax even further by introducing the indexed summation operator $\sum_{i \in I} E_i$ and deriving the inactive agent and the binary summation in terms of the more general operator. This has led to a new polymorphic version of the pure calculus in HOL, namely the type $(\gamma, \beta, \alpha)CCS$, parameterised over the types of the indexing domain, the parameters of agent constants and the ports. Such a type has been derived using an extended version of the HOL type definition package, which can deal with *ty*-valued functions, where *ty* is the type being defined.

Following the approach in [15], the type for the CCS syntax with the indexed summation operator could be defined using the built-in HOL rule for recursive type definition without resorting to any extended version. This can be achieved by defining the CCS type in terms of "names" (i.e. the above mentioned function invocations) for the processes returned by CCS-valued functions. Thus, the operator summ would have the type $\gamma \rightarrow invoc$ and the CCS syntax would also be augmented with a constructor Call which takes a function invocation of type *invoc* and returns the process invoked

[8] An early version of the verification of the modal property for the counter specification can be found in [101].

by such a call. The semantics of the operator Call is just to activate the process associated to the name given by the function invocation.

Once indexed summations have been defined in HOL, different mechanisations of the expansion law for the parallel composition operator have been investigated. Under the assumptions that the indexing domain is infinite and agents have countable transition sets, it is possible to analyse their actions and verify their equivalences. This is another example of the kind of meta-theoretic reasoning that can be carried out in the HOL-CCS environment about the embedded calculus itself. A further example will be provided by the translation from the value-passing calculus to the pure one in the next chapter.

Chapter 4

Value-Passing CCS in HOL

A lot of research has been dedicated to the development of methods and tools for reasoning about pure process calculi. Indeed, most of the verification systems recalled in Section 1.1 work on agent specifications in which communication is simply synchronisation, namely no data is transmitted. In this chapter, the value-passing version of the CCS calculus presented in [97] is considered, and the proof environment developed for pure CCS in HOL is extended to dealing with value-passing specifications.

Value-passing CCS is a process calculus in which actions consist of sending and receiving values through communication ports, and the transmitted data can be tested using a conditional construct. The algebra of value-passing actions is thus richer than the pure one. In fact, a pure action is just a label or port s equipped with information about its status, namely whether the port is an input one (a name s) or an output one (a co-name $\overline{s}$). However, in a pure setting, there is no transmission of data, so directionality is not an issue and it is possible, for example, to rename a name s with a co-name $\overline{s'}$, $s \neq s'$. This is not true any more when value passing is considered. In value-passing actions the complement operation actually denotes directionality. Besides the silent action τ, value-passing expressions can be prefixed with prefixes of the form '$a(x).$' (input prefix) and '$\overline{a}(e).$' (output prefix). Thus, besides the input/output information, a port a is followed by a value expression which, in particular, is a value variable x for input prefixes.

The main difficulty in dealing with value-passing specifications lies in the semantics of the input prefix operator. Given a domain V of values, the meaning to an input prefixed expression $a(x).E$ is given by the rule

$$\text{INPUT:} \quad \frac{}{a(x).E \xrightarrow{a(v)} E\{v/x\}} \qquad v \in V$$

where $E\{v/x\}$ denotes the substitution of the value v for all free occurrences of x in the expression E. Such occurrences are bound in the expression $a(x).E$ by the prefix '$a(x).$'. The rule INPUT asserts that the agent $a(x).E$ can perform the action $a(v)$ of receiving any value $v \in V$ in the variable x through the input port a. Whenever the value domain V is infinite, the above rule gives rise to infinitely branching transition

systems (unbounded non-determinism).

Value-passing versions of the CCS process algebra have been introduced in [97, 59]. In [97] Milner presents a value-passing calculus by defining its signature and giving the semantics to the operators by *translation* into the pure version of the calculus. The transition relation for value-passing expressions, the behavioural semantics as well as their laws, can then be derived from those of the pure calculus through the translation. This approach relies on an operator of (possibly infinite) indexed summation, because the translation $a(\widehat{x}).E$ of an input prefixed agent is defined as the summation $\sum_{v \in V} a_v.E\{\widehat{v/x}\}$. It is generally thought that this indexed summation operator makes the mathematical details more complicated. However, the HOL formalisation of such an operator presented in Section 3.3.2 shows that it is possible to reason about (possibly infinite) indexed summations and prove properties about them without much difficulty.

Given the rule INPUT, the substitutivity rules for the input prefix operator are of the following form:

$$\frac{\forall v \in V \;\; E\{v/x\} \sim_{be} E'\{v/x\}}{a(x).E \sim_{be'} a(x).E'}$$

where the behavioural equivalences $\sim_{be}$ and $\sim_{be'}$ are such that $\sim_{be} = \sim_{be'}$ or $\sim_{be'}$ is stronger than $\sim_{be}$ (e.g. observation equivalence is strengthened into observation congruence). In [59] it is claimed that this kind of rule, called *ω-data-rule*, should be avoided if one wants a "realistic and useful" proof system. This is because this rule is an infinitary (non-recursively enumerable) rule, due to the possibly infinite number of its premisses, and thus makes any proof system including it ineffective, although sound and complete. Some ω-data-rules will be derived in HOL in Section 4.5 and it will be shown how to effectively use this kind of rule in the verification example in Section 4.6.

When presenting his version of value-passing CCS in [59], Hennessy does not resort to any translation and defines a denotational semantics for his calculus based on Acceptance Trees. This model allows him to represent any input prefixed expression as a function mapping values to semantic objects and to get a well-behaved algebraic complete partial order with the standard proof techniques. A proof system is then proposed which is sound and complete for finite terms with respect to the denotational semantics and does not contain ω-data-rules. Such a proof system provides powerful methods for reasoning about value-passing agents based on the idea of separating reasoning about the data from reasoning about the process behaviour as much as possible. The proof systems given in [60] follow the same approach, but there the denotational semantics is replaced by a *symbolic* operational semantics that allows many value-passing agents to be represented in terms of finite symbolic transition systems, although the standard transition systems are infinite. The verification tool VPAM (Section 1.1.2) is based on such proof systems.

The approach taken in this dissertation to formalising value-passing CCS in HOL

is Milner's one based on the translation into the pure calculus. This is motivated by the following considerations:

- The translation is an interesting piece of meta-theory which relates the pure and value-passing versions of a calculus for communicating systems. This meta-reasoning will allow us to compare the two methods for giving the semantics of the value-passing calculus, namely the one based on the translation and the direct one according to the SOS approach, thus showing the correctness of the translation (Section 4.3).

- Given the HOL formalisation for pure CCS, the translation from the value-passing syntax to the pure one will enable us to derive the properties and the laws for the various semantics of the value-passing calculus without having to redo all the proofs. These proofs can be simply carried out by translating the value-passing expressions into their pure versions, and then using the corresponding results already proved for pure CCS.

- Once the properties and behavioural laws have been derived, reasoning about value-passing specifications will be performed without translating them into their pure versions (which would contain possibly infinite indexed summations). The translation is used at meta-level for developing behavioural theories for the value-passing calculus, and users of the verification environment will only work on value-passing specifications.

- Not only is the translation worth mechanising, but it is also interesting from the point of view of its HOL formalisation. In fact, it introduces the use of polymorphism in the HOL type of CCS actions, and raises the problem of defining an operator of (possibly infinite) indexed summation over recursively defined agent expressions. These issues have already been discussed in Section 3.3.

The version of the value-passing calculus under consideration is the one presented in [97]. Differently from Chapter 2, where first *rec*-notation and then agent constants and defining equations were introduced, the syntax of the value-passing calculus is directly given with parameterised agent constants and defining equations. Furthermore, the indexed summation operator is included in its syntax.

In the following sections, the syntax for the value-passing calculus is embedded in HOL and the translation is then defined by exploiting some HOL infrastructure, such as λ-abstraction and β-reduction to represent variable binding and substitution of values for value variables. Then, the operational semantics for the value-passing calculus is recalled and formalised in HOL by defining the transition relation over value-passing expressions. Hence, the translation is shown to be correct with respect to the semantics of value-passing CCS by formally proving that for each value-passing transition there exists a corresponding one between the translations of the agents

and actions involved. Behavioural equivalences for value-passing agents can thus be defined based on the translation and the corresponding equivalences over pure CCS. Their properties and algebraic laws are derived by resorting to the corresponding properties and laws for pure CCS, and reasoning about value-passing agents can be performed by applying the derived theorems without translating them into their pure versions. To illustrate how the resulting proof environment can be used to analyse value-passing specifications, the observation congruence between two different descriptions of a communicating system is verified in HOL. This example system is taken from [59] and the HOL proof is finally compared with the one carried out in Hennessy's proof system.[1]

4.1 The Value-Passing Syntax in HOL

The syntax of Milner's value-passing calculus [97] is recalled in this section. The denotation and meaning of many of the symbols and operators are the same as in Sections 2.1.1, 3.1.2 and 3.3.2. Below, only the new symbols and operators are explained.

Given a domain V of values, let value constants, value variables and value expressions over V be denoted by v, x and e respectively. Let $\mathcal{X}$ be a set of agent variables ranged over by X and $\mathcal{K}$ be a set of agent constants ranged over by A. An arity $n \geq 0$ is assigned to each constant $A \in \mathcal{K}$. Let I and b denote indexing sets and boolean expressions respectively. The label a ranges over the set of ports at which data can be input. For any input port a there is a corresponding output port $\overline{a}$, from which data can be sent. The set of input and output labels (or ports) is ranged over by l, and L ranges over subsets of labels. The set of value-passing actions, ranged over by u, includes input actions $a(v)$, output actions $\overline{a}(e)$ and the silent action τ. Actions with no value parameters are special cases of value-passing actions.

The syntax of value-passing expressions, ranged over by E, is the following:[2]

$$\begin{aligned} E \ ::= \ & X \mid A(e_1, \ldots, e_n) \mid \textstyle\sum_{i \in I} E_i \mid a(x).E \mid \overline{a}(e).E \mid \tau.E \mid \\ & E \,|\, E \mid E \backslash L \mid E[f] \mid \textbf{if } b \textbf{ then } E \end{aligned}$$

As in pure CCS, agent variables give the possibility of writing open expressions. Each agent constant A with arity n has a defining equation $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} E$, where the expression E may contain no agent variables (thus, E is an agent) and no free value variables except $x_1, \ldots, x_n$.

The meaning of the new operators is as follows.

[1] Preliminary versions of the work described in this chapter are given in [103, 104].

[2] Note the overloading of symbols: the same symbols used for the operators, variables and constants over the pure calculus also denote the value-passing counterparts. They will be distinguished in the HOL formalisation.

- The agent expression $a(x).E$ can perform the action $a(v)$ of receiving any value $v \in V$ at the input port a, and then behaves like $E\{v/x\}$, i.e. the agent expression obtained by substituting the value v for all free occurrences of the variable x in the expression E.

- The agent expression $\overline{a}(e).E$ can output the value expression e by performing the action $\overline{a}(e)$ and then behaves like E.

- The agent expression $\tau.E$ can perform the action τ and then behaves like E.

- The conditional expression **if** b **then** E behaves like E if the boolean expression b is true, and cannot perform any action otherwise. The two-armed conditional expression **if** b **then** E **else** E' can be defined in terms of the binary summation, one-armed conditional and boolean negation as (**if** b **then** E)+(**if** $\neg b$ **then** E').

The operational semantics for value-passing agent expressions will be formally given and mechanised in HOL in Section 4.3. In what follows, the syntax for the value-passing calculus is embedded in HOL.

A label is simply the name of a port at which communication is performed, and a port can be either an input or an output port. Labels can be restricted and renamed via the restriction and relabelling operators respectively, while values (which appear in actions) are neither restricted nor renamed.

The polymorphic versions $(\alpha)labelv$ and $(\alpha, \beta)actionv$ of the types for value-passing labels and actions are defined similarly to the ones for pure labels and actions (Section 3.3.1):

$$labelv = \mathsf{in}\ \alpha \mid \mathsf{out}\ \alpha$$

$$actionv = \mathsf{tauv} \mid \mathsf{labelv}\ (\alpha)labelv\ \beta$$

where α and β are the type variables for ports and values respectively. For example, let a and *hello* be constants of type *string*. Then, the value-passing action labelv (in a) 5 is an input action of type $(string, num)actionv$, the labels of ports and the values under consideration being strings and natural numbers respectively, and labelv (out T) *hello* is an action of type $(bool, string)actionv$.

The complement operation Complv is defined on the polymorphic type $(\alpha)labelv$ similarly to its definition in the formalisation of pure CCS, and then extended to the type $(\alpha, \beta)actionv$ in a straightforward way by defining the new function Compl_Actv.

Because there is no data transmission in pure CCS, the direction of a communication is not relevant. Therefore, it is possible to change the status of a port by renaming a label which is a name into a co-name and vice versa. On the contrary, in value-passing CCS the status of a port is fixed and an input port cannot be renamed into an output port and vice versa. Thus, relabelling functions over value-passing expressions are simply defined as functions that rename ports, i.e. the relabelling type is the function type $\alpha \rightarrow \alpha$. The renaming of ports is then extended to value-passing

labels and actions by means of the function relab_labv : $(\alpha \rightarrow \alpha) \rightarrow (\alpha)labelv \rightarrow (\alpha)labelv$ defined as follows:

$$(\forall\, rf\ p.\ \mathsf{relab\_labv}\ rf\ (\mathsf{in}\ p)\ =\ \mathsf{in}\ (rf\ p)) \wedge$$
$$(\forall\, rf\ p.\ \mathsf{relab\_labv}\ rf\ (\mathsf{out}\ p)\ =\ \mathsf{out}\ (rf\ p))$$

and the function relabelv : $(\alpha \rightarrow \alpha) \rightarrow (\alpha, \beta)actionv \rightarrow (\alpha, \beta)actionv$:

$$(\forall\, rf.\ \mathsf{relabelv}\ rf\ \mathsf{tauv}\ =\ \mathsf{tauv}) \wedge$$
$$(\forall\, rf\ l\ x.\ \mathsf{relabelv}\ rf\ (\mathsf{labelv}\ l\ x)\ =\ \mathsf{labelv}\ (\mathsf{relab\_labv}\ rf\ l)\ x)$$

These functions respect the complement operation and τ is renamed as τ. Moreover, a constructor RELABv : $(\alpha \times \alpha)list \rightarrow (\alpha \rightarrow \alpha)$, namely the value-passing counterpart of RELAB (Section 2.2.1), allows one to write a relabelling function in the substitution-like notation. Given a list of pairs of ports, RELABv returns the value-passing relabelling defined through that list.

The syntax for value-passing agent expressions is defined in HOL similarly to that for pure CCS (Section 3.3.2). Besides the clause for indexed summation, a function type is also used for the mechanisation of the input prefix operator. In fact, an input prefixed expression $a(x).E$ is formalised in HOL by means of the operator In which takes as arguments a port a and a function f from values to agent expressions. This function will be given as a λ-abstraction $\lambda x.\, E[x]$ which binds the value variable x in the expression $E[x]$. In this way, the notion of λ-abstraction in HOL is used to formalise the variable binding in an input prefixed expression.

The specification for the type $(\gamma, \beta, \alpha)CCSv$, where the type variables γ, β and α represent the types of indexes, data and ports respectively, is thus the following:

$$\begin{aligned} CCSv = {} & \mathsf{Var}\ string\ | \\ & \mathsf{Conp}\ string\ \beta\ | \\ & \mathsf{Summ}\ (\gamma \rightarrow CCSv)\ (\gamma)set\ | \\ & \mathsf{In}\ \alpha\ (\beta \rightarrow CCSv)\ | \\ & \mathsf{Out}\ \alpha\ \beta\ CCSv\ | \\ & \mathsf{Tau}\ CCSv\ | \\ & \mathsf{Par}\ CCSv\ CCSv\ | \\ & \mathsf{Restr}\ CCSv\ ((\alpha)labelv)set\ | \\ & \mathsf{Relab}\ CCSv\ (\alpha \rightarrow \alpha)\ | \\ & \mathsf{Cond}\ bool\ CCSv \end{aligned}$$

Note that the type variable β represents the type of both communication values and parameters of agent constants. It often happens that values and parameters are of the same type, e.g. the register example at the end of Section 3.1.1. Whenever values and parameters happen to range over different types, the type variable β will be instantiated to the disjoint union of the two types, or two type variables for the data component could be introduced in the mechanisation of value-passing expressions.

The following theorem of higher order logic characterises the type $(\gamma, \beta, \alpha)CCSv$:

$$
\begin{array}{l}
\vdash \forall f_1\ f_2\ f_3\ f_4\ f_5\ f_6\ f_7\ f_8\ f_9\ f_{10}. \\
\quad \exists!\ fn. \\
\qquad (\forall s1.\ fn(\mathsf{Var}\ s1) = f_1\ s1) \wedge \\
\qquad (\forall s1\ x2.\ fn(\mathsf{Conp}\ s1\ x2) = f_2\ s1\ x2) \wedge \\
\qquad (\forall f1\ s2.\ fn(\mathsf{Summ}\ f1\ s2) = f_3\ (fn \circ f1)\ f1\ s2) \wedge \\
\qquad (\forall x1\ f2.\ fn(\mathsf{In}\ x1\ f2) = f_4\ (fn \circ f2)\ x1\ f2) \wedge \\
\qquad (\forall x1\ x2\ C3.\ fn(\mathsf{Out}\ x1\ x2\ C3) = f_5\ (fn\ C3)\ x1\ x2\ C3) \wedge \\
\qquad (\forall C1.\ fn(\mathsf{Tau}\ C1) = f_6\ (fn\ C1)\ C1) \wedge \\
\qquad (\forall C1\ C2.\ fn(\mathsf{Par}\ C1\ C2) = f_7\ (fn\ C1)\ (fn\ C2)\ C1\ C2) \wedge \\
\qquad (\forall C1\ s2.\ fn(\mathsf{Restr}\ C1\ s2) = f_8\ (fn\ C1)\ C1\ s2) \wedge \\
\qquad (\forall C1\ f2.\ fn(\mathsf{Relab}\ C1\ f2) = f_9\ (fn\ C1)\ C1\ f2) \wedge \\
\qquad (\forall b1\ C2.\ fn(\mathsf{Cond}\ b1\ C2) = f_{10}\ (fn\ C2)\ b1\ C2)
\end{array}
$$

This theorem is the basis for reasoning about value-passing agent expressions in HOL. The formalisation of the inactive agent **nil**, the binary summation '+' and parameterless agent constants over the value-passing calculus, represented in HOL by the operators Nil, Sum and Con respectively, is similar to the one for the operators nil, sum and con in pure CCS. The two-armed conditional operator is defined as follows:

$$\forall b\ E\ E'.\ \mathsf{Cond\_Else}\ b\ E\ E' = \mathsf{Sum}\ (\mathsf{Cond}\ b\ E)\ (\mathsf{Cond}\ \neg b\ E')$$

and theorems asserting that Cond_Else is one-to-one and distinct from the other operators are derived similarly to the ones for Nil, Sum and Con.

The defining equations for value-passing agent constants are introduced through a type $(\beta, \gamma, \alpha)Def\_Funv$,[3] which is similar to the corresponding type for pure defining equations (Section 3.1.2). The definition of $(\beta, \gamma, \alpha)Def\_Funv$ is based on the function Fv_v for computing the free variables in value-passing expressions and the predicate Is_Agentv over the value-passing syntax. The type $(\beta, \gamma, \alpha)Def\_Funv$ is provided with functions Lookupv and Def_Funv, namely the value-passing versions of Lookup and Def_Fun.

Pure actions are just a special case of value-passing actions, namely those which do not take any value parameter. Likewise, prefixing a pure action to a value-passing expression is a particular case of the value-passing prefixing operations. In what follows, the HOL formalisation of these special cases is briefly discussed.

Actions with no value parameters can be defined in terms of value-passing actions of type $(\alpha, \beta)actionv$ by assigning them a special value different from all the values in the type β. This can be done by augmenting the value domain with a type which provides this special value. In HOL the disjoint sum $one + \beta$ can be taken as the type for the new value domain, where *one* is the HOL type containing the only constant one

[3]Note that the type variables before the type constructor *Def_Funv* occur in an order different from the one in the type $(\gamma, \beta, \alpha)CCSv$. This is due to the order in which they appear in the associated HOL definitions.

(Section A.2) and β is the actual value domain. In this way, any value-passing action $u : (\alpha, one + \beta)actionv$ with no value parameters can be defined through a constant definition as follows:

$$u = \mathsf{labelv}\ (\mathsf{in}\ p)\ (\mathsf{Inl}\ \mathsf{one})$$

if u is an input action (corresponding to a name in the pure calculus) and

$$u = \mathsf{labelv}\ (\mathsf{out}\ p)\ (\mathsf{Inl}\ \mathsf{one})$$

if u is an output action (corresponding to a co-name in the pure calculus), where $p : \alpha$ is the port of the given action. As an example, let α be the HOL type *string* and let the common acknowledgement action *ack* and its complement $\overline{ack}$ be defined as

$$ack = \mathsf{labelv}\ (\mathsf{in}\ \text{`}ack\text{`})\ (\mathsf{Inl}\ \mathsf{one})$$
$$ack\_bar = \mathsf{labelv}\ (\mathsf{out}\ \text{`}ack\text{`})\ (\mathsf{Inl}\ \mathsf{one})$$

It can now be proved that *ack_bar* is indeed the complementary action of *ack*. By rewriting with the definitions of *ack*, *ack_bar*, Compl_Actv and Complv, the following theorem is derived:

$$\vdash \mathsf{Compl\_Actv}\ ack = ack\_bar$$

Given the type $(\gamma, one + \beta, \alpha)CCSv$ for value-passing expressions, the operators which prefix an expression E with an input or output action with no values can be defined in terms of the constructors In and Out by means of the following constant definitions:

$$\forall p\ E.\ \mathsf{Inp}\ p\ E = \mathsf{In}\ p\ (\lambda x.\ (\mathsf{Isl}\ x \Rightarrow E \mid \mathsf{Nil}))$$
$$\forall p\ E.\ \mathsf{Outp}\ p\ E = \mathsf{Out}\ p\ (\mathsf{Inl}\ \mathsf{one})\ E$$

On one hand, the λ-abstraction in the definition of the "pure" input prefix operator Inp returns the expression E if the input value is of type *one*, namely it is its left component (as values are of type $one + \beta$). On the other hand, the value expression in the definition of the "pure" output prefix operator Outp is just the left injection of the special value one. For example, the terms Inp '*ack*' Nil and Outp '*ack*' Nil denote the agents that send and receive (respectively) an acknowledgement and then terminate. The operational semantics of these particular value-passing expressions will be derived in Section 4.3.

4.2 Translating Value-Passing CCS into Pure CCS

To help understanding the HOL formalisation of the translation from the value-passing syntax to the pure one, Milner's translation from [97] is recalled below. For each value-passing expression E (without free value variables) on the left, the

translation $\widehat{E}$ is given on the right:

X	X
$A(e_1, \ldots, e_n)$	$A_{e_1,\ldots,e_n}$
$\sum_{i\in I} E_i$	$\sum_{i\in I} \widehat{E_i}$
$a(x).E$	$\sum_{v\in V} a_v.E\widehat{\{v/x\}}$
$\overline{a}(e).E$	$\overline{a}_e.\widehat{E}$
$\tau.E$	$\tau.\widehat{E}$
$E_1 \mid E_2$	$\widehat{E_1} \mid \widehat{E_2}$
$E\backslash L$	$\widehat{E}\backslash\{l_v : l \in L,\ v \in V\}$
$E[f]$	$\widehat{E}[\widehat{f}]$ where $\widehat{f}(l_v) = f(l)_v$
if b **then** E	$\begin{cases} \widehat{E} & \text{if } b = \text{true} \\ \textbf{nil} & \text{otherwise} \end{cases}$

Moreover, the defining equation $A(\tilde{x}) \stackrel{\text{def}}{=} E$ for an agent constant, where $\tilde{x}$ stands for $x_1, \ldots, x_n$, is translated into the indexed set of defining equations $\{A_{\tilde{v}} \stackrel{\text{def}}{=} E\widehat{\{\tilde{v}/\tilde{x}\}} : \tilde{v} \in V^n\}$.

The translation from the value-passing calculus into pure CCS is given recursively on the structure of value-passing expressions. It is not primitive recursive though, as in the clause for the input prefix constructor, $\widehat{a(x).E} = \sum_{v\in V} a_v.E\widehat{\{v/x\}}$, the recursive occurrence of the translation function is not applied to the subterm E but to the expression $E\{v/x\}$. However, the formalisation of the input prefix operator through λ-abstraction and composition of functions makes it possible to define the translation in HOL as a primitive recursive function over the type $(\gamma, \beta, \alpha)CCSv$. This is achieved by composing the translation function with the function argument of the input prefix operator and using the extended version of the mechanism for primitive recursive function definition (Section A.4).

Given an expression $E : (\gamma, \beta, \alpha)CCSv$ and a value domain $V : (\beta)set$, the recursive function CCSv_To_CCS : $(\gamma, \beta, \alpha)CCSv \to (\beta)set \to (\gamma + \beta, \beta, \alpha \times \beta)CCS$ translates E into a pure expression whose types for indexes, parameters and ports are $\gamma + \beta$, β and $\alpha \times \beta$ respectively. The disjoint sum $\gamma + \beta$ instantiates the type variable for indexes in the type for pure expressions, because the result of translating an input prefixed expression of type $(\gamma, \beta, \alpha)CCSv$ is a summation over the value domain V: $(\beta)set$. This means that the indexing domain of the resulting expression is extended to include values as indexes too. Moreover, the type variable for parameters in the type for pure agent expressions is the same variable β for the data component in value-passing expressions. The function CCSv_To_CCS also takes as an argument the value domain V. This is due to the presence of predicates of the form $v \in V$ in the translation of the input prefix and restriction operators. The translation function is defined in HOL

as follows:

$$
\begin{array}{l}
(\forall s\ V.\ \mathsf{CCSv\_To\_CCS}\ (\mathsf{Var}\ s)\ V = \mathsf{var}\ s) \wedge \\
(\forall s\ x\ V.\ \mathsf{CCSv\_To\_CCS}\ (\mathsf{Conp}\ s\ x)\ V = \mathsf{conp}\ s\ x) \wedge \\
(\forall f\ I\ V. \\
\quad \mathsf{CCSv\_To\_CCS}\ (\mathsf{Summ}\ f\ I)\ V = \\
\quad \mathsf{summ}\ (\lambda i.\ (\mathsf{CCSv\_To\_CCS} \circ f)\ (\mathsf{Outl}\ i)\ V)\ \{\mathsf{Inl}\ j : j \in I\}) \wedge \\
(\forall p\ f\ V. \\
\quad \mathsf{CCSv\_To\_CCS}\ (\mathsf{In}\ p\ f)\ V = \\
\quad \mathsf{summ} \\
\quad (\lambda v.\ \mathsf{prefix}\ (\mathsf{label}\ (\mathsf{name}\ (p, \mathsf{Outr}\ v)))\ ((\mathsf{CCSv\_To\_CCS} \circ f)\ (\mathsf{Outr}\ v)\ V)) \\
\quad \{\mathsf{Inr}\ v' : v' \in V\}) \wedge \\
(\forall p\ e\ E\ V. \\
\quad \mathsf{CCSv\_To\_CCS}\ (\mathsf{Out}\ p\ e\ E)\ V = \\
\quad \mathsf{prefix}\ (\mathsf{label}\ (\mathsf{coname}\ (p, e)))\ (\mathsf{CCSv\_To\_CCS}\ E\ V)) \wedge \\
(\forall\ E\ V.\ \mathsf{CCSv\_To\_CCS}\ (\mathsf{Tau}\ E)\ V = \mathsf{prefix}\ \mathsf{tau}\ (\mathsf{CCSv\_To\_CCS}\ E\ V)) \wedge \\
(\forall E\ E'\ V. \\
\quad \mathsf{CCSv\_To\_CCS}\ (\mathsf{Par}\ E\ E')\ V = \mathsf{par}\ (\mathsf{CCSv\_To\_CCS}\ E\ V)(\mathsf{CCSv\_To\_CCS}\ E'\ V)) \wedge \\
(\forall E\ L\ V. \\
\quad \mathsf{CCSv\_To\_CCS}\ (\mathsf{Restr}\ E\ L)\ V = \\
\quad \mathsf{restr} \\
\quad (\mathsf{CCSv\_To\_CCS}\ E\ V) \\
\quad (\{\mathsf{name}\ (p, v) : (\mathsf{in}\ p) \in L\ \wedge\ v \in V\} \cup \{\mathsf{coname}\ (p, v) : (\mathsf{out}\ p) \in L\ \wedge\ v \in V\})) \wedge \\
(\forall E\ rf\ V. \\
\quad \mathsf{CCSv\_To\_CCS}\ (\mathsf{Relab}\ E\ rf)\ V = \\
\quad \mathsf{relab}\ (\mathsf{CCSv\_To\_CCS}\ E\ V)\ (\mathsf{ABS\_Relabelling}\ (\mathsf{Relab\_TR}\ rf))) \wedge \\
(\forall\ b\ E\ V.\ \mathsf{CCSv\_To\_CCS}\ (\mathsf{Cond}\ b\ E)\ V = (b\ \Rightarrow\ \mathsf{CCSv\_To\_CCS}\ E\ V \mid \mathsf{nil}))
\end{array}
$$

where ABS_Relabelling is the abstraction function of the type $(\alpha)relabelling$, and the function Relab_TR: $(\alpha \to \alpha) \to (\alpha \times \beta)label \to (\alpha \times \beta)label$ translates a value-passing relabelling into a function that renames pure labels which are, in turn, the translations of value-passing labels (see below for the translation of value-passing labels and actions):

$$
\begin{array}{l}
(\forall\ rf\ px.\ \mathsf{Relab\_TR}\ rf\ (\mathsf{name}\ px)\ =\ \mathsf{name}\ (rf\,(\mathsf{Fst}\ px), \mathsf{Snd}\ px)) \wedge \\
(\forall\ rf\ px.\ \mathsf{Relab\_TR}\ rf\ (\mathsf{coname}\ px)\ =\ \mathsf{coname}\ (rf\,(\mathsf{Fst}\ px), \mathsf{Snd}\ px))
\end{array}
$$

As far as the translation of agent constants is concerned, when $A(e_1, \ldots, e_n)$ is translated into the pure constant $A_{e_1,\ldots,e_n}$, it is implicitly assumed that the set of pure agent constants is infinite (or "big enough", depending on V), and the defining equations are translated into a (possibly infinite) set of pure defining equations. Due to the way parameterised agent constants have been formalised in HOL, a value-passing agent constant Conp s x is simply translated into the corresponding pure one conp s x with the same name and parameters. The value-passing defining equations Df are translated by means of the following function Df_TR, which translates

the value-passing agent Lookupv $Df\ s\ x$ corresponding to the value-passing constant identified by the name s and parameters x:

$$\forall V\ Df.\ \mathsf{Df\_TR}\ V\ Df = \mathsf{Def\_Fun}\ (\lambda s\ x.\ \mathsf{CCSv\_To\_CCS}\ (\mathsf{Lookupv}\ Df\ s\ x)\ V)$$

Note that the condition that the value-passing expression being translated does not contain any free value variables is not formalised in the above definition for CCSv_To_CCS. This is due to the fact that no knowledge is assumed about the value domain, hence it is not possible to express such a condition on value-passing expressions. Whenever a specific data domain is considered, the structure of its expressions will be known together with the operators to construct them and consequently it will be possible to formalise the above condition.

The formalisation of an input prefixed expression $a(x).E$ through a λ-abstraction that binds the variable x in the expression E, allows one to use the β-reduction in the HOL logic to mechanise the substitution of a value constant v for a variable x in value-passing expressions (i.e. $E\{v/x\}$), boolean expressions b and value expressions e. Let E be the agent expression

$$in(x).\ (\mathbf{if}\ x < 5\ \mathbf{then}\ \overline{out}(x+1).\ \mathbf{nil})$$

where the labels of ports are strings, the value domain V is the set of natural numbers, and indexes are of any type γ. The expression E is represented in HOL as the term t

$$\mathsf{In}\ in\ (\lambda x.\ \mathsf{Cond}\ (x<5)\ (\mathsf{Out}\ out\ (x+1)\ \mathsf{Nil})) : (\gamma, num, string)CCSv$$

By applying the translation function CCSv_To_CCS to the term t and the value domain $\mathbb{N}$ of natural numbers, represented in HOL by the universal set Univ : $(num)set$, and by rewriting with the definition of composition of functions and β-reduction, the following theorem is obtained:

$$\begin{array}{l}
\vdash \mathsf{CCSv_To_CCS}\ (\mathsf{In}\ in\ (\lambda x.\ \mathsf{Cond}\ (x<5)\ (\mathsf{Out}\ out\ (x+1)\ \mathsf{Nil})))\ \mathsf{Univ} = \\
\quad \mathsf{summ} \\
\quad (\lambda v. \\
\quad\quad \mathsf{prefix} \\
\quad\quad (\mathsf{label}\ (\mathsf{name}\ (in, \mathsf{Outr}\ v))) \\
\quad\quad ((\mathsf{Outr}\ v) < 5\ \Rightarrow \\
\quad\quad\ \mathsf{prefix}\ (\mathsf{label}\ (\mathsf{coname}\ (out,\ (\mathsf{Outr}\ v)+1)))\ (\mathsf{CCSv_To_CCS}\ \mathsf{Nil}\ \mathsf{Univ})\ \mid \\
\quad\quad\ \mathsf{nil})) \\
\quad \{\mathsf{Inr}\ v' : v' \in \mathsf{Univ}\}
\end{array}$$

whose right-hand side is the translation of t of type $(\gamma{+}num, num, string{\times}num)CCS$. This term is similar to the schema of expressions given in [97], where the bound variable x has been replaced by the value Outr v both in the boolean expression $x<5$ and in the value expression $x+1$. Note that, due to the definitions of the inactive

agent and of CCSv_To_CCS, it is not possible to prove that the translation of the value-passing inactive agent Nil is equal (i.e. '=' in the HOL sense) to nil. However, it is possible to prove that these two (pure) expressions perform the same transitions (none in their case). Thus, any behavioural equivalence between them can be derived, as asserted by the following theorem for observation congruence:

OBS_CONGR_CCSv_To_CCS_Nil: $\vdash \forall V\ Df.$ Obs_Congr Df (CCSv_To_CCS Nil V) nil

The same holds for binary summation

OBS_CONGR_CCSv_To_CCS_Sum:
$\vdash \forall V\ Df.\ \forall E\ E' ::$ Is_Agentv V.
Obs_Congr
Df
(CCSv_To_CCS (Sum $E\ E'$) V)
(sum (CCSv_To_CCS $E\ V$) (CCSv_To_CCS $E'\ V$))

and for the two-armed conditional:

OBS_CONGR_CCSv_To_CCS_Cond_Else:
$\vdash \forall V\ Df\ b.\ \forall E\ E' ::$ Is_Agentv V.
Obs_Congr
Df
(CCSv_To_CCS (Cond_Else $b\ E\ E'$) V)
(b $\Rightarrow$ CCSv_To_CCS $E\ V$ | CCSv_To_CCS $E'\ V$)

Note that the predicate Is_Agentv : $(\beta)set \rightarrow (\gamma, \beta, \alpha)CCSv \rightarrow bool$ is parameterised on the value domain V, in a way similar to CCSv_To_CCS. Rewriting with the above behavioural theorems does not cause any problems, as agent expressions are typically manipulated modulo a behavioural equivalence. Nevertheless, when reasoning about value-passing agents, it would be preferable to work directly on the value-passing expressions without resorting to their translation into pure CCS. In the following sections, the development of a proof environment for the value-passing calculus is described based on the translation and the HOL formalisation of pure CCS. Once this environment has been derived, reasoning will be performed on the value-passing agents without translating them into pure expressions.

Finally, it is worth noting that no translation functions for value-passing labels and actions are used in the definition of CCSv_To_CCS. This is due to the formalisation of the input, output and τ-prefix operators which do not have an action as parameter. However, translation functions for value-passing labels and actions will be necessary in Section 4.4 when proving the correctness of the translation. Their definitions are briefly outlined below.

The presence of a type variable for the ports in the formalisation of pure actions (Section 3.3.1) allows us to translate the value-passing actions into the pure ones

by simply mapping each input action $a(v)$ into a pair $(a, v) : (\alpha \times \beta)action$. For example, if a is a constant of type *string*, then the value-passing action labelv (in a) 5 of type $(string, num)actionv$ is translated into the pure action label (name $(a, 5)$) of type $(string \times num)action$, where the compound type $string \times num$ instantiates the type variable in $(\alpha)action$. Similarly, an output action $\overline{a}(e) : (\alpha, \beta)actionv$ is translated into a pure action $\overline{(a, e)} : (\alpha \times \beta)action$. The translation from value-passing labels to pure ones is thus defined:

$$(\forall p\, x.\ \mathsf{Labelv\_TR}\ (\mathsf{in}\ p)\ x\ =\ \mathsf{name}\ (p, x)) \wedge$$
$$(\forall p\, x.\ \mathsf{Labelv\_TR}\ (\mathsf{out}\ p)\ x\ =\ \mathsf{coname}\ (p, x))$$

The translation from value-passing actions to pure ones is defined as follows:

$$(\mathsf{Actionv\_TR}\ \mathsf{tauv}\ =\ \mathsf{tau}) \wedge$$
$$(\forall l\, x.\ \mathsf{Actionv\_TR}\ (\mathsf{labelv}\ l\ x)\ =\ \mathsf{label}\ (\mathsf{Labelv\_TR}\ l\ x))$$

Both translations for value-passing labels and actions are proved to be one-to-one.

4.3 The Operational Semantics for Value-Passing CCS

Before developing a proof environment for value-passing specifications based on the translation into pure CCS, let us check that the translation defined in the previous section is *correct* with respect to the semantics of the value-passing calculus. In other words, one would like to prove that the translation *preserves* all the transitions that a value-passing agent expression can perform. Given any value-passing expression E, this means that if $E \xrightarrow{u} E'$ for some action u and agent expression E', then $\widehat{E} \xrightarrow{\widehat{u}} \widehat{E'}$, namely the translation of E is able to perform the pure action resulting from the translation of u and then evolve to the translation of E'.

To achieve this correctness result, an operational semantics based on labelled interleaving transitions is first given to the value-passing operators. Given value domain V and defining equations Df, the transition relation $E \xrightarrow{u} E'$ is inductively defined by the inference rules in Figure 4.1.

The transition relation for value-passing agent expressions is formalised in HOL using the derived rule for inductive definitions. A transition $E \xrightarrow{u} E'$ with respect to a value domain V and a defining equations function Df, is represented in HOL by Transv $V\ Df\ E\ u\ E'$. The relation Transv : $(\beta)set \rightarrow (\beta, \gamma, \alpha)Def\_Funv \rightarrow (\gamma, \beta, \alpha)CCSv \rightarrow (\alpha, \beta)actionv \rightarrow (\gamma, \beta, \alpha)CCSv \rightarrow bool$ is parameterised on V and Df, thus returning a class of inductively defined relations, one for each given value domain and defining

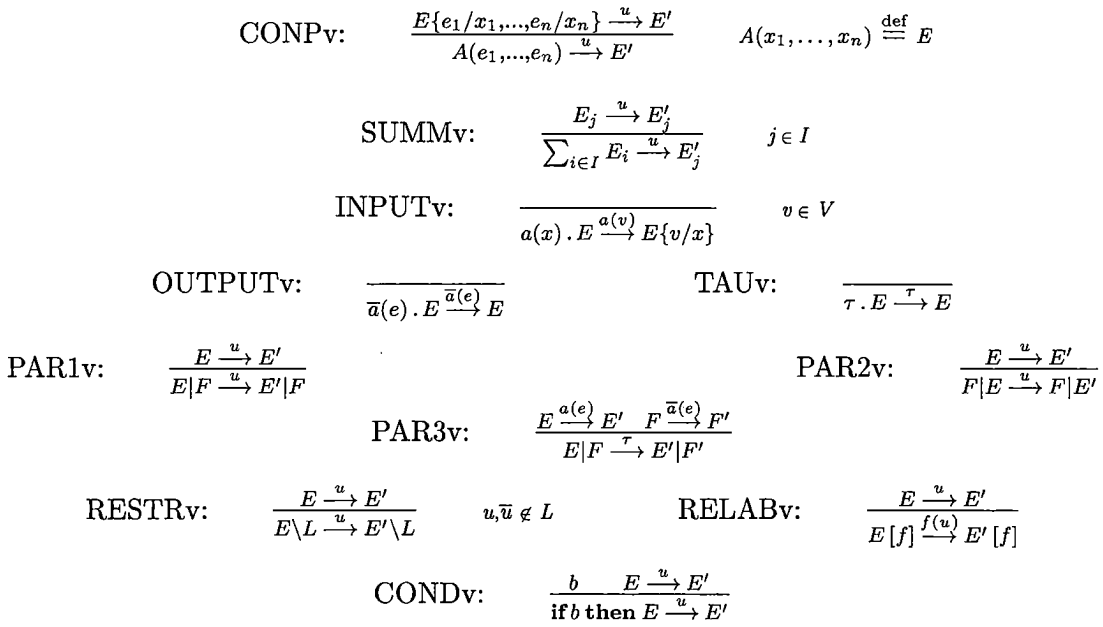

Figure 4.1: The transition rules for value-passing CCS.

equations. The labelled transition rules in Figure 4.1 are the following HOL theorems:

CONPv: $\vdash \forall V\ Df\ s\ e\ u\ E.$
Transv $V\ Df$ (Lookupv $Df\ s\ e$) $u\ E\ \supset$ Transv $V\ Df$ (Conp $s\ e$) $u\ E$

SUMMv: $\vdash \forall V\ Df\ f\ u\ E\ I.$
$(\exists i.$ Transv $V\ Df\ (f\ i)\ u\ E\ \wedge\ i \in I)\ \supset$
Transv $V\ Df$ (Summ $f\ I$) $u\ E$

INPUTv: $\vdash \forall V\ Df\ e.\ e \in V\ \supset\ (\forall p\ f.$ Transv $V\ Df$ (In $p\ f$) (labelv (in p) e) $(f\ e))$

OUTPUTv: $\vdash \forall V\ Df\ p\ e\ E.$ Transv $V\ Df$ (Out $p\ e\ E$) (labelv (out p) e) E

TAUv: $\vdash \forall V\ Df\ E.$ Transv $V\ Df$ (Tau E) tauv E

PAR1v: $\vdash \forall V\ Df\ E\ u\ E1.$
Transv $V\ Df\ E\ u\ E1\ \supset$
$(\forall E'.$ Transv $V\ Df$ (Par $E\ E'$) u (Par $E1\ E'$))

PAR2v: $\vdash \forall V\ Df\ E\ u\ E1.$
Transv $V\ Df\ E\ u\ E1\ \supset$
$(\forall E'.$ Transv $V\ Df$ (Par $E'\ E$) u (Par $E'\ E1$))

PAR3v: $\vdash \forall V\ Df\ E\ E1\ E'\ E2.$
$(\exists l\ e.$
Transv $V\ Df\ E$ (labelv $l\ e$) $E1\ \wedge$
Transv $V\ Df\ E'$ (labelv (Complv l) e) $E2)\ \supset$
Transv $V\ Df$ (Par $E\ E'$) tauv (Par $E1\ E2$)

RESTRv: $\vdash \forall V\ Df\ E\ u\ E'\ L.$
$(\exists l\ e.$
$\mathsf{Transv}\ V\ Df\ E\ u\ E' \wedge$
$((u = \mathsf{tauv}) \vee$
$((u = \mathsf{labelv}\ l\ e) \wedge (l \notin L) \wedge (\mathsf{Complv}\ l \notin L)))) \supset$
$\mathsf{Transv}\ V\ Df\ (\mathsf{Restr}\ E\ L)\ u\ (\mathsf{Restr}\ E'\ L)$

RELABv: $\vdash \forall V\ Df\ E\ u\ E'.$
$\mathsf{Transv}\ V\ Df\ E\ u\ E' \supset$
$(\forall rf.\ \mathsf{Transv}\ V\ Df\ (\mathsf{Relab}\ E\ rf)\ (\mathsf{relabelv}\ rf\ u)\ (\mathsf{Relab}\ E'\ rf))$

CONDv: $\vdash \forall V\ Df\ b\ E\ u\ E'.$
$b \wedge \mathsf{Transv}\ V\ Df\ E\ u\ E' \supset \mathsf{Transv}\ V\ Df\ (\mathsf{Cond}\ b\ E)\ u\ E'$

Note that, due to the HOL formalisation of agent constants and defining equations, the substitution $\{e_1/x_1, \ldots, e_n/x_n\}$ in CONPv is implemented in the corresponding HOL rule by simply replacing the formal parameters $x_1, \ldots, x_n$ with the actual ones $e_1, \ldots, e_n$. Moreover, the HOL rule INPUTv allows any value expression e over V to be received without being first evaluated to its value v. Thus, given an input port p, in HOL an input action is the more general $p(e)$ rather than $p(v)$. Any value expression can be exchanged and its value is evaluated *by need* and checked to be in the value domain V. This guarantees that all the exchanged data belong to V. Finally, the definition of INPUTv implies that the *early* version of behavioural equivalences for value-passing CCS will be considered.

The transitions for value-passing expressions which contains actions with no value parameters are derived from the above rules. If Q is the agent Outp '*ack*' Nil given at the end of Section 4.1, its transitions can be computed by simply rewriting with the definitions of Q, Outp, *ack_bar* and the rule OUTPUTv to get the following theorem:

$$\vdash \mathsf{Transv}\ V\ Df\ Q\ ack\_bar\ \mathsf{Nil}$$

where V is the set $\{v : v = \mathsf{Inl}\ \mathsf{one}\}$ of all values in disjoint sum with one on the left and Df is any defining equations function over value-passing agent constants. If P is the agent Inp '*ack*' Q, its actions can be similarly computed by rewriting with the definitions of Inp, *ack*, the rule INPUTv (suitably instantiated) and the theorem ISL (from the theory sum, Section A.2). The following transition is derived:

$$\vdash \mathsf{Transv}\ V\ Df\ P\ ack\ Q$$

This theorem asserts that the value-passing agent P can do the action *ack* and evolve into the agent Q.

4.4 Proving the Correctness of the Translation

The issue of correctness of the translation, mentioned at the beginning of the previous section, is now considered. The following result is proved in HOL:

```
TRANSv_IMP_TRANS:
⊢ ∀V Df E u E'.
    Transv V Df E u E' ⊃
    Trans (Df_TR V Df) (CCSv_To_CCS E V) (Actionv_TR u) (CCSv_To_CCS E' V)
```

This theorem is proved by rule induction, namely by induction over the structure of the derivations defined by the transition rules for Transv. This induction gives rise to 11 subgoals, one for each transition rule. Let us show the proof of two of them, namely the cases of value-passing indexed summation and input prefixed expressions. The subgoal to be proved for the summation case is the following:

```
{ Trans (Df_TR V Df) (CCSv_To_CCS (f i) V) (Actionv_TR u) (CCSv_To_CCS E V) ;
  i ∈ I }
?⊢ Trans (Df_TR V Df)
   (CCSv_To_CCS (Summ f I) V)
   (Actionv_TR u)
   (CCSv_To_CCS E V)
```

The assumptions of the goal are just the premisses of the inference rule SUMMv to which induction has been applied. The first step of the proof consists of rewriting the goal with the definition of the translation and the theorem o_THM for composition of functions, i.e. $\vdash \forall f\ g\ x.\ (f \circ g)\ x = f\ (g\ x)$:

```
{ Trans (Df_TR V Df) (CCSv_To_CCS (f i) V) (Actionv_TR u) (CCSv_To_CCS E V) ;
  i ∈ I }
?⊢ Trans (Df_TR V Df)
   (summ (λi. CCSv_To_CCS (f (Outl i)) V) {Inl j : j ∈ I})
   (Actionv_TR u)
   (CCSv_To_CCS E V)
```

The transition rule SUMM (Section 3.3.2) for a pure summation agent expression is applied in a backward manner to reduce the goal to the following:

```
{ Trans (Df_TR V Df) (CCSv_To_CCS (f i) V) (Actionv_TR u) (CCSv_To_CCS E V) ;
  i ∈ I }
?⊢ ∃i.
     Trans (Df_TR V Df)
     ((λi. CCSv_To_CCS (f (Outl i)) V) i) (Actionv_TR u) (CCSv_To_CCS E V) ∧
     i ∈ {Inl j : j ∈ I}
```

Beta-reduction is applied and the existentially quantified variable i in the goal is then instantiated with the index $\mathsf{Inl}\ i : \gamma + \beta$ (due to the translation, indexes range over a disjoint sum type):

$$\begin{array}{l}\{\ \mathsf{Trans}\ (\mathsf{Df\_TR}\ V\ Df)\ (\mathsf{CCSv\_To\_CCS}\ (f\ i)\ V)\ (\mathsf{Actionv\_TR}\ u)\ (\mathsf{CCSv\_To\_CCS}\ E\ V)\ ; \\ \quad i \in I\ \} \\ ?\vdash\ \mathsf{Trans}\ (\mathsf{Df\_TR}\ V\ Df) \\ \quad (\mathsf{CCSv\_To\_CCS}\ (f\ (\mathsf{Outl}\ (\mathsf{Inl}\ i)))\ V)\ (\mathsf{Actionv\_TR}\ u)\ (\mathsf{CCSv\_To\_CCS}\ E\ V) \wedge \\ \quad (\mathsf{Inl}\ i) \in \{\mathsf{Inl}\ j : j \in I\}\end{array}$$

By rewriting with the theorem OUTL (from the theory `sum`, Section A.2), the first conjunct of the goal is reduced to the first assumption. Thus, the second conjunct is left to prove:

$$\begin{array}{l}\{\ \mathsf{Trans}\ (\mathsf{Df\_TR}\ V\ Df)\ (\mathsf{CCSv\_To\_CCS}\ (f\ i)\ V)\ (\mathsf{Actionv\_TR}\ u)\ (\mathsf{CCSv\_To\_CCS}\ E\ V)\ ; \\ \quad i \in I\ \} \\ ?\vdash\ (\mathsf{Inl}\ i) \in \{\mathsf{Inl}\ j : j \in I\}\end{array}$$

Using set theory and the assumption $i \in I$, the subgoal for the case of indexed summation is finally proved. A similar proof works for the transitions of an input prefixed expression, except that the translation of actions is needed besides the one for expressions. The subgoal is as follows:

$$\begin{array}{l}\{\ e \in V\ \} \\ ?\vdash\ \forall p\ f. \\ \quad \mathsf{Trans}\ (\mathsf{Df\_TR}\ V\ Df) \\ \quad (\mathsf{CCSv\_To\_CCS}\ (\mathsf{In}\ p\ f)\ V)\ (\mathsf{Actionv\_TR}\ (\mathsf{labelv}\ (\mathsf{in}\ p)\ e))\ (\mathsf{CCSv\_To\_CCS}\ (f\ e)\ V)\end{array}$$

The only assumption is $e \in V$, namely the condition in the above transition rule INPUTv. Given any port p and input function f, rewriting with the definition of the translation and the theorem o_THM produces the new goal:

$$\begin{array}{l}\{\ e \in V\ \} \\ ?\vdash\ \mathsf{Trans}\ (\mathsf{Df\_TR}\ V\ Df) \\ \quad (\mathsf{summ} \\ \quad (\lambda v.\ \mathsf{prefix}\ (\mathsf{label}\ (\mathsf{name}\ (p, \mathsf{Outr}\ v)))\ (\mathsf{CCSv\_To\_CCS}\ (f\ (\mathsf{Outr}\ v))\ V)) \\ \quad \{\mathsf{Inr}\ v : v \in V\}) \\ \quad (\mathsf{Actionv\_TR}\ (\mathsf{labelv}\ (\mathsf{in}\ p)\ e)) \\ \quad (\mathsf{CCSv\_To\_CCS}\ (f\ e)\ V)\end{array}$$

The translations Actionv_TR and Labelv_TR for value-passing actions and labels are applied to rewrite the goal:

$\{\, e \in V \,\}$
$\vdash$ Trans (Df_TR V Df)
(summ
(λv. prefix (label (name (p, Outr v))) (CCSv_To_CCS (f (Outr v)) V))
{Inr $v : v \in V$})
(label (name (p, e)))
(CCSv_To_CCS (f e) V)

The proof now proceeds similarly to the one for the summation case by applying the rule SUMM backward and β-reduction:

$\{\, e \in V \,\}$
$\vdash$ $\exists i$.
Trans (Df_TR V Df)
(prefix (label (name (p, Outr i))) (CCSv_To_CCS (f (Outr i)) V))
(label (name (p, e)))
(CCSv_To_CCS (f e) V) $\wedge$
$i \in$ {Inr $v : v \in V$}

The existentially quantified variable i is instantiated with the index Inr $e : \gamma + \beta$, namely the right injection of the value expression $e \in V$. Rewriting with the theorem OUTR: $\vdash \forall x$. Outr (Inr x) $= x$ yields the following goal:

$\{\, e \in V \,\}$
$\vdash$ Trans (Df_TR V Df)
(prefix (label (name (p, e))) (CCSv_To_CCS (f e) V))
(label (name (p, e)))
(CCSv_To_CCS (f e) V) $\wedge$
(Inr e) $\in$ {Inr $v : v \in V$}

The goal is now solved by applying the theorem PREFIX for the prefix operator and using set theory and the assumption. The remaining subgoals of the proof by transition induction are proved with similar tactics.

One might wonder whether the implicational theorem TRANSv_IMP_TRANS can be made an equivalence. The problem is that the translation CCSv_To_CCS is not one-to-one, i.e. syntactically different value-passing terms can be translated into the same pure term. In fact, the translation is not purely syntactic as the clause for the conditional operator depends on the value of the boolean expression. Infinitely many (finite) value-passing terms can be constructed using the conditional operator so that they are syntactically different but are translated into the same pure term. Nevertheless, these value-passing terms have the same transitions. Thus, the opposite implication of TRANSv_IMP_TRANS could be derived by considering equivalence classes of value-passing terms having the same transitions, namely strong equivalent agents. A rather similar reasoning is carried out in [40] where the ECRINS system is

used for defining a translation from Basic LOTOS to MEIJE and proving its correctness automatically. The proof of correctness is structural and each LOTOS term is shown to have the same semantics as its corresponding MEIJE term.

The correctness of the translation expressed by TRANSv_IMP_TRANS allows one to soundly define the behavioural semantics for value-passing expressions in terms of the translation into the pure calculus and the corresponding semantics over pure expressions. In this way, properties and laws for the value-passing behavioural semantics can be derived through the corresponding theorems for pure CCS, as shown in the next section.

4.5 Behavioural Equivalences over Value-Passing CCS

The behaviour of a value-passing agent expression E can be explored by translating E into its pure version $\widehat{E}$ and then using the inference rules of the operational semantics for the pure calculus. Similarly, the equivalence of any two value-passing agents E_1 and E_2, with respect to a given behavioural semantics, can be verified by checking the behavioural equivalence between the two pure agents $\widehat{E_1}$ and $\widehat{E_2}$ resulting from the translation. In this way, behavioural equivalences over value-passing agents are directly defined in terms of the corresponding equivalences over pure agents without resorting to the notion of bisimulation.

The translation function CCSv_To_CCS is parameterised on the value domain V, as the result of translating a value-passing expression depends on the value domain under consideration. This is easily illustrated by an agent whose behaviour depends on the data component. Consider the recursive agent

$$A \stackrel{\text{def}}{=} a(x).\,\textbf{if } even(x) \textbf{ then } \overline{e}(x).\,A \textbf{ else } \overline{o}(x).\,A$$

The agent A repeatedly inputs a value $n \in V$ through the port a and then emits n through the port $\overline{e}$ if n is even, otherwise n is sent out through the port $\overline{o}$. The pure agent $\widehat{A}$ resulting from translating A is $\sum_{n \in V} a_n.\,(even(n) \Rightarrow \overline{e}_n.\,\widehat{A} \mid \overline{o}_n.\,\widehat{A})$, which denotes different agents depending on V.

When defining the transition relation Transv over value-passing expressions, the value domain V is given as a parameter to its definition, together with the defining equations Df for agent constants. The same thing happens when defining behavioural equivalences, as the behaviour of value-passing specifications depends on their data component. Two value-passing agents may perform the same transitions and thus be behaviourally equivalent with respect to a given value domain, but they are not if the value domain is changed, even slightly. For example, the two agents $B \stackrel{\text{def}}{=} a(x).\,\overline{b}(x+2).\,B$ and $C \stackrel{\text{def}}{=} a(x).\,\textbf{if } even(x) \textbf{ then } \overline{b}(x+2).\,C$ perform exactly the same actions (and are therefore equivalent) over the value domain of the even numbers,

while they are not equivalent over a larger domain including, for example, some odd numbers. What follows shows how behavioural equivalences over value-passing agents are defined in HOL based on the translation and the corresponding behavioural equivalences over pure CCS. In particular, observation equivalence and congruence are considered and the formalisation of some of their properties and laws is presented.

The HOL relation $\mathsf{Obs\_Equiv\_v} : (\beta)set \rightarrow (\beta, \gamma, \alpha)Def\_Funv \rightarrow (\gamma, \beta, \alpha)CCSv \rightarrow (\gamma, \beta, \alpha)CCSv \rightarrow bool$ denotes the observation equivalence over value-passing agents with respect to given value domain V and defining equations Df. The relation Obs_Equiv_v is defined in terms of Obs_Equiv, i.e. the observation equivalence over pure CCS (Section 2.2.3), as follows:

$$\begin{array}{l}\forall V\ Df.\ \forall E\ E' :: \mathsf{Is\_Agentv}\ V.\\ \quad \mathsf{Obs\_Equiv\_v}\ V\ Df\ E\ E' =\\ \quad \mathsf{Obs\_Equiv}\ (\mathsf{Df\_TR}\ V\ Df)\ (\mathsf{CCSv\_To\_CCS}\ E\ V)\ (\mathsf{CCSv\_To\_CCS}\ E'\ V)\end{array}$$

The observation congruence Obs_Congr_v is embedded in HOL in a similar way. Properties of both observation equivalence and congruence over value-passing agents can be easily derived. The standard proof consists of rewriting the property to be proved with the definition of the given behavioural equivalence and applying the corresponding property over pure agents. For example, the following theorems asserting that Obs_Equiv_v is an equivalence relation, are proved using this proof technique:

$$\begin{array}{ll}\text{OBSEQv\_REFL:} & \vdash \forall V\ Df.\ \forall E :: \mathsf{Is\_Agentv}\ V.\ \mathsf{Obs\_Equiv\_v}\ V\ Df\ E\ E\\ \text{OBSEQv\_SYM:} & \vdash \forall V\ Df.\ \forall E\ E' :: \mathsf{Is\_Agentv}\ V.\\ & \quad \mathsf{Obs\_Equiv\_v}\ V\ Df\ E\ E' \supset \mathsf{Obs\_Equiv\_v}\ V\ Df\ E'\ E\\ \text{OBSEQv\_TRANS:} & \vdash \forall V\ Df.\ \forall E\ E'\ E'' :: \mathsf{Is\_Agentv}\ V.\\ & \quad \mathsf{Obs\_Equiv\_v}\ V\ Df\ E\ E' \wedge \mathsf{Obs\_Equiv\_v}\ V\ Df\ E'\ E'' \supset\\ & \quad \mathsf{Obs\_Equiv\_v}\ V\ Df\ E\ E''\end{array}$$

Congruence (or substitutivity) properties are derived in a similar way. The following inference rules are two of them.

OBSEQv_SUBST_OUTPUT:

$$\begin{array}{l}\vdash \forall V\ Df\ e\ e'.\ \forall E\ E' :: \mathsf{Is\_Agentv}\ V.\\ \quad (e = e') \wedge \mathsf{Obs\_Equiv\_v}\ V\ Df\ E\ E' \supset\\ \quad (\forall p.\ \mathsf{Obs\_Equiv\_v}\ V\ Df\ (\mathsf{Out}\ p\ e\ E)\ (\mathsf{Out}\ p\ e'\ E'))\end{array}$$

OBSEQv_SUBST_INPUT:

$$\begin{array}{l}\vdash \forall V\ Df\ f\ f'.\\ \quad (\forall v.\ v \in V \supset\\ \qquad \mathsf{Is\_Agentv}\ V\ (f\ v) \wedge \mathsf{Is\_Agentv}\ V\ (f'\ v) \wedge \mathsf{Obs\_Equiv\_v}\ V\ Df\ (f\ v)\ (f'\ v)) \supset\\ \quad (\forall p.\ \mathsf{Obs\_Equiv\_v}\ V\ Df\ (\mathsf{In}\ p\ f)\ (\mathsf{In}\ p\ f'))\end{array}$$

The inference rule OBSEQv_SUBST_OUTPUT asserts that if the value expressions e and e' are equal and the agents E and E' are observation equivalent, then for any port

p the output prefixed agents $\overline{p}(e).\, E$ and $\overline{p}(e').\, E'$ are also observation equivalent. The theorem OBSEQv_SUBST_INPUT encodes the ω-data-rule for observation equivalence. If this rule is interpreted backward, this means that, in order to prove that two input prefixed agents with equal ports are observation equivalent, it is enough to show that the prefixed expressions (given through λ-abstractions) are observation equivalent agents when applied to any value v in V. In the proof in Section 4.6, a stronger version of this ω-data-rule will be used which strengthens observation equivalence to observation congruence in the presence of a prefix operator. This rule is

OBSEQv_OBSCv_INPUT:
$\vdash \forall V\ Df\ f\ f'.$
$\quad (\forall v.\ v \in V\ \supset$
$\quad\quad \mathsf{Is\_Agentv}\ V\ (f\ v) \wedge \mathsf{Is\_Agentv}\ V\ (f'\ v) \wedge \mathsf{Obs\_Equiv\_v}\ V\ Df\ (f\ v)\ (f'\ v)) \supset$
$\quad (\forall p.\ \mathsf{Obs\_Congr\_v}\ V\ Df\ (\mathsf{In}\ p\ f)\ (\mathsf{In}\ p\ f'))$

and can be seen as the value-passing version of proposition PROP6 (Section 3.2.1) for the input prefix case.

Many of the algebraic laws for the behavioural equivalences over the value-passing calculus, such as the τ-law $\tau.\, E \approx E$ for observation equivalence,

$$\text{TAU\_WEAKv:}\ \vdash \forall V\ Df.\ \forall E :: \mathsf{Is\_Agentv}\ V.\ \mathsf{Obs\_Equiv\_v}\ V\ Df\ (\mathsf{Tau}\ E)\ E$$

are derived with a simple proof technique: rewriting the law to be proved with the definition of the behavioural equivalence and then applying the corresponding law over pure agents. The proof of other laws, such as the ones for the parallel and restriction operators, is not so straightforward, but still rather simple. The following are a few laws for the parallel composition of two prefixed agents:[4]

OBSEQv_PAR_OUT_IN_SYNC:
$\vdash \forall e\ V.$
$\quad e \in V\ \supset$
$\quad (\forall\ Df\ p\ E\ f.$
$\quad\quad \mathsf{Is\_Agentv}\ V\ (\mathsf{Out}\ p\ e\ E)\ \wedge\ \mathsf{Is\_Agentv}\ V\ (\mathsf{In}\ p\ f)\ \supset$
$\quad\quad \mathsf{Obs\_Equiv\_v}\ V\ Df$
$\quad\quad (\mathsf{Par}\ (\mathsf{Out}\ p\ e\ E)\ (\mathsf{In}\ p\ f))$
$\quad\quad (\mathsf{Sum}$
$\quad\quad\ (\mathsf{Sum}\ (\mathsf{Out}\ p\ e\ (\mathsf{Par}\ E\ (\mathsf{In}\ p\ f)))(\mathsf{In}\ p\ (\lambda x.\ \mathsf{Par}\ (\mathsf{Out}\ p\ e\ E)\ (f\ x))))$
$\quad\quad\ (\mathsf{Tau}\ (\mathsf{Par}\ E\ (f\ e)))))$

[4]Note that the premisses about the value-passing expressions being agents could also be stated in a different way. For example, rather than saying '$\mathsf{Is\_Agentv}\ V\ (\mathsf{Out}\ p\ e\ E)$', it is enough to say '$\mathsf{Is\_Agentv}\ V\ E$' as the following theorem holds: $\vdash \forall V\ p\ e\ E.\ \mathsf{Is\_Agentv}\ V\ (\mathsf{Out}\ p\ e\ E) = \mathsf{Is\_Agentv}\ V\ E$. This theorem and a similar one for the operator In are used in the proof of these laws.

OBSCv_PAR_INPUT_INPUT:
$. \vdash \forall V\ Df\ p\ f\ p'\ f'.$
$\quad \mathsf{Is\_Agentv}\ V\ (\mathsf{In}\ p\ f)\ \wedge\ \mathsf{Is\_Agentv}\ V\ (\mathsf{In}\ p'\ f')\ \supset$
$\quad \mathsf{Obs\_Congr\_v}\ V\ Df$
$\quad (\mathsf{Par}\ (\mathsf{In}\ p\ f)\ (\mathsf{In}\ p'\ f'))$
$\quad (\mathsf{Sum}\ (\mathsf{In}\ p\ (\lambda x.\ \mathsf{Par}\ (f\ x)\ (\mathsf{In}\ p'\ f')))(\mathsf{In}\ p'\ (\lambda y.\ \mathsf{Par}\ (\mathsf{In}\ p\ f)\ (f'\ y))))$

OBSEQv_PAR_IN_OUT_NO_SYNC:
$. \vdash \forall p\ p'.$
$\quad \neg(p = p')\ \supset$
$\quad (\forall V\ Df\ f\ e\ E.$
$\quad\quad \mathsf{Is\_Agentv}\ V\ (\mathsf{In}\ p\ f)\ \wedge\ \mathsf{Is\_Agentv}\ V\ (\mathsf{Out}\ p'\ e\ E)\ \supset$
$\quad\quad \mathsf{Obs\_Equiv\_v}\ V\ Df$
$\quad\quad (\mathsf{Par}\ (\mathsf{In}\ p\ f)\ (\mathsf{Out}\ p'\ e\ E))$
$\quad\quad (\mathsf{Sum}\ (\mathsf{In}\ p\ (\lambda x.\ \mathsf{Par}\ (f\ x)\ (\mathsf{Out}\ p'\ e\ E)))\ (\mathsf{Out}\ p'\ e\ (\mathsf{Par}\ (\mathsf{In}\ p\ f)\ E))))$

The above theorems illustrate two important notions in the HOL formalisation of the value-passing calculus. First, there is no need to check that free and bound value variables of parallel agents have empty intersection. Free occurrences of value variables will not be captured because the HOL system automatically applies α-conversion and renames variables, if necessary. Second, the law OBSEQv_PAR_OUT_IN_SYNC is an implicational theorem, whose antecedent requires to check that the value expression being exchanged is in the value domain. This derives from the definition of the translation and it is interesting to note that this occurs only in those theorems where there is data communication, while it is not required when no communication takes place.

Substitution tactics for implicational theorems are defined using some functions for conditional rewriting in HOL developed by Wong [125]. For example, the tactic `OEv_LHS_IMP_SUBST1_TAC` substitutes a given implicational theorem in the left-hand side of an observational goal, by reducing the goal to two subgoals, one about the data and one about the process behaviour. The particular instance of the antecedent of the theorem being applied is added as a new assumption to the assumption list of the subgoal for the process behaviour. Tactics like `OEv_LHS(RHS)_SUBST1_TAC` can be used to rewrite with non-implicational theorems, where `LHS/RHS` means that the substitution is performed only on the left/right-hand side of the observational goal.

4.6 A Verification Example

This section shows how the proof environment derived for the value-passing calculus can be used to verify the correctness of specifications. In particular, Hennessy's example processes given in [59] are considered and proved to be observation congruent.

The following equation defines the abstract specification *Spec* of a simple system:

$$Spec \stackrel{\text{def}}{=} in(x).\ \textbf{if}\ div(6,x)\ \textbf{then}\ \overline{out}(6*x).\ Spec\ \textbf{else}\ Spec$$

Such a system repeatedly inputs a number n through a port in and then checks if n is a multiple of 6. If it is, the number $6*n$ is output along the port $\overline{out}$ and then the system goes back to input and check a new number. If 6 does not divide n, no action is performed and the system simply goes back to its initial state. The same behaviour can be obtained by composing in parallel two communicating agents P and Q, which check the divisibility by 6 of the input number in two steps. The agent constants P and Q are defined by the following equations:

$$P \stackrel{\text{def}}{=} in(x).\ \textbf{if}\ even(x)\ \textbf{then}\ \overline{b}(2*x).\, a(z).\, P\ \textbf{else}\ P$$

$$Q \stackrel{\text{def}}{=} b(x).\ \textbf{if}\ div(3,x)\ \textbf{then}\ \overline{out}(3*x).\, \overline{a}(0).\, Q\ \textbf{else}\ \overline{a}(0).\, Q$$

Then, they are composed in parallel and the ports a and b are made private to P and Q as follows:

$$Impl \stackrel{\text{def}}{=} (P \mid Q)\backslash\{a, b\}$$

Thus, the agent P inputs a number n through the port in and then checks if n is even. If not, P goes back to its initial state. If n is even, P communicates the value expression $2*n$ to Q through the private channel b and then waits for an acknowledgement from Q along the other private channel a. The agent Q tests whether 3 divides the received value $m = 2*n$. If it does, the number $3*m$ is sent out through the port $\overline{out}$ and then Q synchronises with P through the port a by sending an acknowledgement in the form of the value constant 0. After such a communication both P and Q can go back to their initial states. If m is not divisible by 3, only the communication along the channel a takes place and then P and Q are ready to input numbers again.[5]

In order to prove that the agent *Impl* is a correct implementation of the description *Spec*, it will be shown that *Spec* and *Impl* are observation congruent, namely

$$Spec \approx_c Impl \tag{4.1}$$

By unique fixpoint induction this reduces to proving:

$$Impl \approx_c in(x).\ \textbf{if}\ div(6,x)\ \textbf{then}\ \overline{out}(6*x).\, Impl\ \textbf{else}\ Impl \tag{4.2}$$

As unique fixpoint induction over value-passing agents is not yet formalised in the HOL system, the HOL proof of the observation congruence (4.1) will start from the congruence (4.2).

4.6.1 The Proof in HOL

All the expressions in this example are agents. Let DfH be the defining equations function for the parameterless agent constants *Spec*, P, Q and *Impl*. The value domain

[5] The acknowledgement could be defined as an action with no values, but the above definitions are as given in [59], except that *rec*-notation is here replaced by agent constants and defining equations.

under consideration is the universal set of natural numbers, that is Univ : $(num)set$ in HOL. There are no indexed summation agents in this example, so the type variable γ will represent any type of indexes. As done in the previous examples, value-passing agents will be parsed and pretty-printed, so that the notation in Section 4.1 can be adopted instead of its HOL representation. A few steps of the HOL proof, which follows the guideline in [59], are informally presented below. The corresponding HOL transcripts can be found in Appendix D.

Given the universal set of natural numbers as the value domain and the environment defined by DfH, the initial goal is the following observation congruence:

$\{ \exists a\, b.\, \neg(a = b) \}$
? Obs_Congr_v Univ DfH $Impl$ $(in(x).\ \mathbf{if}\ div(6,x)\ \mathbf{then}\ \overline{out}(6 * x).\ Impl\ \mathbf{else}\ Impl)$

A tactic similar to the lazy expansion strategy used for verifying pure specifications (Section 2.4.1) can also be adopted when reasoning about value-passing agents. The only difference is that, as the behaviour of value-passing specifications depends on the values being exchanged, the proof will be influenced by the data component. This means that some proof steps will perform, for example, case analysis on value and boolean expressions, thus splitting the proof in various subproofs.

The first step of the proof expands the left-hand side of the behavioural goal with the definitions of the agents $Impl$, P and Q, up to observation congruence:

$\{ \exists a\, b.\, \neg(a = b) \}$
? Obs_Congr_v Univ DfH
$((in(x).\ \mathbf{if}\ even(x)\ \mathbf{then}\ \overline{b}(2 * x).\ a(z).\ P\ \mathbf{else}\ P\ \mid$
$\quad b(x).\ \mathbf{if}\ div(3,x)\ \mathbf{then}\ \overline{out}(3 * x).\ \overline{a}(0).\ Q\ \mathbf{else}\ \overline{a}(0).\ Q)\backslash\{a,b\})$
$(in(x).\ \mathbf{if}\ div(6,x)\ \mathbf{then}\ \overline{out}(6 * x).\ Impl\ \mathbf{else}\ Impl)$

The left-hand side of the goal is an instance of the expansion theorem. This will be applied in a few steps, the first of which is to rewrite the internal parallel composition with the law OBSCv_PAR_INPUT_INPUT.

$\{ \exists a\, b.\, \neg(a = b) \}$
? Obs_Congr_v Univ DfH
$((in(x).$
$\quad (\mathbf{if}\ even(x)\ \mathbf{then}\ \overline{b}(2 * x).\ a(z).\ P\ \mathbf{else}\ P\ \mid$
$\quad\ b(x').\ \mathbf{if}\ div(3,x')\ \mathbf{then}\ \overline{out}(3 * x').\ \overline{a}(0).\ Q\ \mathbf{else}\ \overline{a}(0).\ Q)\ +$
$\quad b(y).$
$\quad (in(x).\ \mathbf{if}\ even(x)\ \mathbf{then}\ \overline{b}(2 * x).\ a(z).\ P\ \mathbf{else}\ P\ \mid$
$\quad\ \mathbf{if}\ div(3,y)\ \mathbf{then}\ \overline{out}(3 * y).\ \overline{a}(0).\ Q\ \mathbf{else}\ \overline{a}(0).\ Q))\backslash\{a,b\}$
$(in(x).\ \mathbf{if}\ div(6,x)\ \mathbf{then}\ \overline{out}(6 * x).\ Impl\ \mathbf{else}\ Impl)$

Note how the HOL system has renamed the bound value variables of the input prefixed terms inside the parallel composition. By applying the laws for the restriction

operator and for deleting inactive summands, the new goal is as follows:

$\{\ \exists\, a\, b.\, \neg(a = b)\ \}$
$\vdash$ Obs_Congr_v Univ DfH
$((in(x).$
$((\textbf{if}\ even(x)\ \textbf{then}\ \overline{b}(2 * x).\, a(z).\, P\ \textbf{else}\ P\ \mid$
$b(x').\ \textbf{if}\ div(3, x')\ \textbf{then}\ \overline{out}(3 * x').\, \overline{a}(0).\, Q\ \textbf{else}\ \overline{a}(0).\, Q)\backslash\{a, b\}$
$(in(x).\ \textbf{if}\ div(6, x)\ \textbf{then}\ \overline{out}(6 * x).\, Impl\ \textbf{else}\ Impl)$

So far, the steps of the proof have manipulated only the left-hand side of the observational goal with the aim of reducing it to a term provably equivalent to the right-hand side. The current goal is an observation congruence between two input prefixed agents with the same port. The ω-data-rule OBSEQv_OBSCv_INPUT can be applied backward, as the goal matches the conclusion of such a rule, and the whole goal is transformed:

$\{\ \exists\, a\, b.\, \neg(a = b)\ \}$
$\vdash\ \forall v.$
$v \in$ Univ $\supset$
Obs_Equiv_v Univ DfH
$((\lambda x.\ (\textbf{if}\ even(x)\ \textbf{then}\ \overline{b}(2 * x).\, a(z).\, P\ \textbf{else}\ P\ \mid$
$b(x').\ \textbf{if}\ div(3, x')\ \textbf{then}\ \overline{out}(3 * x').\, \overline{a}(0).\, Q\ \textbf{else}\ \overline{a}(0).\, Q)\backslash\{a, b\})\ v)$
$((\lambda x.\ \textbf{if}\ div(6, x)\ \textbf{then}\ \overline{out}(6 * x).\, Impl\ \textbf{else}\ Impl)\ v)$

The antecedent of the current goal is trivially true and then β-reduction is applied:

$\{\ \exists\, a\, b.\, \neg(a = b)\ \}$
$\vdash$ Obs_Equiv_v Univ DfH
$((\textbf{if}\ even(v)\ \textbf{then}\ \overline{b}(2 * v).\, a(z).\, P\ \textbf{else}\ P\ \mid$
$b(x').\ \textbf{if}\ div(3, x')\ \textbf{then}\ \overline{out}(3 * x').\, \overline{a}(0).\, Q\ \textbf{else}\ \overline{a}(0).\, Q)\backslash\{a, b\})$
$(\textbf{if}\ div(6, v)\ \textbf{then}\ \overline{out}(6 * v).\, Impl\ \textbf{else}\ Impl)$

More transformations can be performed on the left-hand side of the goal by applying the following distributivity laws for the parallel and restriction operators with respect to the two-armed conditional:

OBSEQv_PAR_DISTR_COND_L:
. $\vdash\ \forall V\ Df\ b.\ \forall E\ E'\ E'' ::$ Is_Agentv V.
Obs_Equiv_v V Df
(Par (Cond_Else b E E') E'')
(Cond_Else b (Par E E'') (Par E' E''))

OBSEQv_RESTR_DISTR_COND:
. $\vdash\ \forall V\ Df\ b\ L.\ \forall E\ E' ::$ Is_Agentv V.
Obs_Equiv_v V Df
(Restr (Cond_Else b E E') L)
(Cond_Else b (Restr E L) (Restr E' L))

Both sides of the goal are then rewritten with the theorem that gives the observation semantics for the operator Cond_Else:

OBSEQv_COND_ELSE:
. $\vdash \forall V\ Df\ b.\ \forall E\ E' ::$ Is_Agentv V.
Obs_Equiv_v $V\ Df$ (Cond_Else $b\ E\ E'$) $(b \Rightarrow E \mid E')$

thus getting the following goal (do not confuse the parallel operator '|' with the symbol '|' in the HOL conditional):

$\{\ \exists a\ b.\ \neg(a = b)\ \}$
? Obs_Equiv_v Univ DfH
$(even(v) \Rightarrow$
$(\overline{b}(2 * v).\ a(z).\ P\ \mid$
$b(x').$ **if** $div(3, x')$ **then** $\overline{out}(3 * x').\ \overline{a}(0).\ Q$ **else** $\overline{a}(0).\ Q)\backslash\{a, b\}\ \mid$
$(P\ \mid\ b(x').$ **if** $div(3, x')$ **then** $\overline{out}(3 * x').\ \overline{a}(0).\ Q$ **else** $\overline{a}(0).\ Q)\backslash\{a, b\})$
$(div(6, v) \Rightarrow \overline{out}(6 * v).\ Impl \mid Impl)$

The interaction between the data and behaviour components of the agent specifications now becomes interesting. Both sides of the observation equivalence depend on some condition on the data. The way to proceed is to perform a case analysis on one of the two boolean expressions. Since the HOL proof is carried out in an interactive manner, one can take advantage of this interaction when dealing with data. The idea is to derive new facts about data such that some case analysis and inconsistency checking can be avoided. For example, the two boolean expressions in the current goal are not completely unrelated. It is easy to show that if 6 divides v, then v is an even number, while the implication in the opposite direction is not true. Hence, one can perform a case analysis on the divisibility condition and then avoid a further case split on the even condition, when the boolean expression $div(6, v)$ is assumed to be true. The case split on the divisibility condition generates two subgoals as follows:

$$\{\ \exists a\ b.\ \neg(a = b)\ ;\ div(6, v)\ \}\ ?\ \ldots \tag{4.3}$$

$$\{\ \exists a\ b.\ \neg(a = b)\ ;\ \neg div(6, v)\ \}\ ?\ \ldots \tag{4.4}$$

where the dots '...' stand for the observation equivalence shown in the previous step. Let us prove subgoal (4.3). The condition $even(v)$ is derived from its assumptions by applying some theorems about divisibility and multiplication and using built-in conversions for reducing numbers. The goal is then rewritten with the assumptions, thus becoming as follows:

$\{\ \exists a\ b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ \}$
? Obs_Equiv_v Univ DfH
$((\overline{b}(2 * v).\ a(z).\ P\ \mid$
$b(x').$ **if** $div(3, x')$ **then** $\overline{out}(3 * x').\ \overline{a}(0).\ Q$ **else** $\overline{a}(0).\ Q)\backslash\{a, b\})$
$(\overline{out}(6 * v).\ Impl)$

The two prefixed agents in the left-hand side can synchronise through the common port b and exchange the data $2 * v$, which will replace the value variable x'. The theorem OBSEQv_PAR_OUT_IN_SYNC is applied and two subgoals are produced, one for the proof about the exchanged value expression, and the other for the proof about the process behaviour under the assumption on the data just transmitted:

$$\{\ \exists\, a\, b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ \} \tag{4.5}$$
$$\vdash\ (2 * v) \in \mathsf{Univ}$$

$$\{\ \exists\, a\, b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ \} \tag{4.6}$$
$$\vdash\ \mathsf{Obs\_Equiv\_v\ Univ\ DfH}$$
$$\begin{array}{l}(((\overline{b}(2 * v).\\ \quad (a(z).\, P \mid b(x').\, \mathbf{if}\ div(3, x')\ \mathbf{then}\ \overline{out}(3 * x').\, \overline{a}(0).\, Q\ \mathbf{else}\ \overline{a}(0).\, Q) +\\ \quad b(x).\\ \quad (\overline{b}(2 * v).\, a(z).\, P \mid \mathbf{if}\ div(3, x)\ \mathbf{then}\ \overline{out}(3 * x).\, \overline{a}(0).\, Q\ \mathbf{else}\ \overline{a}(0).\, Q)) +\\ \quad \tau.\\ \quad (a(z).\, P \mid\\ \quad \mathbf{if}\ div(3, 2 * v)\ \mathbf{then}\ \overline{out}(3 * (2 * v)).\, \overline{a}(0).\, Q\ \mathbf{else}\ \overline{a}(0).\, Q))\backslash\{a, b\})\\ (\overline{out}(6 * v).\, Impl)\end{array}$$

Subgoal (4.5) is easily solved as the exchanged data belongs to the value domain Univ. As regards the process subgoal (4.6), note that the application of the law for parallel composition has also introduced the summands associated to the actions that the parallel agents can do without synchronising. These two summands are removed by applying the laws for the restriction operator (as the port b appears in the restriction set) and the laws for deleting occurrences of the inactive agent. In the new goal only the τ-prefixed summand associated to the previous communication is left:

$$\{\ \exists\, a\, b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ \}$$
$$\vdash\ \mathsf{Obs\_Equiv\_v\ Univ\ DfH}$$
$$\begin{array}{l}(\tau.\, ((a(z).\, P \mid \mathbf{if}\ div(3, 2 * v)\ \mathbf{then}\ \overline{out}(3 * (2 * v)).\, \overline{a}(0).\, Q\ \mathbf{else}\ \overline{a}(0).\, Q)\backslash\{a, b\}))\\ (\overline{out}(6 * v).\, Impl)\end{array}$$

The left-hand side contains a silent action which is not matched in the right-hand side. This τ-action can be removed using the law TAU_WEAKv for observation equivalence:

$$\{\ \exists\, a\, b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ \}$$
$$\vdash\ \mathsf{Obs\_Equiv\_v\ Univ\ DfH}$$
$$\begin{array}{l}((a(z).\, P \mid \mathbf{if}\ div(3, 2 * v)\ \mathbf{then}\ \overline{out}(3 * (2 * v)).\, \overline{a}(0).\, Q\ \mathbf{else}\ \overline{a}(0).\, Q)\backslash\{a, b\})\\ (\overline{out}(6 * v).\, Impl)\end{array}$$

The distributivity laws for the parallel and restriction operators are applied again,

thus leading to another case analysis situation:

$$\{\ \exists\, a\ b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ \}$$
$$\vdash\ \mathsf{Obs\_Equiv\_v\ Univ\ DfH}$$
$$(div(3, 2 * v)\ \Rightarrow$$
$$(a(z).P\ \mid\ \overline{out}(3 * (2 * v)).\overline{a}(0).Q)\backslash\{a, b\}\ \mid$$
$$(a(z).P\ \mid\ \overline{a}(0).Q)\backslash\{a, b\})$$
$$(\overline{out}(6 * v).Impl)$$

Once more, the case split on the boolean expression can be avoided by deriving it from the assumptions with some forward reasoning:

$$\{\ \exists\, a\ b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ ;\ div(3, 2 * v)\ \}$$
$$\vdash\ \ldots$$

The newly-derived assumption is used for rewriting the goal. The law for the parallel operator OBSEQv_PAR_IN_OUT_NO_SYNC and the laws for restriction and deletion of inactive summands are applied again, thus getting the goal:

$$\{\ \exists\, a\ b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ ;\ div(3, 2 * v)\ \}$$
$$\vdash\ \mathsf{Obs\_Equiv\_v\ Univ\ DfH}$$
$$(\overline{out}(3 * (2 * v)).((a(z).P\ \mid\ \overline{a}(0).Q)\backslash\{a, b\}))$$
$$(\overline{out}(6 * v).Impl)$$

The substitutivity property OBSEQv_SUBST_OUTPUT for the output operator can be used in a backward manner and two subgoals result from its application, one for the data (the two expressions in the output agents have to be proved equal) and the other for the process behaviour:

$$\{\ \exists\, a\ b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ ;\ div(3, 2 * v)\ \} \quad (4.7)$$
$$\vdash\ 3 * (2 * v) = 6 * v$$

$$\{\ \exists\, a\ b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ ;\ div(3, 2 * v)\ \} \quad (4.8)$$
$$\vdash\ \mathsf{Obs\_Equiv\_v\ Univ\ DfH}\ ((a(z).P\ \mid\ \overline{a}(0).Q)\backslash\{a, b\})\ Impl$$

The data subgoal (4.7) is easily solved by applying associativity of multiplication and using the built-in conversion for evaluating arithmetic expressions. As before, the process subgoal (4.8) is rewritten using the laws for parallel and restriction operators and for inactive summands, and the following goal is obtained:

$$\{\ \exists\, a\ b.\ \neg(a = b)\ ;\ div(6, v)\ ;\ even(v)\ ;\ (2 * v) \in \mathsf{Univ}\ ;\ div(3, 2 * v)\ ;\ 0 \in \mathsf{Univ}\ \}$$
$$\vdash\ \mathsf{Obs\_Equiv\_v\ Univ\ DfH}\ (\tau.((P\ \mid\ Q)\backslash\{a, b\}))\ Impl$$

Note that the assumption about the value 0, exchanged during the communication through the port a, has been added to the assumption list of the goal. This is simply

solved by rewriting with the definition of *Impl* and the law TAU_WEAKv. This solves subgoal (4.3), namely the first subgoal generated by the case split on the divisibility condition. The proof of the second subgoal (4.4) is similar to the one for subgoal (4.3), except that the case analysis on the condition $even(v)$ will have to be performed, as it cannot be derived from the data conditions in the assumptions of the subgoal. This concludes the presentation of the proof of correctness for the communicating system in the HOL-CCS environment.

4.6.2 Discussion

The above verification proof shows how delicate and complex reasoning about communicating systems can be, even in the case of very simple examples. Hennessy's idea [59] of dissociating the reasoning about the data as much as possible from the reasoning about the processes, can be achieved in the HOL-CCS environment by splitting the verification process in two parts, one for the data and one for the process behaviour, any time a proof step is performed which involves dealing with values. This can be due to rewriting with an implicational theorem, whose antecedent is a condition on the values, or to the backward application of an inference rule, whose premisses contain conditions on the values. In these cases, HOL tactics generate distinct subgoals for the data and process components, thus separating the reasoning about the data from the reasoning about the process behaviour. However, the behaviour of an agent depends on the data and, when an implicational theorem is applied, the subgoal for the process behaviour is enriched with a data assumption representing this dependence. In [59, 61] proofs about data are separated from proofs about the process behaviour and may be checked using different and possibly specialised theorem provers. In HOL, both kinds of reasoning can be carried out in the same logical framework, namely the HOL system itself, provided that the system is equipped with the means (i.e. theories) for reasoning about the given value domain, such as the various theorems and conversions about numbers plus related operations and predicates used in the above verification example.

The HOL formalisation of value-passing CCS shows that in HOL it is possible to deal with ω-data-rules and use them in a practical way, even when value-passing agents are defined over an infinite value domain. This is a benefit which derives from using a theorem proving system. In the above proof, the inference rule OBSEQv_OBSCv_INPUT is applied backward but, whenever it is possible to prove a theorem which instantiates its antecedent, an ω-data-rule can also be used as a forward rule. When verifying behavioural equivalences in the HOL-CCS environment, several proof styles can be adopted and mixed together. It is possible to rewrite only one side of a behavioural goal by applying the algebraic laws for the equivalence under consideration, using tactics or conversions working on the left/right-hand side of the behavioural goal. Substitution tactics rewrite a goal with a (possibly implicational)

theorem, which needs to be suitably instantiated, while one or more applications of the laws for a particular operator can be automatically applied by means of conversions. Moreover, inference rules like the substitutivity laws can be used backward to transform the whole goal into one (or more) new subgoal(s). Forward and backward styles can be used together: the main proof is usually backward, but some forward inference starting from the assumptions of the goal and/or other theorems may be useful to derive new facts.

Finally, the above proof is fairly interactive, while Hennessy's proof system aims at automatic proofs as much as possible. In [59] case analysis is automatically performed on conditional agents, the various conditions are accumulated and then checked for (in)consistency. Proofs about data are not expected to be run together with the main proof about the process behaviour. They are instead seen as proof obligations that can be checked by separate and possibly specialised theorem provers. For the moment, verification in HOL is very much interactive. This interaction is exploited for some decision making, e.g. how to perform case analysis and avoid some inconsistency checking by deriving new facts about data, and possibly for detecting errors, thus improving the understanding of the specifications under consideration. When dealing with data, it is not possible in general to achieve fully automatic reasoning, but some steps could be made more automatic and the proof less interactive.

4.7 Summary

This chapter has presented an embedding of value-passing CCS in HOL that translates value-passing agent expressions into pure ones. This mechanisation allows one to reason about both the value-passing calculus and its applications. The translation from the value-passing version to the pure one and the proof of its correctness are examples of the kind of meta-reasoning that can be carried out in HOL. However, users of the proof environment who are only interested in applications, will work directly on value-passing specifications without resorting to the translation, which is only used at meta-level for developing behavioural theories for value-passing agents.

More work and experience in verifying communicating systems in the HOL-CCS environment is needed, in particular when the value domain is more complex. This also involves to derive the general formulation of the expansion law for value-passing agents, as the current expansion laws (like the ones in Section 4.5) are of very restricted forms. Moreover, modal logics have been extended to deal with value-passing agents, as in [63]. It would be interesting to formalise a modal logic for the value-passing calculus in HOL based on the current proof environment for CCS. In this way, another verification method for the correctness of specifications would be provided, as done in the previous chapters for the pure CCS setting. Finally, unique fixpoint induction still needs to be embedded in HOL. This recursion law for a value-

passing calculus has recently been formalised in the tool VPAM based on the theory developed in [62].

Related work about verification systems for value-passing calculi can be found in Section 1.1. In particular, besides the tool VPAM, a symbolic approach is also implemented in LOLA (Section 1.1.1), where a parameterised expansion allows full LOTOS specifications to be analysed by representing the labelled transition systems in a symbolic way. This entails to keep variable definitions as such without expanding them over the value domain and to identify behaviours which are equal except for some value expressions. Before applying the parameterised expansion, some preprocessing is performed on the LOTOS specifications using LOTOS behavioural equations as rewriting rules. The data component of full LOTOS is also treated operationally by applying equations on the data types as rewriting rules and using term rewriting techniques.

Chapter 5

Conclusions and Future Work

This chapter gives a brief summary of the work presented in this dissertation and the results achieved. Extensions and directions for further research are then outlined.

5.1 Summary of Thesis

Process calculi provide formal theories to aid the design and verification phases of concurrent and communicating systems. The same notation can be used both for defining descriptions of a system at different levels of abstraction and for reasoning about them. In the work described in this thesis, the HOL theorem prover has been used as a logical framework for building a practical and sound proof-assistant tool for process algebra specifications. Several components of the theory of a particular process calculus, namely Milner's one presented in [97] and here referred to as CCS, have been embedded in the HOL logic. This work started by considering a version of the pure subset of the CCS calculus with *rec*-notation, inactive agent and binary summation. Next, *rec*-notation has been replaced by agent constants and defining equations and the more general indexed summation operator has been included in the HOL formalisation, so that inactive agent and binary summation could be derived as special cases of indexed summation. In this way, Milner's presentation of the pure calculus as given in [97] was embedded in HOL. Finally, the value-passing version has been formalised based on the mechanisation of pure CCS and the definition of a translation between the two calculi.

No contribution to the formal theory of CCS or similar calculi is given in this dissertation, whose subject is instead an investigation on how this theory can be effectively used for reasoning about concurrent and communicating systems and about the process calculus itself. In fact, the idea is to provide a logical environment in which verification of properties of communicating systems *and* meta-theoretic reasoning can be carried out. As recalled in Chapter 1, a wide collection of verification methods and tools have been developed in the past few years. Many of such tools are much more efficient and easy to use than the one proposed in this dissertation. They are

very good and convenient from the application and user's point of view, but they offer no or few facilities for performing meta-reasoning on the process calculi. The aim of the work presented in this thesis was to provide both facilities based on the HOL proof assistant.

The resulting proof environment contains a collection of conversions, tactics and strategies, which allows one to verify properties of process algebra specifications using powerful techniques, like mathematical induction, within various logics, like modal logic. It is possible to reason about a variety of specifications, such as pure expressions, value-passing agents, parameterised or indexed processes, finite state and infinite state systems. With respect to tools based on finite state machines, it is worth noting that the correctness of infinite state systems can be formally proved. These systems can be pure specifications with an evolving structure, such as the infinite counter in Section 3.2, or value-passing specifications defined over an infinite data domain. In these cases, general and powerful proof techniques are often required, such as induction, contradiction, case analysis, equational reasoning, etc., and incremental or interactive proofs need to be carried out. It is also convenient to define proofs parametrically so that they can be used to deal with a class of processes and/or logical properties. No "real" system has been verified in the HOL-CCS proof environment. Properties of simple example systems taken from the literature about process algebra verification have been formally proved in HOL, but their proofs show how many details need to be taken into account, even when dealing with moderate size examples.

Different approaches to verification can be combined in a theorem proving framework. The behaviour of a system can be explored by running the operational semantics of the calculus. Behavioral equivalences between different specifications can be proved by finding a suitable relation between them (e.g. a bisimulation) or the algebraic theory can be applied by means of equational reasoning to transform the symbolic representation of the specifications. Logical properties that give a partial specification of a system can be checked using the operational semantics. The basic tools for performing verification based on both operational and algebraic/axiomatic methods are provided in the HOL formalisation of the CCS calculus.

The theorem proving approach taken in this dissertation also provides facilities for coping with some of the limitations of the automata based approaches. In fact, reasoning about infinite state specifications can be carried out using the mathematical techniques typically available in a theorem proving framework. Moreover, manipulating the symbolic representation of process specifications gives insight into the specifications themselves and the properties one is trying to verify.

The mechanisation exploits the definition mechanisms and 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. Forward and backward styles are

used together: the main proof is usually backward, but in several occasions some forward inference is performed starting from the assumptions of the goal and/or other theorems to derive new useful facts. This is not only typical of the reasoning about the data component in value-passing specifications, as it has been shown in the verification example in Section 4.6. It is used when checking modal properties in Sections 2.4.2 and 3.2.2 and, actually, this style has also been adopted in the proofs of many of the properties and laws for CCS agents.

Examples of proof techniques for deriving properties about the calculus are structural induction, case analysis and rule induction. They are fundamental tools at the meta-reasoning level for proving theoretical results about the various process calculi, such as properties of agent expressions, transition relations and the correctness of the translation between pure and value-passing CCS. As shown by the HOL formalisation of different versions of the CCS calculus, these proof tools can help investigate variants of a process calculus and develop the formal theory for new calculi.

Embedding the formal theory of a specification language in a theorem prover is very time-consuming, specially when following a purely definitional approach to its formalisation. Before proving many interesting theorems, a lot of infrastructure must be encoded in the logic and every result formally checked. More work than is originally expected can be involved when mechanising definitions or proving properties, because axioms written by hand are often packed with notation which itself needs to be formalised. HOL primitive and derived definition mechanisms, the built-in theories, all theorems and tactics provided by the induction definition package greatly aid the formalisation task, but still it is a lot of (often tedious) work. However, it is worth noting that the resulting HOL formalisation for the various notions is very similar to their conventional presentation, thus demonstrating the suitability of HOL for supporting other notations. For example, the formalisation of the algebraic laws for the behavioural equivalences illustrates their similarity to the mathematical presentation.

5.2 Development and Some Measures of the Work

The research described in this thesis started with a non-definitional formalisation of pure CCS in HOL. The idea was to understand whether the HOL system could provide a suitable infrastructure for developing a proof environment for process algebras. The verification examples considered in [23, 100] showed the suitability of the HOL proof assistant for this purpose. Thus, the task of formalising the theory of pure CCS in a purely definitional way was then undertaken. This resulted in the work described in [101, 102] and in a preliminary version of Chapter 2.

An unsatisfactory treatment of inductively defined agents in [100, 101] (i.e. the use of the function `new_axiom` [51]) was recovered by introducing (parameterised) agent constants and defining equations in the HOL mechanisation of pure CCS. The

HOL proofs of correctness of the infinite counter were revised based on the new formalisation, thus yielding a preliminary draft of the work illustrated in Sections 3.1 and 3.2.

The extension to value-passing CCS based on the translation into the HOL embedding of pure CCS took a considerable time, specially the formalisation of the indexed summation operator and the input prefix operator. A preliminary version of Sections 3.3, 4.1 and 4.2 was presented in [103]. The research continued by deriving the behavioural theories for value-passing CCS and by revisiting the expansion law for pure CCS, thus completing the study presented in Chapters 3 and 4.

An important feature missing in all this work was the distinction between agents and agent expressions, in the sense that all CCS terms were assumed to be agents. At that time the interest was mainly in reasoning about CCS applications, and agents were always considered in verification examples. The need for the distinction between open and closed agent expressions arose later, when proving meta-theorems about CCS recursive agents. This led to a final revision of the theories and examples developed in the dissertation with the introduction of the predicate Is_Agent and the use of restricted quantification.

In HOL88 only a subset of basic conversions, rules and functions is available in the restricted quantified version [125]. The restricted counterparts of more rules, functions and definition mechanisms were necessary when formalising the CCS theory in HOL. For this purpose a few rules and tactics have been defined for dealing with restricted quantified variables directly. However, when doing proofs involving restricted quantification, the conversions for transforming a term with restricted quantification into one without it, have often been used. As far as the definition mechanisms are concerned, the approach was mostly to define the new notion by means of predicates and the usual definition mechanisms, and then derive the desired formulation with restricted quantification. A wider collection of tools for dealing with restricted quantification, and in general conditional rewriting, has been developed in hol90 [119].

This section ends with some information about the "size" of various theories of the HOL formalisation of the CCS calculus. The measure is given through (i) the number of lines of ML code, (ii) the number of HOL definitions (e.g. type, constant and function definitions) and theorems which have been proved, and (iii) the number of primitive inferences generated when building the theories.

The ML code for the mechanisation of the syntax of pure CCS (Section 2.2.1) is about 500 lines (including that of auxiliary functions and conversions) and contains 19 definitions and 44 theorems (7 of which are auxiliary theorems on sets, that were not provided in the theory `sets`). The theories for the pure syntax generate 19094 primitive inferences. The ML source of the theories for the operational semantics (Section 2.2.2) is about 500 lines and includes 1 definition, namely that of the transition relation, and 44 theorems. The primitive inferences generated when building these theories are 17144.

The ML code for the strong semantics is 756 lines. Most of this code is for the proofs of substitutivity of the CCS operators: 4 definitions are introduced in HOL and 22 theorems are proved with 11768 primitive inferences. The formalisation of the notation for the expansion law and its properties (Section 2.2.4) occupies 372 lines with 6 definitions and 18 theorems. The proof of the strong version of the expansion law is 186 lines and generates 3609 primitive inferences. The same law for observation congruence is derived with a proof of just 44 lines of ML code and 556 primitive inferences. In the proof of the reader-write problem, one application of the tactic `OC_EXP_THM_TAC` for the expansion law generates between 5000 and 7800 primitive inferences. When moving to the polymorphic CCS type with agent constants and indexed summations, the strong version of the law COUNTABLE_PAR_LAW (Section 3.4) is 252 lines of ML code and is proved by means of 3521 primitive inferences.

As far as the HOL theories of value-passing CCS are concerned, the ML source for the syntax is about 600 lines with 14 definitions and 27 theorems. The primitive inferences produced are 134201. The ML code for the translations of value-passing labels, actions and agent expressions is about 200 lines, consists of 5 definitions and 12 theorems, and generates 6023 primitive inferences.

5.3 Future Work

The HOL-CCS proof environment can be extended, combined with other tools, made more practical, efficient and user-friendly. The work presented in this thesis has laid the fundamentals for reasoning about process specifications in a formal way using the theorem proving technology. Future research will consider the following issues: more automation in verification proofs, combination of the HOL-CCS logical framework with an automata based tool, extensions to the process calculi, semantics and logics which have been addressed in this dissertation.

So far, verification proofs in the HOL-CCS environment are pretty much interactive, but several steps could be automated. For example, the application of the algebraic laws, typically the expansion theorem, could be scaled up so that several expansions are performed in one step instead of a single expansion at a time. The basic steps implemented by the current versions of the various tactics can be combined together and more powerful tactics can be obtained. This has already been done up to a certain extent and the tactic for the expansion theorem is itself an example of a "bigger" tactic which puts together the tactics for relabelling, restriction and parallel composition. Likewise, conversions and tactics like the one for the restriction operator are able to apply several laws for restriction in one step (Section 2.3.1). The same tactic could be modified in such a way that all possible rewritings with the laws for that operator are applied, so that the resulting term is in a reduced form with respect to those laws. During the verification phase, understanding of the manipu-

lation being applied, the rewriting steps, their outcome, etc. is a major factor, thus tactics for basic steps or a simple combination of basic steps have been implemented. Nevertheless, they can be made more automatic and several steps can be performed by only one tactic, possibly with the help of other tools, such as a new simplifier using ideas from other systems, e.g. Isabelle [108, 119].

The HOL formalisation of process calculi can be combined with a tool based on a different approach to verification, such as a finite state machine tool (see Section 1.1.1 for related work about combining model checking and theorem proving). A theorem prover allows properties to be verified in a rigorous and formal way, while automata based tools are very efficient and can deal with very large specifications. The verification process could be run on a finite state machine tool, whenever possible, and then the theorem prover could be used to check the result formally or to help detecting errors in specifications by means of interactive proofs. In particular, a combination of the theorem prover Isabelle and the SMV system is currently being investigated by Paulson [109] with the more general aim of integrating "oracles", namely external and independent reasoning tools, to help verifying temporal properties of system specifications.

Extensions to the process calculi, the behavioural semantics and the logics considered in the thesis can be easily embedded in the HOL system. The theorem proving methodology has indeed the advantage of being flexible and open-ended, in the sense that the formal theory of different process calculi can be represented in the HOL logic, various behavioural semantics and logics for a given process calculus can be formalised and proof tools can be soundly derived.

The theory of process calculi is still developing in many directions, such as process mobility (π-calculus), time, probabilities, non-interleaving semantics. In this thesis the well-studied core of the theory for process algebra, made of interleaving semantics, pure calculi, modal logics, has been embedded in HOL and then extended to a value-passing calculus. Probabilistic and real-time process algebras could be mechanised in HOL. Behavioural semantics could be enriched with the formalisation of preorder and refinement relations, besides equivalences. An interesting extension to the work done so far would be investigating the formalisation of algebraic specification languages in which data are given through abstract data types, e.g. LOTOS. As the process component of the LOTOS formalism is close to the CCS algebra, it might be possible to derive proof techniques for LOTOS specifications from the HOL-CCS environment based on the translation from Basic LOTOS to CCS given in [13]. This work would parallel and could be compared with the research being carried out using the PSF tool and the theorem prover Coq for the specification language μCRL (Section 1.1.2).

The modal logic considered in this dissertation can be extended to the value-passing calculus by adapting the work in [63]. More expressive temporal logic [118, 16] can be represented in higher order logic and proof tools, e.g. the tableau system (extended to deal with infinite state processes in [17]), can be soundly mechanised.

The tableau system decision procedure has been implemented in some verification tools, e.g. the Concurrency Workbench. Such a technique can also be naturally described as a goal directed proof system and, as such, is amenable to be formalised in a theorem proving system which provides goal directed proofs. 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.

Appendix A

The HOL System

This appendix contains a brief introduction to the HOL proof assistant. The aim is to present those notions of higher order logic, definition mechanisms and theorem proving infrastructure, which have been used in the formalisation of process calculi in HOL. A detailed description of the HOL system can be found in [51]. The version of the system used in this dissertation is the original one, referred to as HOL88, where the so-called 'Classic ML' (an early version of ML derived from LCF [32]) is adopted as the meta-language.

A.1 The Meta-Language ML

ML is an interactive programming language. At top level one can evaluate expressions and perform declarations. The result of evaluating an expression is its value and type being printed; making a declaration results in a value being bound to a name.

In what follows, HOL sessions will be displayed through boxes in sequence, each box representing some interaction steps with the HOL system via the ML language. The ML prompt is #, thus lines beginning with # show text typed by the user, which is always terminated by a double semicolon ';;'. The other lines in the boxes show the system's response.

```
#‘hello!‘;;
‘hello!‘ : string
```

The ML expression `‘hello!‘` is a string, i.e. a sequence of characters enclosed between string quotes. Its type is `string`.

A declaration `let` x = e evaluates the expression e and binds the resulting value to the name x. The value of the last expression evaluated is always recorded in the variable `it` (note that declarations do not affect the value of `it`):

```
#let x = 3 * 4;;
x = 12 : int

#it;;
'hello!' : string
```

The general form of declaration `let` $x_1 = e_1$ `and` $\cdots$ `and` $x_n = e_n$ results in binding the value of each expression e_i to the name x_i. A declaration d can be made local to the evaluation of an expression e by evaluating the expression d `in` e.

```
#let x = 3 and y = 4 in x * y;;
12 : int

#it;;
12 : int
```

A declaration `let` $f\ x = e$ defines a function f with formal parameter x and body e.

```
#let f x = x * x;;
f = - : (int -> int)

#f 4;;
16 : int
```

Functions are printed as a dash followed by their type, because a function as such is not printable. The application of a function f to an argument x can be written as $f\,x$, even though the notation $f(x)$ is also allowed. Functions can also be curried and partially applied.

```
#let f x y = x * y;;
f = - : (int -> int -> int)

#let g = f 3;;
g = - : (int -> int)

#g 4;;
12 : int
```

Functions can be written as λ-abstractions. The expression `\`x`.`e evaluates to a function with formal parameter x and body e. The declaration `let` f `=` `\`x`.`e is thus equivalent to `let` $f\ x$ `=` e.

```
#\x. x * x;;
- : (int -> int)

#it 4;;
16 : int
```

The ML type checker is able to infer the type of expressions if there is enough information.

```
#[3; 4];;
[3; 4] : int list

#tl it;;
[4] : int list
```

The ML expression `[3; 4]` is a list of integers and its type is `int list`, where `list` is a unary type constructor. The function `tl` takes a non-empty list and returns the tail of that list.

ML types can be *polymorphic*, i.e. they can contain type variables (denoted by `*`, `**`, etc. in HOL notation). The type `list` is indeed polymorphic and so are functions operating on lists, e.g. `tl: * list -> * list`. In the above example, the type variable `*` is instantiated to type `int`, thus getting the particular type `int list`.

Besides `list`, type constructors include `#` for pairs (product type) and `+` for the disjoint union of types. These are both binary type constructors, `* # **` and `* + **`, and are provided with primitive functions for performing typical operations on such types. Among others are the ML functions `fst: (* # **) -> *` and `snd: (* # **) -> **` for extracting the first and second components of a pair, and `inl: * -> (* + **)` and `outl: (* + **) -> *` for injecting and projecting the left summand of a disjoint union.

A.2 Higher Order Logic

The formulation of higher order logic in HOL is based on an extension of Church's simple theory of types [26]. The standard predicate calculus is extended in the HOL logic by allowing variables to range over functions and predicates, and arguments of functions can themselves be functions (hence higher order). Moreover, functions can be written as λ-abstractions and terms of the HOL logic can be polymorphic (with type variables ranged over by $\alpha, \beta, \gamma, \ldots$ represented in HOL by `*`, `**`, ... as mentioned above).

Terms of the HOL logic (object-language terms) are represented in ML by an abstract type called `term` and they are distinguished from ML expressions by enclosing them in double quotes. Terms can be manipulated through various built-in ML

functions. For example, given the expression `x \/ y`, which in ML evaluates to a term representing the disjunction $x \vee y$, the function `dest_disj: term -> (term # term)` splits the disjunction into the pair of its two disjuncts:

```
#"x \/ y";;
"x \/ y" : term

#dest_disj it;;
("x", "y") : (term # term)
```

The types of terms of the HOL logic are similar to those of ML expressions. The ML type called `type` represents the types of HOL terms, which are expressions of the form `": ··· "`. The built-in function `type_of` returns the logical type of a term.

```
#(3,4);;
(3, 4) : (int # int)

#"(3,4)";;
"3,4" : term

#type_of it;;
":num # num" : type
```

There are four kinds of terms: variables, constants, function applications and λ-abstractions. A λ-term `\`$x.t$ denotes a function $v \mapsto t[v/x]$, where $t[v/x]$ denotes the result of substituting v for x in the term t.

```
#"\x. x * x";;
"\x. x * x" : term

#type_of it;;
":num -> num" : type
```

The type checking algorithm tries to infer the type of HOL terms using the types of constants and operators that occur in the same quotation. Above, the type of the multiplication operator `":num -> (num -> num)"` is used to determine the type of the function denoted by the λ-term. If there is not enough type information, a type checking error results and the user has to provide some more type information explicitly.

In HOL conditional expressions are represented by `"`b `=>` t_1 `|` t_2`"` (pretty-printed as $b \Rightarrow t_1 \mid t_2$) with the obvious meaning 'if b then t_1 else t_2':

```
#"(x = (y + 1)) => x | y";;
"((x = y + 1) => x | y)" : term

#type_of it;;
":num" : type
```

The ML language is used to manipulate terms of the HOL logic. In particular, ML is used to prove that certain terms are *theorems*. Theorems of the logic are also represented by abstract data types, and the user can work on them only through ML functions. The ML type for theorems is `thm`. 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, $\Gamma \vdash t$ (or simply $\vdash t$, if Γ is empty) denotes the corresponding theorem. In order to introduce theorems into the logic, they must either be postulated as axioms or derived from existing theorems and definitions by formal proof.

A *theory* is a collection of logical types, type operators, constants, definitions, axioms and theorems. Theories enable a hierarchical organisation of facts, i.e. if facts from other theories need to be used, the relevant theories must be declared as *parents*. There are several built-in theories in HOL. Examples are the theories `bool`, `sum`, `one`, `prim_rec` and `sets`.

The theory `bool` contains the definitions of various constants and type operators, such as the usual logical constants of the predicate calculus `T`, `F`, `~`, `/\`, `\/`, `==>`, `=`, `!`, `?` and `?!` to represent true T, false F, negation $\neg$, conjunction $\wedge$, disjunction $\vee$, implication $\supset$, equality $=$, universal quantification $\forall$, existential quantification $\exists$ and unique existential quantification $\exists!$, respectively. Hilbert's choice operator ε (written `@` in HOL notation) is another primitive constant defined in the theory `bool`. If P has type $\alpha \to \mathit{bool}$, then the term $\varepsilon x.\, P\ x$ denotes some element of the set whose characteristic function is P. If the set is empty, then $\varepsilon x.\, P\ x$ denotes an arbitrary element of the set denoted by α. This implies that all logical types must denote non-empty sets, since for any type α, the term $\varepsilon x : \alpha.\, \mathsf{T}$ represents an element of the set denoted by α. The product type $\alpha \times \beta$ for ordered pairs is also defined in the theory `bool` together with its constants, such as $\mathsf{Fst} : \alpha \times \beta \to \alpha$ and $\mathsf{Snd} : \alpha \times \beta \to \beta$ for selecting the first and second components of pairs, and the theorems describing such operations.

The theory `bool` also includes the definitions of constants for dealing with *restricted quantification*. For example, the semantics for restricted universal and existential quantification is given by the constants Res_Forall and Res_Exists defined as follows:

$$\forall P\ t.\ \mathsf{Res\_Forall}\ P\ t = (\forall x : \alpha.\ P\ x\ \supset\ t\ x)$$

$$\forall P\ t.\ \mathsf{Res\_Exists}\ P\ t = (\exists x : \alpha.\ P\ x\ \wedge\ t\ x)$$

Given a quantifier Q, a variable $x : \alpha$ and any predicate $P : \alpha \to \mathit{bool}$, the HOL parser

and pretty-printer allow one to write terms of the form $\mathsf{Q}\, x :: P.\, t[x]$ to denote the (restricted) quantification of x over those values of type α which satisfy P. The notation '::' can be used with abstraction and any binder, including user defined ones. A few theorems about the properties of restricted universal and existential quantifiers are provided in the theory `res_quan` developed by Wong [125].

The built-in theory `sum` contains the definition of the type operator for the (binary) disjoint union $\alpha + \beta$ of types together with the associated constants, such as the ones for injecting and projecting left and right summands

$$\mathsf{Inl} : \alpha \to \alpha + \beta \qquad \mathsf{Outl} : \alpha + \beta \to \alpha$$
$$\mathsf{Inr} : \beta \to \alpha + \beta \qquad \mathsf{Outr} : \alpha + \beta \to \beta$$

and those for testing membership of the left and right summands

$$\mathsf{Isl} : \alpha + \beta \to \mathit{bool}$$
$$\mathsf{Isr} : \alpha + \beta \to \mathit{bool}$$

Several theorems about the properties of these constants are built-in in the theory `sum`. Some of them are used in the proofs presented in this thesis, such as

$$\text{OUTL:} \quad \vdash \forall x.\, \mathsf{Outl}\,(\mathsf{Inl}\, x) = x$$
$$\text{ISL:} \quad \vdash (\forall x.\, \mathsf{Isl}\,(\mathsf{Inl}\, x)) \;\wedge\; (\forall y.\, \neg\, \mathsf{Isl}\,(\mathsf{Inr}\, y))$$

The theory `one` defines the type *one* which contains exactly one element. This element is denoted by the constant `one` and is defined as $\varepsilon x : \mathit{one}.\, \mathsf{T}$. Among the theorems which characterise the type *one* is the theorem $\vdash \forall v : \mathit{one}.\, v = \mathsf{one}$ asserting that there is only one value of type *one*.

Constants, definitions and theorems concerning primitive recursive functions are given in the theory `prim_rec`. The following is the basic theorem `num_Axiom`

$$\vdash \forall e\, f.\, \exists!\, \mathit{fn}.\, (\mathit{fn}\, 0 = e) \;\wedge\; (\forall n.\, \mathit{fn}\,(n+1) = f\,(\mathit{fn}\, n)\, n)$$

stating the validity of primitive recursive definitions on the natural numbers. This means that for any e and f there exists a unique total function *fn* which satisfies the primitive recursive definition whose form is determined by e and f.

The theory `sets` contains constants, definitions and theorems about finite and infinite sets. The basis is the polymorphic type $(\alpha)\mathit{set}$ which is just an object-language abbreviation for the type $\alpha \to \mathit{bool}$. In fact, a set is represented by its characteristic function and the elements of a set $s : (\alpha)\mathit{set}$ are just those values of type α for which the corresponding predicate is true. Generalised set specifications are also supported, that is for any expression $E[x]$ and predicate $P[x]$, $\{E[x] : P[x]\}$ (represented in HOL by `{E[x] | P[x]}`) is the set of all values $E[x]$ for which $P[x]$ holds. The theory `sets` provides a wide collection of theorems about the various operations on sets and their properties.

Rules of definition are included in the HOL logic for extending theories in a purely definitional way. This is done by defining new constants and types in terms of properties of existing ones. This feature extends Church's formulation and plays a fundamental role in the formalisation described in this dissertation. The rules of definition are briefly presented in the following sections.

A.3 Primitive Rules of Definition

The primitive basis of the HOL logic includes three rules of definition for extending the logic in a sound way.

The rule of *constant definition* allows one to introduce a new constant c as an object-language abbreviation for a closed (no free variables) term t. This is achieved by defining an equational axiom $\vdash \mathsf{c} = t$. Among the various properties that the constant c must satisfy is the condition that c may not occur in the term t. Thus, recursive definitions cannot be introduced into the logic using the rule of constant definition. The functions `new_definition` and `new_infix_definition` are some of the ML functions implementing the rule of constant definition. Constants whose arguments can be restricted quantified variables are defined through the ML function `new_resq_definition`.

Given a theorem of the form $\vdash \exists x_1 \ldots x_n.P[x_1, \ldots, x_n]$, the rule of *constant specification* (implemented by the ML function `new_specification`) allows one to give a name to existing values $x_1, \ldots, x_n$ for which $P[x_1, \ldots, x_n]$ holds. This is obtained by introducing new constants $\mathsf{c}_1, \ldots, \mathsf{c}_n$ and deriving the theorem $\vdash P[\mathsf{c}_1, \ldots, \mathsf{c}_n]$.

The rule of *type definition* (provided by the ML function `new_type_definition`) allows one to define a new type constant or type operator. Given a type α, let $P : \alpha \to \mathit{bool}$ be the characteristic function of some (non-empty) subset of the set denoted by α. From the theorem $\vdash \exists x : \alpha.\, P\, x$, the rule of type definition derives the existence of a bijection from the elements of a new type β to the subset of elements of type α that satisfy P, as asserted by the theorem

$$\vdash \exists f : \beta \to \alpha.\ (\forall x\ y.\ (f x = f y) \supset (x = y)) \wedge (\forall x.\ P x = (\exists y.\ x = f y))$$

This theorem introduces the new type β to name the non-empty subset of elements of type α that satisfy the predicate P. Functions to denote the above bijection and its inverse can be defined. A *representation* function $\mathsf{REP}_\beta : \beta \to \alpha$ maps a value of the new type β into the value of type α which represents it. The *abstraction* function $\mathsf{ABS}_\beta : \alpha \to \beta$ maps a representation of type α to its abstract value of type β. ABS_β is the left inverse of REP_β and, for those elements of type α that satisfy P, REP_β is the left inverse of ABS_β:

$$\vdash \forall a.\ \mathsf{ABS}_\beta\ (\mathsf{REP}_\beta\ a) = a$$
$$\vdash \forall r.\ P\ r = (\mathsf{REP}_\beta\ (\mathsf{ABS}_\beta\ r) = r)$$

These mappings allow one to define operations on the values of the new type β in terms of operations on values of the representing type α.

A.4 Derived Rules of Definition

The primitive rules of definition 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 formalisations. However, several derived rules of definition have been mechanised 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 [91, 92].

The derived rule of *recursive type definition* (`define_type` in ML) allows one to define arbitrary concrete recursive types in terms of their constructors [91]. 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:

$$(\alpha_1,\ldots,\alpha_n)rty \ ::= \ \mathsf{C}_1\, ty_1^1 \ldots ty_1^{k_1} \ \mid \ \ldots \ \mid \ \mathsf{C}_m\, ty_m^1 \ldots ty_m^{k_m}$$

where $\mathsf{C}_1,\ldots,\mathsf{C}_m$ $(m \geq 1)$ are distinct constructors, each taking k_i arguments $(k_i \geq 0)$, and each ty_i^j is either the recursive type rty or an existing logical type (not containing rty). If one or more of the ty_i^j is rty, then the type specification defines a recursive type. Non-recursive types defined through this type definition mechanism are just special cases. If the type being defined is recursive, at least one constructor must be non-recursive, i.e. the type of all its arguments may not be rty. The type $(\alpha_1,\ldots,\alpha_n)rty$ is polymorphic in the type variables $\alpha_1,\ldots,\alpha_n$ if $n \geq 1$; if $n = 0$ then rty is a type constant.

The above type specification denotes the set of all expressions which can be finitely generated using the constructors $\mathsf{C}_1,\ldots,\mathsf{C}_m$, which are distinct and one-to-one. Given such a type specification, the rule of recursive type definition performs all the formal inference necessary to define the type in higher order logic and derives an abstract characterisation of the type $(\alpha_1,\ldots,\alpha_n)rty$ in a fully automatic way. This characterisation is a theorem of higher order logic asserting that there exists a unique function which satisfies the primitive recursion defined by the type specification for $(\alpha_1,\ldots,\alpha_n)rty$. The recursive type definition package also provides functions which automatically prove that the type constructors are distinct and one-to-one and derive theorems for structural induction and case analysis.

The derived mechanism of *primitive recursive function definition* (provided by the ML function `new_recursive_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.

Shepherd [115] has extended the recursive type definition package in HOL88 to deal with specifications of the form

$$(\ldots, \alpha, \ldots)ty = \ldots \mid \mathsf{C}_i \ldots (\alpha \rightarrow ty) \ldots \mid \ldots$$

where function types occur as the type of some arguments of some constructors, provided that the type ty does not occur in α [55]. The idea is to represent the recursive type by a pair of types, where the elements of type $\alpha \rightarrow ty$ are encoded as a set of pairs which define the function. The inductive definition package (see below) is then used to introduce a relation which plays the role of the uniquely defined function in a recursive type definition. Hence, this relation is shown to induce such a unique function (i.e. each element has exactly one target in the relation). Functions for proving that the type constructors are distinct and one-to-one, plus theorems for structural induction and case analysis, are provided similarly to the ones in the built-in type definition package. The mechanism of primitive recursive function definition is extended as well.

The derived rule for *inductive definitions* (`new_inductive_definition` in ML) allows one to define relations which are inductively defined by a set of rules [92]. Let R be an n-place relation defined through a set of rules of the form

$$\frac{R(t_1^1, \ldots, t_n^1) \quad \ldots \quad R(t_1^k, \ldots, t_n^k)}{R(t_1, \ldots, t_n)} \qquad c_1 \ldots c_r$$

where the terms $R(t_1^i, \ldots, t_n^i)$ for $1 \leq i \leq k$ are the premisses of the rule, $c_1 \ldots c_r$ are the side conditions (not involving the relation R being defined) and $R(t_1, \ldots, t_n)$ is the conclusion of the rule. The relation R is closed under the above rule if the conclusion is true whenever the premisses and the side conditions are true. The relation R is inductively defined by a set of such rules if R is the least relation closed under all the rules.

In the inductive definition package, any such relation is simply defined as the intersection of all relations closed under a given 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 this definition mechanism constitute a complete characterisation of the properties of the newly-defined relation. They include a list of theorems (one for each rule) which assert that the relation satisfies those rules, and a theorem which states a principle of *rule induction* for the relation. Given the above relation R, rule induction allows one to prove that every element in R has a property P, i.e. $R(x_1, \ldots, x_n)$ implies $P(x_1, \ldots, x_n)$, by simply proving that the relation $\{(x_1, \ldots, x_n) \mid P(x_1, \ldots, x_n)\}$ is closed under the rules that define R. This is due to the fact that R is the least relation closed under the same rules. The theorem of rule induction allows proofs by induction to be performed over the structure of the derivations defined by the set of rules. Furthermore, the inductive

definition package provides a theorem for performing exhaustive case analysis over the inductively defined relation.

The rule for inductive definitions also enables one to formalise *classes* of inductively defined relations. Let R be an n-place curried function to be defined and $v_1, \ldots, v_n$ be n distinct variables. Let $v_{j_1}, \ldots, v_{j_s} (1 \leq j_k \leq n, 1 \leq k \leq s)$ be those variables that denote the parameters of the class of inductively defined relations. R can be seen as a function which maps the parameters $v_{j_1}, \ldots, v_{j_s}$ to inductively defined relations, one for each value of $v_{j_1}, \ldots, v_{j_s}$, thus representing a class of inductively defined relations.

A.5 Proofs in HOL

Theories are extended in a sound way by deriving new theorems by formal proof. To prove a theorem in a theory, one must apply a sequence of proof steps to either axioms or previously proved theorems using ML programs called *inference rules* (forward proof). The core of the deductive system in HOL is made up of eight primitive inference rules, from which all other rules are derived. Among the primitive inference rules is β-conversion, implemented by the ML function `BETA_CONV`, which reduces a β-redex $(\lambda x.\, u)\, v$ to the term $u[v/x]$ by returning the theorem $\vdash (\lambda x.\, u)\, v = u[v/x]$ (by renaming variables where necessary to avoid free variable capture).

```
#BETA_CONV "(\x y. x + y)y";;
|- (\x y. x + y)y = (\y'. y + y')
```

In fact, *conversions* are ML functions that map a term t to a theorem $\vdash t = u$ expressing the equality of t with some other term u [106]. Inference rules typically transform theorems into other theorems, e.g. the rule `BETA_RULE` which, given a theorem $\Gamma \vdash t$, returns the theorem resulting from reducing all the β-redexes, at any depth, in the conclusion t.

```
#let thm = ASSUME "f = ((\x y. x + y) y)";;
thm = f = (\x y. x + y)y |- f = (\x y. x + y)y

#BETA_RULE thm;;
f = (\x y. x + y)y |- f = (\y'. y + y')
```

`BETA_RULE` is defined in terms of `BETA_CONV` using the ML functions `DEPTH_CONV`, that applies a conversion repeatedly to all subterms in a bottom-up order, and `CONV_RULE`, which transforms a conversion into an inference rule.

Among the derived inference rules is the basic rewriting rule `REWRITE_RULE` which, given a list of equational theorems (namely theorems whose conclusion is of the form $t = u$) and a theorem *thm*, replaces any subterm in *thm* that matches the left-hand

side of any of the theorems in the list by the corresponding instance of the right-hand side.

```
#MULT_0;;
|- !m. m * 0 = 0

#MULT_ASSOC;;
|- !m n p. m * (n * p) = (m * n) * p

#REWRITE_RULE [MULT_0] (SPECL ["1"; "0"; "0"] MULT_ASSOC);;
|- T
```

Note how the theorem MULT_ASSOC for associativity of multiplication has been 'specialised' using the inference rule SPECL. Given a list of terms and a theorem, SPECL instantiates (some of) the universally quantified variables in the conclusion of the theorem (by renaming variables where necessary to avoid free variable capture). When specialising only one variable, the rule SPEC can also be used. Restricted universally quantified variables can be instantiated using the inference rules RESQ_SPECL and RESQ_SPEC. Given a term list $[u_1; \ldots; u_n]$ and a theorem $\Gamma \vdash \forall x_1 :: P_1. \ \ldots \forall x_n :: P_n. t$, the rule RESQ_SPECL returns the theorem Γ, $P_1\, u_1, \ldots, P_n\, u_n \vdash t\,[u_1/x_1] \ldots [u_n/x_n]$.

Some built-in basic tautologies are also implicitly used by REWRITE_RULE. The search for subterms to be replaced is performed top-down and recursively, until no more replacements can be done. This may lead to unwanted and/or unnecessary reductions or even non-termination of the rewriting process. Other versions of the rewriting rule, such as ONCE_REWRITE_RULE that rewrites subterms once, and substitution rules, such as SUBST that replaces selected subterms, may be used instead.

```
#ONCE_REWRITE_RULE [MULT_0] (SPECL ["1"; "0"; "0"] MULT_ASSOC);;
|- 1 * 0 = 0
```

Besides forward proofs, 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. The HOL system provides a *subgoal package* due to Paulson [107], which implements a simple framework for interactive proofs. A goal given by an assumption list Γ and a term t, written Γ ?- t (if Γ is empty, the goal is written ?- t), can be set by invoking either the function set_goal or the function g (an abbreviation of set_goal whenever Γ is empty), which initialises the subgoal package with a new goal.

As an example, let us prove the above theorem MULT_0:

```
#set_goal([],"!m. (m * 0 = 0)");;
"!m. m * 0 = 0"
```

The current goal can be expanded using the function expand (written e for short) which applies a tactic to the top goal on the stack and pushes the resulting subgoals onto the goal stack. By applying mathematical induction with the built-in tactic INDUCT_TAC, two subgoals corresponding to the basis case (the subgoal at the bottom of the subgoal list) and to the induction step are generated:

```
#e INDUCT_TAC;;
OK..
2 subgoals
"(SUC m) * 0 = 0"
    [ "m * 0 = 0" ]

"0 * 0 = 0"
```

Tactics corresponding to conversions and inference rules are defined in HOL. For example, rewriting tactics such as REWRITE_TAC and ASM_REWRITE_TAC, which adds the assumptions of the goal to the given list of theorems, are fundamental in goal directed proofs. The basis case is solved by rewriting with the definition of multiplication:

```
#MULT;;
|- (!n. 0 * n = 0) /\ (!m n. (SUC m) * n = (m * n) + n)

#e (REWRITE_TAC [MULT]);;
OK..
goal proved
|- 0 * 0 = 0

Previous subproof:
"(SUC m) * 0 = 0"
    [ "m * 0 = 0" ]
```

When a tactic solves a subgoal, the package computes a part of the proof and presents the user with the next subgoal. The definition of multiplication is used once more to transform the induction subgoal:

```
#e (REWRITE_TAC [MULT]);;
OK..
"(m * 0) + 0 = 0"
  1  ["m * 0 = 0" ]
```

The zero summand can be deleted by applying properties of addition (given in HOL by the theorem ADD_CLAUSES):

```
#e (REWRITE_TAC [ADD_CLAUSES]);;
OK..
"m * 0 = 0"
  1  ["m * 0 = 0" ]
```

The induction subgoal is thus reduced to the inductive hypothesis and simply solved by rewriting with it:

```
#e (ASM_REWRITE_TAC[]);;
OK..
goal proved
. |- m * 0 = 0
. |- (m * 0) + 0 = 0
. |- (SUC m) * 0 = 0
|- !m. m * 0 = 0

Previous subproof:
goal proved
() : void
```

Conversions can be mapped into tactics using the ML function `CONV_TAC`, such as the tactic `BETA_TAC` (defined in terms of `BETA_CONV`) for applying β-conversion to the conclusion of a goal.

```
#g "(\x. x + 1)1 = SUC 1";;
"(\x. x + 1)1 = SUC 1"

#e BETA_TAC;;
OK..
"1 + 1 = SUC 1"
```

Tactics can be composed using other ML functions called *tacticals*. For example, tactics can be sequenced by means of the (infixed) tactical `THEN`: given tactics T_1 and T_2, the ML expression T_1 `THEN` T_2 evaluates to a tactic that first applies T_1 and then applies T_2 to each subgoal produced by T_1. Another sequencing tactical is `THENL`: given a tactic T that generates n subgoals and a tactic list $[T_1;\ldots;T_n]$, then the tactic T `THENL` $[T_1,\ldots,T_n]$ first applies T and then applies T_i to the ith subgoal produced by T.

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 $?\vdash t$ by applying *tac*. If it succeeds, the resulting theorem is saved under

the name s in the current theory. For example, the above goal about multiplication is proved by the following tactic and stored under the name MULT_0:

```
#let MULT_0 =
#  prove_thm
#   (`MULT_0`,
#    "!m. (m * 0 = 0)",
#    INDUCT_TAC THEN ASM_REWRITE_TAC [MULT;ADD_CLAUSES]);;
MULT_0 = |- !m. m * 0 = 0
```

A mixed approach of backward and forward proofs is also possible and can sometimes be convenient. When developing a tactic in a goal directed proof, it may be useful to derive new theorems from the assumptions and/or other theorems in a forward manner, and then use these new facts to manipulate the goal.

Appendix B

The Reader-Writer System

The interactive HOL sessions in this appendix show how the correctness of the reader-writer system (Section 2.4) can be formally verified using the proof environment for the pure CCS calculus developed in Chapter 2. This correctness result has been achieved in two ways. First, an implementation of the reader-writer system has been proved to be observation congruent to its abstract specification. Appendix B.1 presents the interactive HOL session(s) for the backward proof(s) given in Section 2.4.1. Second, the same implementation has been checked against a modal property that provides a partial specification of the system. Excerpts of the HOL code for the proof in Section 2.4.2 are shown in Appendix B.2.

Common to the two verification methods is the implementation *Impl* of the reader-writer system. Agents are introduced in HOL by invoking the ML function `new_definition` for constant definitions (Section A.3). The semaphore *Sem*, which is one of the components of the agent *Impl*, is defined as follows:

```
#let Sem =
# new_definition
#  (‘Sem‘,
#   "Sem =
#    rec ‘X‘
#    (prefix(label(name ‘p‘))
#     (prefix(label(name ‘v‘)) (var ‘X‘)))");;
Sem =
|- Sem =
   rec
   ‘X‘
   (prefix(label(name ‘p‘))(prefix(label(name ‘v‘))(var ‘X‘)))
```

By pretty-printing the above definition in such a way that the usual CCS notation is adopted and by removing the string quotes around constants of type `string` occurring in the CCS terms, the definition becomes the following:

```
#let Sem = new_definition(`Sem`,"Sem = rec X. p.v.X");;
Sem = ⊢ Sem = rec X. p.v.X
```

The agents *Reader*, *Writer* and *Impl* are defined in HOL in a similar way:

```
#let Reader =
# new_definition(`Reader`,"Reader = rec X. -p.br.er.-v.X");;
Reader = ⊢ Reader = rec X. -p.br.er.-v.X

#let Writer =
# new_definition(`Writer`,"Writer = rec X. -p.bw.ew.-v.X");;
Writer = ⊢ Writer = rec X. -p.bw.ew.-v.X

#let Impl =
# new_definition(`Impl`,"Impl = (Reader | (Sem | Writer))\{p,v}");;
Impl = ⊢ Impl = (Reader | (Sem | Writer))\{p,v}
```

Using the conversion `Is_Agent_CONV`, it can be derived that these expressions are all agents. For *Impl* this yields the following theorem:

```
#Is_Agent_Impl;;
⊢ Is_Agent Impl
```

B.1 Proving Behavioural Equivalences

The abstract specification *Spec* of the reader-writer system is defined as follows:

```
#let Spec =
# new_definition(`Spec`,"Spec = rec X. τ.br.er.X + τ.bw.ew.X");;
Spec = ⊢ Spec = rec X. τ.br.er.X + τ.bw.ew.X
```

The agent variable X in *Spec* is guarded and sequential. Thus, in order to show that *Impl* is a correct implementation of *Spec*, by unique fixpoint induction the following observation congruence is to be proved:

```
#g "Obs_Congr Impl (τ.br.er.Impl + τ.bw.ew.Impl)";;
"Obs_Congr Impl (τ.br.er.Impl + τ.bw.ew.Impl)"
```

This goal is proved by applying the lazy expansion strategy. The left-hand side of the behavioural goal is rewritten so to transform it into an expression provably congruent to the right-hand side. The tactic `REW_LHS_TAC` expands only the left-hand side with the definitions of the agents:

```
#e (REW_LHS_TAC [Impl; Reader; Sem; Writer]);;
OK..
"Obs_Congr
 ((rec X. -p.br.er.-v.X |
  (rec X. p.v.X | rec X. -p.bw.ew.-v.X))\{p,v})
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

The recursive agents are unfolded using the tactic `OC_REC_UNF_TAC` and then the recursive subterms replacing the agent variables are folded back using the symmetric version of the definitions of *Reader*, *Writer* and *Sem*. This allows the size of the goal to be reduced making it more readable, but these subterms can be expanded whenever needed.

```
#e (OC_REC_UNF_TAC THEN
#   REWRITE_TAC [SYM Reader; SYM Sem; SYM Writer]);;
OK..
"Obs_Congr
 ((-p.br.er.-v.Reader | (p.v.Sem | -p.bw.ew.-v.Writer))\{p,v})
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

The expansion theorem for observation congruence is applied by means of the tactic `OC_EXP_THM_TAC`. In particular, this means rewriting with the expansion law (A11) and the restriction laws (A6)–(A7) (Section 2.1.3) for deleting those summands obtained from the parallel expansion that are prefixed by the actions in the restriction set. Only the synchronisations are left.

```
#e OC_EXP_THM_TAC;;
OK..
"Obs_Congr
 (τ.((-p.br.er.-v.Reader | (v.Sem | bw.ew.-v.Writer))\{p,v}) +
  τ.((br.er.-v.Reader | (v.Sem | -p.bw.ew.-v.Writer))\{p,v}))
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

Another expansion step pulls the only possible actions *bw* and *br* out of the parallel and restriction subexpressions:

```
#e OC_EXP_THM_TAC;;
OK..
"Obs_Congr
 (τ.bw.((-p.br.er.-v.Reader | (v.Sem | ew.-v.Writer))\{p,v}) +
  τ.br.
   ((er.-v.Reader |
     (v.(Sem | -p.bw.ew.-v.Writer) +
      -p.(v.Sem | bw.ew.-v.Writer)))\{p,v}))
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

Note the expanded subexpression in the second summand of the left-hand side. This is due to the bottom-up strategy of OC_EXP_THM_TAC and to the fact that the action *br* of the other agent in parallel, i.e. $br.\ er.\overline{v}.\ Reader$, is taken out of the composition.

At this point the proof can proceed in two different ways. The first one consists of manipulating only the left-hand side of the behavioural goal using the lazy expansion strategy, as done so far. Two further expansion steps lead to the following goal:

```
#e (OC_EXP_THM_TAC THEN OC_EXP_THM_TAC);;
OK..
"Obs_Congr
 (τ.bw.ew.τ.((-p.br.er.-v.Reader | (Sem | Writer))\{p,v}) +
  τ.br.er.τ.((Reader | (Sem | -p.bw.ew.-v.Writer))\{p,v}))
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

Either the writer or the reader has terminated its task and then released the resource by synchronising again with the semaphore. The new occurrences of the action τ are removed using the τ-law TAU_1 (Section 2.2.4) with the tactic OC_TAU1_TAC:

```
#e OC_TAU1_TAC;;
OK..
"Obs_Congr
 (τ.bw.ew.((-p.br.er.-v.Reader | (Sem | Writer))\{p,v}) +
  τ.br.er.((Reader | (Sem | -p.bw.ew.-v.Writer))\{p,v}))
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

By folding the subexpressions in the left-hand side using the behavioural theorems:

```
#Reader_lemma;;
⊢ Obs_Congr (-p.br.er.-v.Reader) Reader

#Writer_lemma;;
⊢ Obs_Congr (-p.bw.ew.-v.Writer) Writer
```

the following goal is obtained:

```
#e (OC_SUBST_TAC [Reader_lemma; Writer_lemma]);;
OK..
"Obs_Congr
 (τ.bw.ew.((Reader | (Sem | Writer))\{p,v}) +
  τ.br.er.((Reader | (Sem | Writer))\{p,v}))
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

This goal is solved by rewriting with the symmetric form of the definition of *Impl* and the commutativity law SUM_COMM for observation congruence (Section 2.2.4):

```
#e (REWRITE_TAC [SYM Impl] THEN
#   OC_LHS_SUBST1_TAC
#   (RESQ_SPECL ["τ.bw.ew.Impl"; "τ.br.er.Impl"] SUM_COMM));;
OK..
goal proved
⊢ Obs_Congr Impl (τ.br.er.Impl + τ.bw.ew.Impl)
```

A different proof technique can be adopted after the first two expansion steps. It manipulates both sides of the behavioural goal using the substitutivity properties of the CCS operators with respect to observation congruence. Given the goal

```
"Obs_Congr
 (τ.bw.((-p.br.er.-v.Reader | (v.Sem | ew.-v.Writer))\{p,v}) +
  τ.br.
  ((er.-v.Reader |
    (v.(Sem | -p.bw.ew.-v.Writer) +
     -p.(v.Sem | bw.ew.-v.Writer)))\{p,v}))
 (τ.br.er.Impl + τ.bw.ew.Impl)"
```

the substitutivity property for the binary summation operator:

```
#OBS_CONGR_PRESD_BY_SUM;;
⊢ ∀E1 E1' E2 E2' :: Is_Agent.
   Obs_Congr E1 E1' ∧ Obs_Congr E2 E2' ⊃
   Obs_Congr (E1 + E2) (E1' + E2')
```

is used to transform the whole goal by means of the tactic RULE_TAC that applies an inference rule backward, whenever the goal matches the conclusion of that rule. This means that in order to prove the current goal, it is enough to prove the two subgoals obtained by suitably instantiating the premisses of the substitutivity rule. The commutativity law for binary summation is first applied once to the right-hand side of the goal, in order to get the subgoals in the right order:

```
#e (OC_RHS_SUBST1_TAC
#   (RESQ_SPECL ["τ.br.er.Impl"; "τ.bw.ew.Impl"] SUM_COMM));;
"Obs_Congr
 (τ.bw.((-p.br.er.-v.Reader | (v.Sem | ew.-v.Writer))\{p,v}) +
  τ.br.
  ((er.-v.Reader |
    (v.(Sem | -p.bw.ew.-v.Writer) +
     -p.(v.Sem | bw.ew.-v.Writer)))\{p,v}))
 (τ.bw.ew.Impl + τ.br.er.Impl)"
```

The substitutivity property for binary summation is then applied:

```
#e (RULE_TAC OBS_CONGR_PRESD_BY_SUM);;
OK..
2 subgoals
"Obs_Congr
 (τ.br.
  ((er.-v.Reader |
    (v.(Sem | -p.bw.ew.-v.Writer) +
     -p.(v.Sem | bw.ew.-v.Writer)))\{p,v}))
 (τ.br.er.Impl)"

"Obs_Congr
 (τ.bw.((-p.br.er.-v.Reader | (v.Sem | ew.-v.Writer))\{p,v}))
 (τ.bw.ew.Impl)"
```

The substitutivity rule of the prefix operator with respect to observation congruence:

```
#SUBST_PREFIX;;
⊢ ∀E E' :: Is_Agent. Obs_Congr E E' ⊃ (∀u. Obs_Congr (u.E) (u.E'))
```

can be applied backward twice to both subgoals and the resulting goals are then proved by means of lazy expansion. The following tactic solves both subgoals above:

```
#e (RULE_TAC SUBST_PREFIX THEN RULE_TAC SUBST_PREFIX THEN
#   OC_EXP_THM_TAC THEN OC_EXP_THM_TAC THEN OC_TAU1_TAC THEN
#   OC_LHS_SUBST_TAC [Reader_lemma; Writer_lemma] THEN
#   REWRITE_TAC [SYM Impl] THEN
#   OC_SUBST_TAC [RESQ_SPEC "ew.Impl" OBS_CONGR_REFL;
#                 RESQ_SPEC "er.Impl" OBS_CONGR_REFL]);;
OK..
goal proved
   ⋮
⊢ Obs_Congr Impl (τ.br.er.Impl + τ.bw.ew.Impl)

Previous subproof:
goal proved
```

The HOL system prints a trace (here replaced by the dots) of the subgoals which have been proved. There are no further subgoals to solve, thus the desired correctness result is formally verified.

B.2 Checking Modal Properties

As discussed in Section 2.4.2, the correctness of the implementation *Impl* for the reader-writer system can also be checked by verifying that the reading and writing tasks on the common resource are mutually exclusive. This means to prove $Impl \models \Phi$, where the modal formula Φ is defined as

```
#let Fi =
#  new_definition
#   (‘Fi‘,
#    "Fi = (box{tau}(conj(box{br}(nec er))(box{bw}(nec ew))))");;
Fi = ⊢ Fi = box{tau}(conj(box{br}(nec er))(box{bw}(nec ew)))
```

and the derived modal operator of necessity is introduced in HOL as follows:

```
#let nec =
#  new_definition
#   (‘nec‘, "nec u = conj(dmd Univ tt)(box(Univ Diff {u})ff)");;
nec = ⊢ ∀u. nec u = conj(dmd Univ tt)(box(Univ Diff {u})ff)
```

where Univ : $(action)set$ denotes the universe of actions and Diff is the operator of set difference (from the HOL theory `sets`). The satisfaction relation for the necessity operator, $E \models \mathsf{nec}\, u$, is derived from the one for the modal operators in the definition of nec and is given by the following theorem directly in terms of the transitions of the agent E:

```
#SAT_nec;;
⊢ ∀E:: Is_Agent. ∀u.
    Sat E (nec u) =
    (∃E':: Is_Agent. Trans E u E') ∧
    ¬(∃u'. ∃E':: Is_Agent. ¬(u' = u) ∧ Trans E u' E')
```

This theorem asserts that an agent E necessarily performs an action u if and only if there exists a transition from E under u into some E' and E cannot perform any other action different from u.

The goal $\stackrel{?}{\vdash} Impl \models \Phi$ is set in HOL as follows:

```
#g "Sat Impl Fi";;
"Sat Impl Fi"
```

The proof starts by rewriting the goal with the definition of Φ and applying the tactic `SAT_box_TAC` that reduces goals matching the left-hand side of the satisfaction relation for the box operator:

```
#e (PURE_ONCE_REWRITE_TAC [Fi] THEN SAT_box_TAC);;
OK..
2 subgoals
"∀u E'.
  u ∈ {τ} ∧ Trans Impl u E' ⊃
  Sat E' (conj(box{br}(nec er))(box{bw}(nec ew)))"

"Is_Agent Impl"
```

The first subgoal (which, in the HOL subgoal package, is the one at the bottom of the subgoal list) is trivially solved by means of the theorem Is_Agent_Impl. The HOL theorem IN_SING: $\vdash \forall x\ y.\ x \in \{y\} = (x = y)$ is used to rewrite the condition on the action u, and the transitions of *Impl* are derived using the theorem Impl_run (which is obtained using the conversion `Run_CONV`):

```
#e (REWRITE_TAC [IN_SING; Impl_run]);;
OK..
"∀u E'.
  (u = τ) ∧
  ((u = τ) ∧ (E' = (Reader | (v.Sem | bw.ew.-v.Writer))\{p,v}) ∨
   (u = τ) ∧ (E' = (br.er.-v.Reader | (v.Sem | Writer))\{p,v})) ⊃
  Sat E' (conj(box{br}(nec er))(box{bw}(nec ew)))"
```

The outermost connectives of the goal, namely the universal quantifiers, implication, conjunction and disjunction, are removed by repeatedly applying the tactic `STRIP_TAC`. This moves the antecedent of the implication into the assumption list of the goal and splits the conjunction and the disjunction by producing two subgoals:

```
#e (REPEAT STRIP_TAC);;
OK..
2 subgoals
"Sat E' (conj(box{br}(nec er))(box{bw}(nec ew)))"
    [ "u = τ" ]
    [ "E' = (br.er.-v.Reader | (v.Sem | Writer))\{p,v}" ]

"Sat E' (conj(box{br}(nec er))(box{bw}(nec ew)))"
    [ "u = τ" ]
    [ "E' = (Reader | (v.Sem | bw.ew.-v.Writer))\{p,v}" ]
```

The ML code for proving the first subgoal is given below, followed by a few comments.

```
#e (ASSUME_TAC (EQT_ELIM
#   (REWRITE_CONV
#     [ASSUME "E' = (Reader | (v.Sem | bw.ew.-v.Writer))\{p,v}";
#      Is_Agent_restr; Is_Agent_par; Is_Agent_prefix;
#      Is_Agent_Reader; Is_Agent_Sem; Is_Agent_Writer]
#    "Is_Agent E'")) THEN
#   SAT_conj_TAC THEN
#   SAT_box_TAC THEN REWRITE_TAC [IN_SING] THEN
#   REPEAT STRIP_TAC THEN
#   STRIP_ASSUME_TAC (REWRITE_RULE
#   [ASSUME "E' = (Reader | (v.Sem | bw.ew.-v.Writer))\{p,v}";
#    Impl1_run] (ASSUME "TRANS E' u' E''")) THENL
#   [CHECK_ASSUME_TAC
#    (REWRITE_RULE [ASSUME "u' = br"; Action_EQ_CONV "br = bw"]
#     (ASSUME "u' = bw"))
#    ;
#    ASSUME_TAC
#    (MATCH_MP (MATCH_MP TRANS_Is_Agent (ASSUME "TRANS E' u' E''"))
#     (ASSUME "Is_Agent E')) THEN
#    RESQ_REWRITE1_TAC SAT_nec THEN
#    CONV_TAC (DEPTH_CONV RESQ_EXISTS_CONV) THEN
#    REPEAT STRIP_TAC THENL
#    [EXISTS_TAC "(Reader | (v.Sem | -v.Writer))\{p,v}" THEN
#     ASM_REWRITE_TAC
#     [Is_Agent_restr; Is_Agent_par; Is_Agent_prefix;
#      Is_Agent_Reader; Is_Agent_Sem; Is_Agent_Writer] THEN
#     RESTR_TAC THEN EXISTS_TAC "ew" THEN
#     REWRITE_TAC [Action_Distinct_label; COMPL;
#                  Label_IN_CONV "ew" "{p,v}";
#                  Label_IN_CONV "-ew" "{p,v}"] THEN
#     PAR2_TAC THEN PAR2_TAC THEN PREFIX_TAC
#     ;
#     STRIP_ASSUME_TAC (REWRITE_RULE
#     [ASSUME "E'' = (Reader | (v.Sem | ew.-v.Writer))\{p,v}";
#      Impl2_run] (ASSUME "TRANS E'' u'' E'''")) THEN
#     CHECK_ASSUME_TAC (REWRITE_RULE [ASSUME "u'' = ew"]
#                       (ASSUME "¬(u'' = ew")) ]]);;
OK..
goal proved

Previous subproof:
"Sat E' (conj(box{br}(nec er))(box{bw}(nec ew)))"
    [ "u = τ" ]
    [ "E' = (br.er.-v.Reader | (v.Sem | Writer))\{p,v}" ]
```

Some of the above steps require the assumption that E' is an agent. This can be dealt with in various ways. For example, instead of having a subgoal '$\mathsf{Is\_Agent}\ E''$' generated at every such step, the fact that E' is an agent is derived (using the properties of Is_Agent and knowing that *Reader*, *Sem* and *Writer* are all agents) and added to the assumption list of the goal. Otherwise, the tactics requiring these assumptions can be defined so that they check such predicates automatically. Later in the proof, before applying the theorem SAT_nec, the fact that E'' is an agent is derived using the fact that E'' is a derivative of agent E'. Moreover, restricted existential quantifications (due to the rewriting with SAT_nec) are removed by means of the conversion `RESQ_EXISTS_CONV` before the outermost connectives of the goal are stripped.

The theorems Impl1_run and Impl2_run have been derived using `Run_CONV` and give the transitions of the agents involved in the proof. The conversion `Label_IN_CONV` decides whether or not a label is in a given set of labels.

Finally, the HOL system reminds the user that there is another subgoal to prove. This can be solved with a tactic similar to the one above.

Appendix C

The Infinite Counter

This appendix contains the HOL transcripts of the proofs of correctness of the infinite counter (Section 3.2). Similarly to the reader-writer system described in Appendix B, the correctness of the counter has been shown by following two different approaches. First, two different descriptions of the counter have been proved to be observation congruent. Appendix C.1 below presents the interactive HOL session for the (backward) proof illustrated in Section 3.2.1. Second, a specification of the counter has been shown to possess a given modal property. The HOL code for the theorems and lemmas in Section 3.2.2 is reported in Appendix C.2.

The specification $Counter_n$ of the counter is used in both verification proofs. It denotes a family of agents and is introduced in HOL as a function Counter : $num \rightarrow (num)CCS$ as follows:[1]

```
#let Counter =
# new_definition(`Counter`,"Counter (n:num) = conp `Counter` n");;
Counter = ⊢ ∀n. Counter n = conp `Counter` n
```

C.1 Verifying Behavioural Equivalences

The family of agents $Impl(n)$ is defined through a function Impl : $num \rightarrow (num)CCS$, as done for $Counter_n$:

```
Impl = ⊢ ∀n. Impl n = conp `Impl` n
```

The agents C, D and B are instead encoded in HOL as parameterless agent constants of type $(num)CCS$:

[1]Note that the HOL notation for strings, i.e. a string s is denoted by `s`, is maintained in the transcripts of this appendix to avoid confusion between an agent constant and its name.

```
C = ⊢ C = con 'C'

D = ⊢ D = con 'D'

B = ⊢ B = con 'B'
```

The linking operator Link is mechanised as a binary infix operator as follows:

```
#let Link =
# new_infix_definition
#  ('Link',
#   "∀E E': (β)CCS.
#     E Link E' =
#     ((E [u'/u,a'/a,d'/d]) |
#      (E' [u'/up,a'/around,d'/down]))\{u',a',d'}");;
Link =
⊢ ∀E E'.
   E Link E' =
   ((E[u'/u,a'/a,d'/d]) | (E'[u'/up,a'/around,d'/down]))\{u',a',d'}
```

The defining equations for the agent constants are given as a system of mutually recursive equations, which is represented in HOL via the function CDfr:

```
#CDfr;;
⊢ CDfr =
  (λs n.
    ((s = 'Counter') ⇒
     ((n = 0) ⇒
      up.(Counter 1) + around.(Counter 0) |
      up.(Counter(n+1)) + down.(Counter(n-1)) |
     ((s = 'C') ⇒ up.(C Link C) + down.D |
      ((s = 'D') ⇒ -d.C + -a.B |
       ((s = 'B') ⇒ up.(C Link B) + around.B |
        ((s = 'Impl') ⇒ ((n = 0) ⇒ B | C Link (Impl(n-1))) |
         ARBag))))))
```

The expressions associated to the above constants are all agents. In fact, CDfr is proved to be a defining equations function using PROVE_IS_DEF_FUN. Let CDf be defined as the abstraction of CDfr, namely CDf = Def_Fun CDfr. The behavioural theorems that assert the equivalence between a constant and the associated agent are derived using the conversions CON_EVAL_CONV and CONP_EVAL_CONV and the unfolding law. These theorems are needed for rewriting agents up to behavioural equivalences. For observation equivalence and constants C, D and B, such theorems are:

```
C_OE_THM = ⊢ Obs_Equiv CDf C (up.(C Link C) + down.D)

D_OE_THM = ⊢ Obs_Equiv CDf D (-d.C + -a.B)

B_OE_THM = ⊢ Obs_Equiv CDf B (up.(C Link B) + around.B)
```

The corresponding theorems for observation congruence are C_OC_THM, D_OC_THM and B_OC_THM respectively. Moreover, the following theorems involving the linking operator will be useful when proving the inductive case of the main proof:

```
OBS_EQUIV_D_C = ⊢ Obs_Equiv CDf (D Link C) (C Link D)

OBS_EQUIV_D_B = ⊢ Obs_Equiv CDf (D Link B) (B Link B)

OBS_CONGR_B_B = ⊢ Obs_Congr CDf (B Link B) B

OBS_EQUIV_D_Impl = ⊢ ∀n. Obs_Equiv CDf (D Link(Impl n)) (Impl n)
```

By applying unique fixpoint induction, showing that Impl and Counter are observation congruent means proving the following goal:

```
#g "∀n.
#   Obs_Congr CDf
#   (Impl n)
#   (Ag_Subst (CDf `Counter` n) `Impl` `Counter`)";;
"∀n.
  Obs_Congr CDf(Impl n)(Ag_Subst(CDf `Counter` n)`Impl` `Counter`)"
```

Given an agent expression E and strings s, s', the function Ag_Subst substitutes s for all occurrences of s' in E. In this case, given the agent identified by the string *Counter* and parameter n in CDf, Ag_Subst substitutes occurrences of the agent constant Impl for occurrences of Counter by replacing their names. Similarly to the function CCS_Subst (Section 2.2.1), Ag_Subst : $(\beta)CCS \rightarrow string \rightarrow string \rightarrow (\beta)CCS$ is defined as a primitive recursive function over $(\beta)CCS$.

The proof is by induction on the parameter n. By applying the built-in tactic INDUCT_TAC, the two cases to be proved are:

```
#e INDUCT_TAC;;
OK..
2 subgoals
"Obs_Congr CDf
 (Impl(n+1))
 (Ag_Subst(CDf `Counter`(n+1))`Impl` `Counter`)"
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]

"Obs_Congr CDf(Impl 0)(Ag_Subst(CDf `Counter` 0)`Impl` `Counter`)"
```

The basis case is solved by rewriting with the definitions and the behavioural theorems for the agents involved. First, the definitions of Counter (where Counter0_THM is the clause of the definition for $n=0$), Ag_Subst and Impl (by taking its symmetric form with the rule GSYM) are used:

```
#e (REWRITE_TAC [Counter0_THM; Counter; Ag_Subst; GSYM Impl]);;
OK..
"Obs_Congr CDf(Impl 0)(up.(Impl 1) + around.(Impl 0))"
```

The agent constant Impl is then rewritten up to observation congruence for $n=1$ and $n=0$:

```
#e (OC_SUBST_TAC [Impl1_OC_THM; Impl0_OC_THM]);;
OK..
"Obs_Congr CDf B(up.(C Link B) + around.B)
```

The agent B is observation congruent to the right-hand side of the current goal and the basis case is solved by reflexivity of observation congruence (implicitly applied by the substitution tactic):

```
#e (OC_SUBST1_TAC B_OC_THM);;
OK..
goal proved
⊢ Obs_Congr CDf(Impl 0)(Ag_Subst(CDf `Counter` 0)`Impl` `Counter`)

Previous subproof:
"Obs_Congr CDf
 (Impl(n+1))
 (Ag_Subst(CDf `Counter`(n+1))`Impl` `Counter`)"
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
```

The first steps of the proof of the inductive case are as follows:

```
#e (OC_LHS_SUBST1_TAC (SPEC "n: num" ImplSUC_OC_THM) THEN
#   REWRITE_TAC [Link; CounterSUC_THM; Counter;
#                Ag_Subst; GSYM Impl]);;
OK..
"Obs_Congr CDf
 ((C[u'/u,a'/a,d'/d] | (Impl n)[u'/up,a'/around,d'/down])\{u',a',d'})
 (up.(Impl(n+2)) + down.(Impl n))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]

#e (STRIP_ASSUME_TAC (SPEC "n: num" num_CASES));;
OK..
2 subgoals
"Obs_Congr CDf
 ((C[u'/u,a'/a,d'/d] | (Impl n)[u'/up,a'/around,d'/down])\{u',a',d'})
 (up.(Impl(n+2)) + down.(Impl n))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = n'+1" ]

"Obs_Congr CDf
 ((C[u'/u,a'/a,d'/d] | (Impl n)[u'/up,a'/around,d'/down])\{u',a',d'})
 (up.(Impl(n+2)) + down.(Impl n))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
```

The first step rewrites the inductive subgoal with the definitions of Impl, Counter, Ag_Subst and Link. Next, the inductive hypothesis needs to be rewritten as well using the definitions of Counter and Ag_Subst so that it can be applied to rewrite the goal. Note that the agent expression associated to name *Counter* and parameter n in CDf depends on the value of n, as the following theorem says:

```
#CounterN_THM;;
⊢ ∀n.
   CDf `Counter` n =
   ((n = 0) ⇒
    up.(Counter 1) + around.(Counter 0) |
    up.(Counter(n+1)) + down.(Counter(n-1)))
```

Using the tactic STRIP_ASSUME_TAC with the HOL theorem

```
num_CASES = ⊢ ∀m. (m = 0) ∨ (∃n. m = n+1)
```

case analysis is performed on the value of the parameter n, thus splitting the proof of the inductive case into two subproofs. The first subgoal is rewritten using the new assumption $n=0$ and the definition of C:

```
#e (ASM_REWRITE_TAC[] THEN OC_LHS_SUBST1_TAC C_OC_THM);;
OK..
"Obs_Congr CDf
 (((up.(C Link C) + down.D)[u'/u,a'/a,d'/d] |
   (Impl 0)[u'/up,a'/around,d'/down])\{u',a',d'})
 (up.(Impl 2) + down.(Impl 0))"
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
```

The inductive hypothesis is now rewritten with the assumption $n=0$ and the definitions of Counter, Ag_Subst and Impl, and added to the assumption list:

```
#e (ASSUME_TAC
#   (REWRITE_RULE
#    [CounterN_THM; ASSUME "n = 0"; Counter; Ag_Subst; GSYM Impl]
#    (ASSUME "Obs_Congr CDf
#             (Impl n)
#             (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)")));;
OK..
"Obs_Congr CDf
 (((up.(C Link C) + down.D)[u'/u,a'/a,d'/d] |
   (Impl 0)[u'/up,a'/around,d'/down])\{u',a',d'})
 (up.(Impl 2) + down.(Impl 0))"
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
    [ "Obs_Congr CDf(Impl 0)(up.(Impl 1) + around.(Impl 0))" ]
```

The (new formulation of the) inductive hypothesis is then applied to the goal:

```
#e (OC_LHS_SUBST1_TAC
#   (ASSUME "Obs_Congr CDf
#            (Impl 0)
#            (up.(Impl 1) + around.(Impl 0))"));;
OK..
"Obs_Congr CDf
 (((up.(C Link C) + down.D)[u'/u,a'/a,d'/d] |
   (up.(Impl 1) +
    around.(Impl 0))[u'/up,a'/around,d'/down])\{u',a',d'})
 (up.(Impl 2) + down.(Impl 0))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
    [ "Obs_Congr CDf(Impl 0)(up.(Impl 1) + around.(Impl 0))" ]
```

The expansion theorem is used to transform the left-hand subterm by applying the laws for relabelling, parallel composition and restriction:

```
#e OC_EXP_THM_TAC;;
OK..
"Obs_Congr CDf
 (up.
  (((C Link C)[u'/u,a'/a,d'/d] |
    (u'.((Impl 1)[u'/up,a'/around,d'/down]) +
     a'.((Impl 0)[u'/up,a'/around,d'/down])))\{u',a',d'}) +
 down.
  ((D[u'/u,a'/a,d'/d] |
    (u'.((Impl 1)[u'/up,a'/around,d'/down]) +
     a'.((Impl 0)[u'/up,a'/around,d'/down])))\{u',a',d'}))
 (up.(Impl 2) + down.(Impl 0))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
    [ "Obs_Congr CDf(Impl 0)(up.(Impl 1) + around.(Impl 0))" ]
```

The above expansion shows that the actions of the second component in the parallel compositions are hidden by restriction because they are renamed with labels occurring in the restriction set. Hence, this agent component does not contribute to the behaviour of the left-hand side of the goal and can be replaced by the agent $(Impl\,0)\,[u'/up, a'/around, d'/down]$, as such agents are observation congruent by the inductive hypothesis and the expansion theorem. The following goal is obtained:

```
"Obs_Congr CDf
 (up.
  (((C Link C)[u'/u,a'/a,d'/d] |
    (Impl 0)[u'/up,a'/around,d'/down])\{u',a',d'}) +
  down.
  ((D[u'/u,a'/a,d'/d] |
   (Impl 0)[u'/up,a'/around,d'/down])\{u',a',d'}))
 (up.(Impl 2) + down.(Impl 0))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
    [ "Obs_Congr CDf(Impl 0)(up.(Impl 1) + around.(Impl 0))" ]
```

The goal is folded back using the definition of the linking operator:

```
#e (REWRITE_TAC [GSYM Link]);;
OK..
"Obs_Congr CDf
 (up.((C Link C) Link (Impl 0)) + down.(D Link (Impl 0)))
 (up.(Impl 2) + down.(Impl 0))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
    [ "Obs_Congr CDf(Impl 0)(up.(Impl 1) + around.(Impl 0))" ]
```

The last steps of the proof apply the associativity of the linking operator and the previously proved theorem OBS_EQUIV_D_Impl:

```
#e (OC_LHS_SUBST_TAC
#   [RESQ_MATCH_MP OBS_CONGR_SYM
#    (ISPECL ["CDf"; "C"; "C"; "Impl 0"] OC_Link_Assoc);
#    SPEC "down"
#    (RESQ_MATCH_MP PROP6 (SPEC "0" OBS_EQUIV_D_Impl))]);;
OK..
"Obs_Congr CDf
 (up.(C Link (C Link (Impl 0))) + down.(Impl 0))
 (up.(Impl 2) + down.(Impl 0))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = 0" ]
    [ "Obs_Congr CDf(Impl 0)(up.(Impl 1) + around.(Impl 0))" ]
```

where RESQ_MATCH_MP is a version of the inference rule for modus ponens with automatic matching that deals with restricted quantification, and the theorem

```
#PROP6;;
⊢ ∀Df. ∀E E' :: Is_Agent.
    Obs_Equiv Df E E' ⊃ (∀u. Obs_Congr Df (u.E) (u.E'))
```

asserts that observation equivalence is strengthened by the prefix operator. Finally, the goal is solved by rewriting with the definition of Impl:

```
#e (OC_RHS_SUBST_TAC [Impl2_OC_THM; Impl1_OC_THM]);;
OK..
goal proved
.. |- Obs_Congr CDf
      ((C [u'/u,a'/a,d'/d] |
       (Impl n) [u'/up,a'/around,d'/down])\{u',a',d'})
      (up.(Impl(n+2)) + down.(Impl n))

Previous subproof:
"Obs_Congr CDf
 ((C[u'/u,a'/a,d'/d] | (Impl n)[u'/up,a'/around,d'/down])\{u',a',d'})
 (up.(Impl(n+2)) + down.(Impl n))
    [ "Obs_Congr CDf
       (Impl n)
       (Ag_Subst(CDf `Counter` n)`Impl` `Counter`)" ]
    [ "n = n'+1" ]
```

The remaining subgoal is proved with a tactic similar to the one for the case $n=0$, thus obtaining a formal verification of the correctness of the implementation $Impl(n)$ with respect to the specification $Counter_n$ modulo observation congruence.

C.2 Checking Modal Properties

This appendix contains the HOL definitions, theorems and tactics for checking that the infinite counter possesses a given modal property, as described in Section 3.2.2.

The specification of the counter $Counter_n$ has already been encoded in HOL and the environment defined by the function CDf is still used for evaluating the agents corresponding to $Counter_n$ for each n. The m-times application of a modal operator f to a set of actions A and a modal formula Fm is defined in HOL by introducing an operator Raise in a primitive recursive way using the ML function new_prim_rec_definition:

```
#let Raise =
#  new_prim_rec_definition
#   (`Raise`,
#    "(Raise (f:(action)set → eHML → eHML) A 0 Fm = Fm) ∧
#     (Raise f A (m+1) Fm = f A (Raise f A m Fm))");;
Raise =
⊢ (∀f A Fm. Raise f A 0 Fm = Fm) ∧
   (∀f A m Fm. Raise f A(m+1)Fm = f A(Raise f A m Fm))
```

In order to prove Key_Lemma, the following theorems are needed:

```
#Box_Dmd_THM;;
⊢ ∀n Fm.
   Sat CDf(Counter n)(box{up}(dmd{down}Fm)) = Sat CDf(Counter n)Fm

#Box_THM;;
⊢ ∀n Fm.
   Sat CDf(Counter n)(box{up}Fm) = Sat CDf(Counter(n+1))Fm

#Dmd_THM;;
⊢ ∀n Fm. Sat CDf(Counter(n+1))(dmd{down}Fm) = Sat CDf(Counter n)Fm

#Raise_Perm;;
⊢ ∀m f A Fm. Raise f A(m+1)Fm = Raise f A m(f A Fm)
```

Key_Lemma is proved by mathematical induction on m and the tactic that proves it in HOL is given below. Rewriting with the definition of the operator Raise and the theorem Box_Dmd_THM solves the basis case of the induction. Then, the definition of Raise and the theorems Box_THM and Raise_Perm (suitably instantiated) allow one to manipulate the subgoal for the induction step, so that the inductive hypothesis can be applied to the left-hand side of the goal using the tactic `ONCE_ASM_REW_LHS_TAC` (i.e. a version of `ASM_REWRITE_TAC` which rewrites only once the left-hand side of a goal with a list of theorems and assumptions). The last step of the proof consists of folding back the induction subgoal by applying the symmetric versions of those theorems used before for expanding the subgoal.

```
#let Key_Lemma =
#  prove_thm
#   (`Key_Lemma`,
#    "∀m n Fm.
#      Sat CDf (Counter n)
#      (Raise box{up}(m+1)(Raise dmd{down}(m+1)Fm)) =
#      Sat CDf (Counter n) (Raise box{up}m(Raise dmd{down}m Fm))",
#    INDUCT_TAC THENL
#    [REWRITE_TAC [Raise; Box_Dmd_THM]
#     ;
#     REPEAT GEN_TAC THEN
#     ONCE_REW_LHS_TAC [SPEC "box" (CONJUNCT2 Raise)] THEN
#     PURE_ONCE_REWRITE_TAC [Box_THM] THEN
#     ONCE_REW_LHS_TAC [SPECL ["m+1"; "dmd"] Raise_Perm] THEN
#     ONCE_ASM_REW_LHS_TAC[] THEN
#     REWRITE_TAC [GSYM Raise_Perm; GSYM Box_THM; Raise] ]);;
Key_Lemma =
⊢ ∀m n Fm.
   Sat CDf(Counter n)(Raise box{up}(m+1)(Raise dmd{down}(m+1)Fm)) =
   Sat CDf(Counter n)(Raise box{up}m(Raise dmd{down}m Fm))
```

Finally, the main property can be checked in HOL by induction on m:

```
#let Counter_Proof =
#  prove_thm
#   (`Counter_Proof`,
#    "∀m n.
#     Sat CDf (Counter n) (Raise box{up}m(Raise dmd{down}m tt))",
#    INDUCT_TAC THENL
#    [REWRITE_TAC [Raise; SAT_tt]
#     ;
#     PURE_ONCE_REWRITE_TAC [Key_Lemma] THEN ASM_REWRITE_TAC[]]);;
Counter_Proof =
⊢ ∀m n. Sat CDf(Counter n)(Raise box{up}m(Raise dmd{down}m tt))
```

The basis case is solved by rewriting with the definitions of the Raise operator and the satisfaction relation for the true formula. The induction step is proved by applying Key_Lemma and then rewriting with the inductive hypothesis.

Appendix D

The Value-Passing Example

This appendix presents the interactive HOL session for the proof of correctness of the communicating system given in Section 4.6. As in the previous example sessions, the HOL transcripts will be edited to help readability, HOL indentation will be altered to save space, type information will be usually removed and value-passing expressions will be parsed and pretty-printed, so that the notation in Section 4.1 can be adopted instead of its HOL representation.

In this example ports are assumed to be denoted by strings, and the exchanged values are natural numbers. There are no indexed summation agents, so the type variable γ will not be instantiated. The value-passing agents that describe the communicating system are defined as parameterless agent constants of type $(\gamma, num, string)CCSv$:

```
Spec = ⊢ Spec = Con 'Spec'

P = ⊢ P = Con 'P'

Q = ⊢ Q = Con 'Q'

Impl = ⊢ Impl = Con 'Impl'
```

Their defining equations are introduced in HOL via the function DfHr below. The expressions associated to the agent constants are all agents. Thus, DfHr represents a defining equations function and let DfH be its abstraction, i.e. DfH = Def_Funv DfHr.

The value domain under consideration is the universal set of natural numbers, namely Univ : $(num)set$ in HOL, and two predicates on numbers are used in the agent specifications: the predicate Even (built-in in the HOL system) and the divisibility predicate divides, for which a HOL theory has been developed by Harrison [56]. Arithmetic conversions and a few properties about these predicates will be useful in the proof for deriving new facts about the data in the agent specifications.

```
DfHr =
⊢ DfHr =
  (λs p.
    ((s = 'Spec') ⇒
     In 'in'
     (λx. Cond_Else(6 divides x)(Out 'out'(6 * x)Spec)Spec) |
     ((s = 'P') ⇒
      In 'in'
      (λx. Cond_Else(Even x)(Out 'b'(2 * x)(In 'a'(λz. P)))P) |
      ((s = 'Q') ⇒
       In 'b'
       (λx.
         Cond_Else
         (3 divides x)
         (Out 'out'(3 * x)(Out 'a' 0 Q))
         (Out 'a' 0 Q)) |
       ((s = 'Impl') ⇒ Restr(Par P Q){in 'a',in 'b'} | ARBagv)))))
```

Agents will be manipulated modulo observation congruence and equivalence. In the case of observation congruence, these theorems are as follows:

```
Spec_OC_THM =
⊢ Obs_Congr_v Univ DfH
  Spec
  (In 'in'(λx. Cond_Else(6 divides x)(Out 'out'(6 * x)Spec)Spec))

P_OC_THM =
⊢ Obs_Congr_v Univ DfH
  P
  (In 'in'(λx. Cond_Else(Even x)(Out 'b'(2 * x)(In 'a'(λz. P)))P))

Q_OC_THM =
⊢ Obs_Congr_v Univ DfH
  Q
  (In 'b'
   (λx.
     Cond_Else
     (3 divides x)
     (Out 'out'(3 * x)(Out 'a' 0 Q))
     (Out 'a' 0 Q)))

Impl_OC_THM =
⊢ Obs_Congr_v Univ DfH Impl(Restr(Par P Q){in 'a',in 'b'})
```

Given the value domain Univ and the environment defined by DfH, the initial goal is the following observation congruence:

```
"Obs_Congr_v Univ DfH
 Impl
 (in(x). if (6 divides x) then -out(6 * x).Impl else Impl)"
    [ "∃a b. ¬(a = b)" ]
```

The first steps of the proof are as follows:

```
#e (OCv_LHS_SUBST_TAC [Impl_OC_THM; P_OC_THM; Q_OC_THM]);;
OK..
"Obs_Congr_v Univ DfH
 ((in(x). if (Even x) then -b(2 * x).a(z).P else P |
   b(x). if (3 divides x)
         then -out(3 * x).-a(0).Q else -a(0).Q)\{a,b})
 (in(x). if (6 divides x) then -out(6 * x).Impl else Impl)"
    [ "∃a b. ¬(a = b)" ]

#e (OCv_LHS_SUBST1_TAC
#   (BETA_RULE
#    (ISPECL
#     ["Univ"; "DfH";
#      "in"; "λx. if (Even x) then -b(2 * x).a(z).P else P";
#      "b"; "λx. if (3 divides x)
#                then -out(3 * x).-a(0).Q else -a(0).Q"]
#     OBSCv_PAR_INPUT_INPUT)));;
OK..
"Obs_Congr_v Univ DfH
 ((in(x).
   (if (Even x) then -b(2 * x).a(z).P else P |
    b(x').
    if (3 divides x') then -out(3 * x').-a(0).Q else -a(0).Q) +
   b(y).
   (in(x). if (Even x) then -b(2 * x).a(z).P else P |
    if (3 divides y) then -out(3 * y).-a(0).Q else -a(0).Q))\{a,b})
 (in(x). if (6 divides x) then -out(6 * x).Impl else Impl)"
    [ "∃a b. ¬(a = b)" ]
```

First, the left-hand side of the behavioural goal is expanded with the definitions of the agents involved, up to observation congruence. The resulting left-hand side is an instance of the expansion theorem, which is applied in various steps. The law OBSCv_PAR_INPUT_INPUT (Section 4.5) is applied to rewrite the internal parallel composition. The appropriate instance of this law which matches the parallel subterm is substituted in the left-hand side using the tactic OCv_LHS_SUBST1_TAC (β-reduction

is used to reduce application of functions in the right-hand side of the theorem being applied). Note how the HOL system has renamed the bound value variables of the input prefixed terms inside the parallel composition.

Several laws for the restriction operator are then used for reducing the goal. The tactic OCv_RESTR_ELIM_TAC applies one or more laws for restriction on disjoint subterms at any depth. This tactic is parameterised on a conversion that decides equality between ports. In this example, the built-in conversion string_EQ_CONV for deciding equality of strings is given as the parameter. The first application of the laws pushes the restriction operator down into the binary summation. The second one applies the laws for the restriction of an input prefixed agent to both summands in the left-hand side. The port b of the second summand occurs in the set of restricted labels, thus the result is the inactive agent Nil, which is deleted by the tactic OCv_SUM_NIL_TAC that applies the laws for inactive summands.

```
#e (OCv_RESTR_ELIM_TAC string_EQ_CONV THEN
#   OCv_RESTR_ELIM_TAC string_EQ_CONV THEN OCv_SUM_NIL_TAC);;
OK..
"Obs_Congr_v Univ DfH
 (in(x).
  ((if (Even x) then -b(2 * x).a(z).P else P |
    b(x'). if (3 divides x')
           then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b}))
 (in(x). if (6 divides x) then -out(6 * x).Impl else Impl)"
    [ "∃a b. ¬(a = b)" ]
```

The current goal is an observation congruence between two input prefixed agents with the same port. The ω-data-rule OBSEQv_OBSCv_INPUT (Section 4.5) is used to transform the whole goal by means of the tactic RULE_TAC that applies an inference rule backward, whenever the goal matches the conclusion of that rule:

```
#e (RULE_TAC OBSEQv_OBSCv_INPUT);;
OK..
"∀v.
  v ∈ Univ ⊃
  Is_Agentv (...) ∧ Is_Agentv (...) ∧
  Obs_Equiv_v Univ DfH
  ((λx.
    (if (Even x) then -b(2 * x).a(z).P else P |
     b(x'). if (3 divides x')
            then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b})v)
  ((λx. if (6 divides x) then -out(6 * x).Impl else Impl)v)"
    [ "∃a b. ¬(a = b)" ]
```

The antecedent of the goal can be reduced to true by applying the HOL theorem IN_UNIV: $\vdash \forall x{:}\,\alpha.\ x \in \mathsf{Univ}$ asserting that any element of any type is in the universal set of that type. The conditions about the two CCS terms being agents are easily verified (in these conditions the terms are replaced by the dots '...' to save space, while they are given in the behavioural equivalence). Then, β-reduction is applied:

```
#e (GEN_TAC THEN REWRITE_TAC [IN_UNIV] THEN BETA_TAC);;
OK..
"Obs_Equiv_v Univ DfH
 ((if (Even v) then -b(2 * v).a(z).P else P |
   b(x'). if (3 divides x')
          then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b})
 (if (6 divides v) then -out(6 * v).Impl else Impl)"
    [ "∃a b. ¬(a = b)" ]
```

More transformations are performed on the left-hand side of the goal using the distributivity laws OBSEQv_PAR_DISTR_COND_L and OBSEQv_RESTR_DISTR_COND for the parallel and restriction operators with respect to the two-armed conditional (Section 4.6.1). The theorem OBSEQv_COND_ELSE is then applied to both sides of the resulting goal, thus yielding the new goal:

```
"Obs_Equiv_v Univ DfH
 (Even v ⇒
  (-b(2 * v).a(z).P |
   b(x'). if (3 divides x')
          then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b} |
  (P |
   b(x'). if (3 divides x')
          then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b})
 (6 divides v ⇒ -out(6 * v).Impl | Impl)"
    [ "∃a b. ¬(a = b)" ]
```

Both sides of the observation equivalence depend on some condition on the data. The way to proceed is to perform a case analysis on one of the two boolean expressions. If a case analysis on the divisibility condition is performed, a further case split on the even condition can be avoided when the boolean expression '6 divides v' is assumed to be true. The case split on the divisibility condition leads to the following subgoals:

```
#e (ASM_CASES_TAC "6 divides v");;
OK..
2 subgoals
...
    [ "∃a b. ¬(a = b)" ]
    [ "¬6 divides v" ]

...
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
```

where the dots '...' stand for the goal shown in the previous box. Given the first subgoal, the condition Even v is derived from its assumptions by applying some theorems about divisibility and multiplication and using the built-in conversion REDUCE_CONV for reducing numbers [51].

```
#e (ASSUME_TAC
#   (REWRITE_RULE [DIVIDES_2]
#    (MATCH_MP DIVIDES_LMUL2
#     (REWRITE_RULE [SYM (REDUCE_CONV "3 * 2")]
#      (ASSUME "6 divides v")))));;
OK..
...
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
```

The subgoal is then rewritten with the assumptions:

```
#e (ASM_REWRITE_TAC[]);;
OK..
"Obs_Equiv_v Univ DfH
 ((-b(2 * v).a(z).P |
   b(x'). if (3 divides x')
          then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b})
 (-out(6 * v).Impl)"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
```

The two prefixed agents in the left-hand side can synchronise through the common port b and exchange the data $2 * v$, which will replace the value variable x'. The theorem OBSEQv_PAR_OUT_IN_SYNC (Section 4.5) is applied by means of the substitution tactic for implicational theorems. Two subgoals are produced, one for the

proof about the exchanged value expression, and the other for the proof about the process behaviour under the assumption on the data just transmitted.

```
#e (OEv_LHS_IMP_SUBST1_TAC
#   (BETA_RULE
#    (ISPECL
#     ["DfH"; "b"; "a(z).P";
#      "λx'.
#       if (3 divides x') then -out(3 * x').-a(0).Q else -a(0).Q"]
#     (UNDISCH
#      (ISPECL ["2 * v"; "Univ"] OBSEQv_PAR_OUT_IN_SYNC)))));;
OK..
2 subgoals
"Obs_Equiv_v Univ DfH
 (((-b(2 * v).
    (a(z).P |
     b(x'). if (3 divides x')
            then -out(3 * x').-a(0).Q else -a(0).Q) +
    b(x).
    (-b(2*v).a(z).P |
     if (3 divides x) then -out(3 * x).-a(0).Q else -a(0).Q)) +
   τ.
    (a(z).P |
     if (3 divides (2 * v))
     then -out(3 * (2 * v)).-a(0).Q else -a(0).Q))\{a,b})
 (-out(6 * v).Impl)"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]

"(2 * v) ∈ Univ"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
```

The exchanged data is verified to be in the value domain by rewriting once more with the theorem IN_UNIV. As regards the process subgoal, the first two summands introduced by the expansion law are removed by applying the laws for the restriction operator, as the port b appears in the restriction set, and the laws for deleting occurrences of the inactive agent. In the new goal only the τ-prefixed summand associated to the previous communication is left:

```
"Obs_Equiv_v Univ DfH
 (τ.
  (a(z).P |
   if (3 divides (2 * v))
   then -out(3 * (2 * v)).-a(0).Q else -a(0).Q)\{a,b})
 (-out(6 * v).Impl)"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
```

The left-hand side contains a silent action which is not matched in the right-hand side. This occurrence of τ is removed with the law TAU_WEAKv for observation equivalence (Section 4.5).

```
#e (OEv_LHS_SUBST1_TAC
#   (RESQ_SPEC "(a(z).P |
#                if (3 divides (2 * v))
#                then -out(3 * (2 * v)).-a(0).Q
#                else -a(0).Q)\{a,b}"
#    (ISPECL ["Univ"; "DfH"] TAU_WEAKv)));;
OK..
"Obs_Equiv_v Univ DfH
 ((a(z).P |
   if (3 divides (2 * v))
   then -out(3 * (2 * v)).-a(0).Q else -a(0).Q)\{a,b})
 (-out(6 * v).Impl)"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
```

The distributivity laws for the parallel and restriction operators are applied again, thus leading to another case analysis situation:

```
"Obs_Equiv_v Univ DfH
 (3 divides (2 * v) ⇒
  (a(z).P | -out(3 * (2 * v)).-a(0).Q)\{a,b} |
  (a(z).P | -a(0).Q)\{a,b})
 (-out(6 * v).Impl)"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
```

Once more, the case split on the boolean expression can be avoided by deriving it from the assumptions with some forward reasoning:

```
#e (ASSUME_TAC
#   (SPEC "2"
#    (MATCH_MP DIVIDES_LMUL
#     (MATCH_MP DIVIDES_RMUL2
#      (REWRITE_RULE [SYM (REDUCE_CONV "3 * 2")]
#       (ASSUME "6 divides v"))))));;
OK..
...
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
    [ "3 divides (2 * v)" ]
```

The newly-derived assumption is used for rewriting the goal. The law for the parallel operator OBSEQv_PAR_IN_OUT_NO_SYNC (Section 4.5) and the laws for restriction and deletion of inactive summands are applied again, thus getting the goal:

```
"Obs_Equiv_v Univ DfH
 (-out(3 * (2 * v)).((a(z).P | -a(0).Q)\{a,b}))
 (-out(6 * v).Impl)"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
    [ "3 divides (2 * v)" ]
```

The substitutivity property OBSEQv_SUBST_OUTPUT for the output operator (Section 4.5) can be used in a backward manner to manipulate both specifications in the current goal. Two subgoals result from this application, one for the data (the two expressions in the output agents have to be proved equal) and the other for the process behaviour:

```
#e (RULE_TAC OBSEQv_SUBST_OUTPUT);;
OK..
2 subgoals
"Obs_Equiv_v Univ DfH
 ((a(z).P | -a(0).Q)\{a,b})
 Impl"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
    [ "3 divides (2 * v)" ]

"3 * (2 * v) = 6 * v"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
    [ "3 divides (2 * v)" ]
```

The data subgoal is easily solved by applying associativity of multiplication and using the built-in conversion for evaluating arithmetic expressions. As before, the process subgoal is rewritten using the laws for parallel and restriction operators and for inactive summands, and the following goal is obtained:

```
"Obs_Equiv_v Univ DfH
 (τ.((P | Q)\{a,b}))
 Impl"
    [ "∃a b. ¬(a = b)" ]
    [ "6 divides v" ]
    [ "Even v" ]
    [ "(2 * v) ∈ Univ" ]
    [ "3 divides (2 * v)" ]
    [ "0 ∈ Univ" ]
```

Note that the assumption about the value 0, exchanged during the communication through the port a, has been added to the assumption list of the goal. This goal is simply solved by rewriting with the definition of the agent Impl (taking the symmetric equivalence with OBSEQv_SYM (Section 4.5)) and with the law TAU_WEAKv:

```
#e (OEv_LHS_SUBST_TAC
#   [RESQ_MATCH_MP OBSEQv_SYM Impl_OE_THM;
#    RESQ_SPEC "Impl" (ISPECL ["Univ"; "DfH"] TAU_WEAKv)]);;
goal proved
  ⋮

Previous subproof:
"Obs_Equiv_v Univ DfH
 (Even v ⇒
  (-b(2 * v).a(z). P |
   b(x'). if (3 divides x')
          then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b} |
  (P |
   b(x'). (3 divides x')
          then -out(3 * x').-a(0).Q else -a(0).Q)\{a,b})
 (6 divides v ⇒ -out(6 * v).Impl | Impl)"
    [ "∃a b. ¬(a = b)" ]
    [ "¬6 divides v" ]
```

The HOL system now presents the second subgoal generated by the case split on the divisibility condition. Its proof is similar to the one of the first subgoal, except that the case analysis on the condition Even v will have to be performed, as it cannot be derived from the data conditions in the assumption list.

Bibliography

[1] Aït Mohamed, O., 'Mechanizing a π-calculus equivalence in HOL', in [113], pp. 1–16.

[2] Akkerman, G.J. and J.C.M. Baeten, 'Term Rewriting Analysis in Process Algebra', Technical Report P9006, University of Amsterdam, 1990.

[3] Archer, M., J. J. Joyce, K. N. Levitt, and P. J. Windley (eds.), Proceedings of the *1991 International Workshop on the HOL Theorem Proving System and its Applications*, IEEE Computer Society Press, 1992.

[4] Arnold, A., 'MEC: a system for constructing and analysing transition vectors', in [116], pp. 117–132.

[5] Aujla, S. S. and M. Fletcher, 'The Boyer-Moore Theorem Prover and LOTOS', in [121].

[6] Austry, D. and G. Boudol, 'Algèbre de Processus et Synchronisation', *Journal of Theoretical Computer Science* **30**, 1984, pp. 91–131.

[7] Back, R. J. R. and J. von Wright, 'Refinement Concepts Formalised in Higher Order Logic', *Formal Aspects of Computing* **2**, No. 3, 1990, pp. 247–272.

[8] Baeten, J.C.M. and W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science **18**, Cambridge University Press, 1990.

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

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

[11] Bolognesi, T., E. Brinksma, and C. A. Vissers (eds.), Proceedings of the *3rd LotoSphere Workshop & Seminar*, CNR-CNUCE, Pisa, September 1992.

[12] Bolognesi, T. and M. Caneve, 'Squiggles — A Tool for the Analysis of LOTOS Specifications', in [121], pp. 201–216.

[13] 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.

[14] Boudol, G., V. Roy, R. de Simone, and D. Vergamini, 'Process Calculi, from Theory to Practice: Verification Tools', in [116], pp. 1–10.

[15] Brackin, S. H. and S.-K. Chin, 'Server-Process Restrictiveness in HOL', in [74], pp. 450–463.

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

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

[18] Bryant, R. E., 'Graph-Based Algorithms for Boolean Function Manipulation', *IEEE Transactions on Computers* **C-35**, No. 8, 1986, pp. 677–691.

[19] Bruns, G., 'A language for value-passing CCS', Technical Report ECS-LFCS-91-175, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1991.

[20] Burch, J. R., E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, 'Symbolic Model Checking: 10^{20} States and Beyond', in Proceedings of the *5th Annual Symposium on Logic in Computer Science*, IEEE, 1990.

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

[22] 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.

[23] Camilleri, A. J., P. Inverardi, and M. Nesi, 'Combining Interaction and Automation in Process Algebra Verification', in Proceedings of the *4th International Joint Conference on the Theory and Practice of Software Development*, Lecture Notes in Computer Science, Springer-Verlag, 1991, Vol. 494, pp. 283–296.

[24] Chou, C.-T., 'A Sequent Formulation of a Logic of Predicates in HOL', in [27], pp. 71–80.

[25] Chou, C.-T., 'A Formal Theory of Undirected Graphs in Higher-Order Logic', in Proceedings of the *7th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Valletta, Malta, 1994, Lecture Notes in Computer Science, Springer-Verlag, Vol. 859, pp. 144–157.

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

[27] Claesen, L. J. M. and M. J. C. Gordon (eds.), Proceedings of the *1992 International Workshop on Higher Order Logic Theorem Proving and Its Applications*, Leuven, September 1992, IFIP Transactions A-20, North-Holland, 1993.

[28] Clarke, E. M., E. A. Emerson, and A. P. Sistla, 'Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications', *ACM Transactions on Programming Languages and Systems* **8**, No. 2, 1986, pp. 244–263.

[29] Clarke, E. M., D. E. Long, and K. L. McMillan, 'Compositional Model Checking', in Proceedings of the *4th Annual Symposium on Logic in Computer Science*, IEEE, 1989.

[30] Cleaveland, R., J. Parrow, and B. Steffen, 'The Concurrency Workbench', in [116], pp. 24–37.

[31] Courcoubetis, C. (ed.), Proceedings of the *5th International Conference on Computer Aided Verification*, Elounda, Greece, Lecture Notes in Computer Science, Springer-Verlag, Vol. 697, 1993.

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

[33] De Nicola, R., A. Fantechi, S. Gnesi, and G. Ristori, 'An action based framework for verifying logical and behavioural properties of concurrent systems', in [82], pp. 37–47.

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

[35] De Nicola, R., P. Inverardi, and M. Nesi, 'Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Specifications', in [116], pp. 54–67.

[36] De Nicola, R., P. Inverardi, and M. Nesi, 'Equational Reasoning about LOTOS Specifications: A Rewriting Approach', in Proceedings of the *6th International Workshop on Software Specification and Design*, Como, IEEE, 1991, pp. 148–155.

[37] Dershowitz, N. and J.-P. Jouannaud, 'Rewrite Systems', in [85], pp. 243–320.

[38] Dershowitz, N., S. Kaplan, and D. A. Plaisted, 'Rewrite, rewrite, rewrite, rewrite, rewrite, ...', *Journal of Theoretical Computer Science* **83**, North-Holland, 1991, pp. 71–96.

[39] De Santis, A. (ed.), Proceedings of the 5*th Italian Conference on Theoretical Computer Science*, Ravello, Italy, World Scientific Publishing Company, 1996.

[40] Doumenc, G., E. Madelaine, and R. de Simone, 'Proving process calculi translations in ECRINS: The PureLOTOS $\mapsto$ MEIJE example', Technical Report 1192, INRIA-Sophia Antipolis, March 1990.

[41] Fernandez, J. C., 'Aldébaran: A tool for verification of communicating processes', Technical Report Spectre C14, LGI–IMAG, Grenoble, 1989.

[42] Fernandez, J. C. and L. Mounier, ' "On the Fly" Verification of Behavioural Equivalences and Preorders', in [82], pp. 181–191.

[43] Formal Systems (Europe) Ltd, 'Failures Divergence Refinement', User Manual and Tutorial, Version 1.2β, 1992.

[44] Garavel, H. and J. Sifakis, 'Compilation and verification of LOTOS specifications', in Proceedings of the 10*th International Symposium on Protocol Specification, Testing and Verification*, Ottawa, 1990, IFIP, North-Holland.

[45] 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.

[46] Gnesi, S. and S. Larosa, 'A sound and complete axiom system for the logic ACTL', in [39], pp. 343–358.

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

[48] Gordon, M., 'Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware', in *Formal Aspects of VLSI Design*, G. Milne and P. A. Subrahmanyam (eds.), North-Holland, 1986.

[49] 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.

[50] 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.

[51] Gordon, M. J. C. and T. F. Melham, '*Introduction to* HOL*: a theorem proving environment for higher order logic*', Cambridge University Press, 1993.

[52] Graf, S. and B. Steffen, 'Compositional Minimization of Finite State Systems', in Proceedings of the *2nd Workshop on Computer Aided Verification*, New Brunswick, New Jersey, 1990, Lecture Notes in Computer Science, Springer-Verlag, Vol. 531, 1991, pp. 186–196.

[53] Groote, J. F. and A. Ponse, 'The Syntax and Semantics of μ-CRL', Technical Report CS-R9076, CWI, Amsterdam, 1990.

[54] Groote, J. F. and A. Ponse, 'Proof Theory for μ-CRL', Technical Report CS-R9130, CWI, Amsterdam, 1991.

[55] Gunter, E. L., 'Why We Can't have SML Style `datatype` Declarations in HOL', in [27], pp. 561–568.

[56] Harrison, J., Private communication, November 1993.

[57] Harrison, J., 'Theory of countable sets', at `http://lal.cs.byu.edu/lal/holdoc/info-hol/17xx/1769.html`.

[58] Hennessy, M., *Algebraic Theory of Processes*, MIT Press, 1988.

[59] Hennessy, M., 'A Proof System for Communicating Processes with Value-Passing', *Formal Aspects of Computing* **3**, 1991, pp. 346–366.

[60] Hennessy, M. and H. Lin, 'Proof Systems for Message-Passing Process Algebras', in Proceedings of *CONCUR '93*, Lecture Notes in Computer Science, Vol. 715, 1993, pp. 202–216.

[61] Hennessy, M. and H. Lin, 'Symbolic bisimulations', *Journal of Theoretical Computer Science* **138**, 1995, pp. 353–389.

[62] Hennessy, M. and H. Lin, 'Unique Fixpoint Induction for Message-Passing Process Calculi', Computer Science Technical Report 95:06, University of Sussex, 1995.

[63] Hennessy, M. and X. Liu, 'A Modal Logic for Message Passing Processes', in [31], pp. 359–370.

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

[65] Hermann, M., 'Chain Properties of Rule Closures', *Formal Aspects of Computing* **2**, 1990, pp. 207–225.

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

[67] Holzmann, G. J., *Design and Validation of Computer Protocols*, Prentice Hall, New Jersey, 1991.

[68] Hungar, H., 'Combining Model Checking and Theorem Proving to Verify Parallel Processes', in [31], pp. 154–165.

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

[70] Inverardi, P. and M. Nesi, 'Deciding Observational Congruence of Finite-State CCS Expressions by Rewriting', *Journal of Theoretical Computer Science* **139**, 1995, pp. 315–354.

[71] Inverardi, P. and M. Nesi, 'Infinite Normal Forms for Non-linear Term Rewriting Systems', *Journal of Theoretical Computer Science* **152**, 1995, pp. 285–303.

[72] Inverardi, P. and C. Priami, 'Evaluation of Tools for the Analysis of Communicating Systems', in Bulletin of the *European Association for Theoretical Computer Science*, No. 45, October 1991, pp. 158–185.

[73] Joyce, J. and C. Seger, 'The HOL-Voss System: Model-Checking inside a General-Purpose Theorem Prover', in [74], pp. 185–198.

[74] Joyce, J. J. and C.-J. H. Seger (eds.), Proceedings of the 6*th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Vancouver, August 1993, Lecture Notes in Computer Science, Springer-Verlag, Vol. 780, 1994.

[75] Kalvala, S., 'A Formulation of TLA in Isabelle', in [113], pp. 214–228.

[76] Kirkwood, C., 'Verification of LOTOS Specifications using Term Rewriting Techniques', Ph.D. Thesis, Technical Report No. FM-94-07, Department of Computing Science, University of Glasgow, July 1994.

[77] Korver, H. and J. Springintveld, 'A Computer-Checked Verification of Milner's Scheduler', in Proceedings of the *International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, Lecture Notes in Computer Science, Springer-Verlag, 1994, Vol. 789, pp. 161–178.

[78] Kozen, D., 'Results on the Propositional μ-Calculus', *Journal of Theoretical Computer Science* **27**, North-Holland, 1983, pp. 333–354.

[79] Kurshan, R. P. and L. Lamport, 'Verification of a Multiplier: 64 Bits and Beyond', in [31], pp. 166–179.

[80] Labella, M. 'Meccanizzazione del linguaggio CCS e della logica ACTL con il sistema HOL', Tesi di Laurea, Department of Computer Science, University of Pisa, 1995.

[81] Larsen, K. G., 'Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion', *Journal of Theoretical Computer Science* **72**, 1990, pp. 265–288.

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

[83] Larsen, K. G. and B. Thomsen, 'A Modal Process Logic', in Proceedings of the *3rd Annual Symposium on Logic in Computer Science*, IEEE, 1988.

[84] Larsen, K. G. and L. Xinxin, 'Equation Solving using Modal Transition Systems', in Proceedings of the *5th Annual Symposium on Logic in Computer Science*, IEEE, 1990, pp. 108–117.

[85] van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, North-Holland, 1990.

[86] Lin, H., 'PAM: A Process Algebra Manipulator', in [82], pp. 136–146.

[87] Lin, H., 'A Verification Tool for Value-Passing Processes', in Proceedings of the *13th IFIP Symposium on Protocol Specification, Testing and Verification*, Liege, 1993.

[88] Madelaine, E., 'Verification Tools from the CONCUR project', in Bulletin of the *European Association for Theoretical Computer Science*, No. 47, June 1992, pp. 110–128.

[89] Mauw, S. and G. J. Veltink, 'A Proof Assistant for PSF', in [82], pp. 158–168.

[90] McMillan, K. L., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[91] 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.

[92] Melham, T. F., 'A Package for Inductive Relation Definitions in HOL', in [3], pp. 350–357.

[93] Melham, T. F., 'A Mechanized Theory of the π-calculus in HOL', *Nordic Journal of Computing* **1**, 1994, pp. 50–76. Also available as Technical Report No. 244, Computer Laboratory, University of Cambridge, 1992.

[94] Melham, T. F. and M. Newey, 'The HOL `sets` Library', in the HOL library documentation, Computer Laboratory, University of Cambridge, January 1995.

[95] Milner, R., 'A Calculus of Communicating Systems', Lecture Notes in Computer Science, Springer-Verlag, Vol. 92, 1980.

[96] Milner, R., 'A Complete Axiomatization for Observational Congruence of Finite-State Behaviours', *Journal of Information and Computation* **81**, 1989, pp. 227–247.

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

[98] Milner R., 'Operational and Algebraic Semantics of Concurrent Processes', in [85], pp. 1201–1242.

[99] Milner, R., J. Parrow and D. Walker, 'A calculus of mobile processes, Parts I and II', *Journal of Information and Computation* **100**, 1992, pp. 1–77.

[100] Nesi, M., 'Mechanizing a Proof by Induction of Process Algebra Specifications in Higher Order Logic', in [82], pp. 288–298.

[101] Nesi, M., 'Formalizing a Modal Logic for CCS in the HOL Theorem Prover', in [27], pp. 495–507.

[102] Nesi, M., 'A Formalization of the Process Algebra CCS in Higher Order Logic', Technical Report No. 278, Computer Laboratory, University of Cambridge, December 1992.

[103] Nesi M., 'Value-Passing CCS in HOL', in [74], pp. 352–365.

[104] Nesi M., 'Reasoning about Value-Passing Calculi in HOL', in [39], pp. 434–450.

[105] Paige, R. and R. E. Tarajan, 'Three Partition Refinement Algorithms', *SIAM Journal of Computing* **16**, No. 6, 1987.

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

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

[108] Paulson, L. C., 'Isabelle: A Generic Theorem Prover', Lecture Notes in Computer Science, Springer-Verlag, Vol. 828, 1994.

[109] Paulson, L. C., 'Mechanising Temporal Reasoning', at `http://www.cl.cam.ac.uk/users/lcp/Temporal`.

[110] Plotkin, G. D., 'A Structural Approach to Operational Semantics', Technical Report DAIMI FN–19, Computer Science Department, Århus University, September 1981.

[111] Quemada, J., S. Pavon and A. Fernandez, 'State Exploration by Transformation with LOLA', in [116], pp. 294–302.

[112] Schubert, E. T., K. Levitt, and G. C. Cohen, 'Formal Mechanization of Device Interactions with a Process Algebra', NASA Contractor Report 189644, Langley Research Center, Hampton, Virginia, November 1992.

[113] Schubert, E. T., P. J. Windley, and J. Alves-Foss (eds.), Proceedings of the *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, 1995, Lecture Notes in Computer Science, Springer-Verlag, Vol. 971.

[114] Sellink, M. P. A., 'Verifying Process Algebra Proofs in Type Theory', Technical Report No. 87, Department of Philosophy, Utrecht University, March 1993.

[115] Shepherd, D. E., Private communication, March 1993.

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

[117] de Simone, R. and D. Vergamini, 'Aboard AUTO', Technical Report 111, INRIA-Sophia Antipolis, 1989.

[118] 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.

[119] Syme, D., 'The Simplifier Library Manual', in the hol90.8 system documentation, pre-release, 1995.

[120] Taubner, D., 'A note on the notation of recursion in process algebras', *Information Processing Letters* **37**, 1991, pp. 299–303.

[121] Turner, K. J. (ed.), *Formal Description Techniques*, Proceedings of *FORTE '88*, North-Holland, 1989.

[122] Victor, B. and F. Moller, 'The Mobility Workbench: A Tool for the π-calculus', in Proceedings of the 6*th Conference on Computer Aided Verification*, Stanford University, Lecture Notes in Computer Science, Springer-Verlag, Vol. 818, 1994.

[123] Walker, D. J., 'Bisimulation equivalence and divergence in CCS', in Proceedings of the 3*rd Annual Symposium on Logic in Computer Science*, IEEE, 1988, pp. 186–192.

[124] Wolper, P. and V. Lovinfosse, 'Verifying Properties of Large Sets of Processes with Network Invariants', in [116], pp. 68–80.

[125] Wong, W., 'The HOL `res_quan` library', in the HOL library documentation, Computer Laboratory, University of Cambridge, February 1993.

[126] von Wright, J. 'Mechanising the Temporal Logic of Actions in HOL', in [3], pp. 155–159.