Basic Research in Computer Science

A Machine Verified Distributed Sorting Algorithm

Jørgen H. Andersen
Ed Harcourt
K.V.S. Prasad

BRICS Report Series **RS-96-4**

ISSN 0909-0878 **February 1996**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

A Machine Verified Distributed Sorting Algorithm

Jørgen H. Andersen
BRICS*
Department of Computer Science
University of Aalborg
Aalborg, Denmark

Ed Harcourt†
K.V.S. Prasad†
Department of Computing Science
Chalmers University of Technology
Göteborg, Sweden

February 15, 1996

Abstract

We present a verification of a distributed sorting algorithm in ALF, an implementation of Martin Löf's type theory. The implementation is expressed as a program in a priortized version of CBS, (the Calculus of Broadcasting Systems) which we have implemented in ALF. The specification is expressed in terms of an ALF type which represents the set of all sorted lists and an HML (Hennesey–Milner Logic) formula which expresses that the sorting program will input any number of data until it hears a value triggering the program to begin outputting the data in a sorted fashion. We gain expressive power from the type theory by inheriting the language of data, state expressions, and propositions.

†Funding from the Swedish Government agency TFR, and ESPRIT BRA CONCUR2
*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

1 Introduction

In this paper we present a machine checked verification of a distributed sorting algorithm in type theory. The verification was done in ALF, an implementation of Martin-Löf's constructive type theory [CNSvS95]. The sorter is expressed in a prioritized version of CBS, The Calculus of Broadcasting Systems [Pra95, Pra94, Pra93] and the specification is expressed as an ALF type that represents the set of sorted lists. We wrap this specification in an HML [Sti93] (Hennessy Milner Logic) formula which provides a layer of abstraction familiar to concurrency theorists.

The verification is interesting because it is done in a value-passing process calculus with a possibly infinite value domain. Even if we restrict ourselves to finite data domains the CBS sorter is still infinite state because it works over lists of arbitrary length. This contrasts with current approaches of doing verification with automatic techniques based on finite-model checking. The price paid is that our verification was done in a proof assistant rather than proved automatically in a theorem prover.

The verification is an induction proof. ALF provides an induction principle for every inductively defined data type. Proofs by induction are useful in verifying properties about infinite state systems. In [Mil89, page 136 section 6.2] Milner proves the correctness of a sorter in CCS by induction (Milner's verification uses bisimulation equivalence and has, to the best of our knowledge, not been machine checked). In our ALF setup a few of the inductively defined sets we use are, the set of natural numbers, lists, CBS processes, and the transition relation that represents the operational semantics of CBS. The induction principles provided by ALF correspond to, respectively, usual mathematical induction over the naturals, induction on the structure of a list, induction on the syntax of processes, and transition induction [Mil89, page 58 section 2.1]. Contrast this with other frameworks such as Isabelle [Pau94], HOL [GM93], and LF [Pfe91] which are based on weaker logics that do not automatically provide induction principles.

To carry out the verification we must have a representation of value-passing CBS in ALF. This representation is interesting in its own right because we borrow the domain of CBS value and state expressions directly from ALF in a way that allows us to identify ALF variables with CBS data variables, making the representation simple and robust. Process substitution and data variable substitution are borrowed from ALF through lambda-

abstraction and function application. We no longer need to include the syntax and semantics of data expressions and we can borrow "off-the-shelf" lemmas about data domains (*e.g.*, that multiplication is commutative). The correctness of the ALF representation is given in [HPP95].

1.1 Outline of the paper

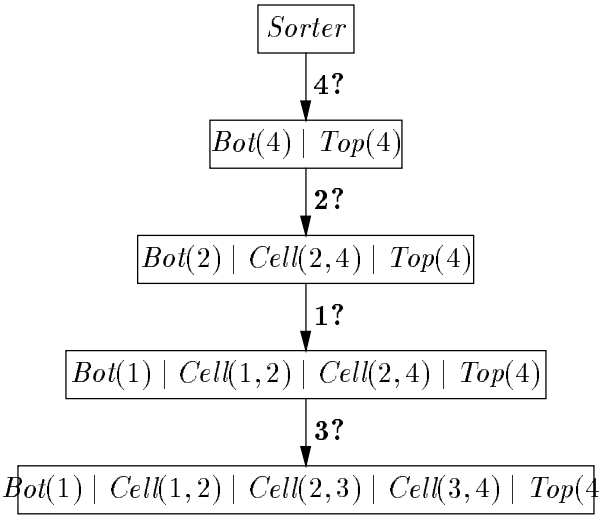
In the rest of this section we explain, informally, the parallel sorting algorithm we verify and discuss what our specification of the sorter is. Section 2 presents, formally, the syntax and semantics of prioritized CBS and describes the parallel sorting algorithm in CBS. Section 3 introduces HML for CBS and gives the HML specification of the sorter. Section 4 introduces ALF by example. Section 5 shows how CBS is presented in ALF and also presents the ALF version of the CBS parallel sorter. Section 6 gives the proof of correctness of the sorter. Section 7 concludes.

1.2 Informal Overview of The Problem

Sorter Implementation. In CBS when a process outputs a value, all of the other processes may hear that value. This *broadcast* is the communication primitive provided by CBS and is quite different than other process algebras where communication occurs between exactly two processes. In prioritized CBS, transitions are tagged with priorities and if two processes try to speak at the same time then the transition with higher priority is taken. If both processes try to speak simultaneously at the same priority, then one is chosen non-deterministically.

The broadcast sort is a kind of parallelized insertion sort. A list of input data is spoken sequentially to the sorter along with a final sentinel value done. In the scope of this paper we only consider naturals as our data domain. The sentinel prohibits the sorter from speaking until all of the input has been read. Two special processes $Bot(n)$ and $Top(m)$ keep track of the smallest integer n and the greatest integer m read so far. If the input list consisted of one integer n the sorter is configured as $Bot(n) | Top(n)$.

In addition to the processes *Bot* and *Top* there are processes $Cell(m, n)$ which keep track of intermediate values. $Cell(m, n)$ has the property that $m \leq n$ and we think of n as the current value of the cell and m as a link to



the cell containing the next lowest value, $Cell(l, m)$. After the list is read, each $Cell(m, n)$ outputs its current value n when it hears m.

As an example, given a list $[4, 2, 1, 3]$ the transition graph to the left traces the configurations of the sorter after each value in the list is spoken (assuming *Sorter* is the initial configuration). After the last configuration there is no more input so *Bot* says its value, 1. When this happens, $Cell(1, 2)$ hears the 1 and then speaks its value 2, and so on resulting in the output trace $[1, 2, 3, 4]$.

Sorter Specification. The sorter outputs a finite list of sorted integers so an obvious choice of a specification is to consider the set of all sorted lists. This set will be defined by an inductively defined relation $\mathsf{Ordered}(\ell)$. Assuming we have a type of lists with the empty list nil, the operation cons, and a relation $\mathsf{Minimal}(n, \ell)$ which says that n is smaller than every element in ℓ then the following rules define $\mathsf{Ordered}$ along with a relation $\mathsf{Sorted}(\ell_1, \ell_2)$ which

$$\frac{}{\mathsf{Ordered}(\mathsf{nil})} \qquad \frac{\mathsf{Minimal}(n, \ell) \quad \mathsf{Ordered}(\ell)}{\mathsf{Ordered}(\mathsf{cons}(n, \ell))}$$

$$\frac{\mathsf{Perm}(\ell_1, \ell_2) \quad \mathsf{Ordered}(\ell_2)}{\mathsf{Sorted}(\ell_1, \ell_2)}$$

says that ℓ_2 is a sorted version of ℓ_1 iff ℓ_2 and ℓ_1 are permutations of each other and ℓ_2 is ordered. We can summarize this into a specification (*SortSpec*) of the *Sorter*.

SortSpec: The *Sorter* will first input a list l of integers followed by the sentinel done. Then it will output the Sorted permutation of l and nothing else.

1.2.1 The Verification

We use a slight variation of CBS which allows the user to specify lists of parallel processes. That is, the tree of parallelism is flattened into a list. We thereby directly inherit the laws of associativity of parallel composition and *Nil* being neutral with parallel composition and the semantics will automatically clean up any extraneous *Nil* processes. Using a list of parallel processes gives us a better handle on doing proofs by induction. Correctness of this version of CBS with CBS as presented in [Pra93] is omitted but is straightforward. The verification we study in the rest of the paper then is that the sorter, called *Sorter* meets the specification Sorted.

2 CBS — Syntax and Semantics

In this section we formally present the syntax and semantics of (prioritized) CBS. As mentioned in the introduction we present a slightly stylized version version of CBS. Rather than expressing parallelism using a binary combinator "|" parallelism pervades the syntax at every level by using *lists* of parallel processes. For example, the process $p|q|r$ becomes $[p, q, r]$. The empty list is the *Nil* process.

Syntax. The syntax of CBS is given by the following grammar.

$$p \ ::= \ \lambda x.[p] \ \Big| \ w!_\pi[p] \ \Big| \ A(s) \tag{1}$$

In the grammar, $[p]$ is a "list of processes", τ is the unique silent action, and α, β are types such that $\tau \notin \alpha, \beta$. The priority π ranges over $\mathbb{N}$ and $w \in \alpha \cup \{\tau\}$. $A(s)$ is a process constant with parameter $s \in \beta$. Associated with A is a set of defining equation of the form $A(x) \stackrel{\text{def}}{=} ps$ where x is of type β and ps a list of processes. **Proc** is the set of processes defined by the above BNF. We regard f as a function from $\alpha \to$ [**Proc**]. The priority 0 is the highest and ∞ is the lowest.

$$\textbf{Pars1}\quad \frac{ps \xrightarrow{w?}_\pi qs \qquad r \overset{w?}{\longmapsto}_\pi rs}{r : ps \xrightarrow{w?}_\pi rs +\!\!+ qs} \qquad \textbf{Pars2}\quad \frac{}{[] \xrightarrow{w?}_\pi []}$$

$$\textbf{Pars3}\quad \frac{qs \xrightarrow{w?}_\pi qs' \qquad p \overset{w!}{\longmapsto}_\pi ps'}{p : qs \xrightarrow{w!}_\pi ps' +\!\!+ qs'} \qquad \textbf{Pars4}\quad \frac{qs \xrightarrow{w!}_\pi qs' \qquad p \overset{w?}{\longmapsto}_\pi ps'}{p : qs \xrightarrow{w!}_\pi ps' +\!\!+ qs'}$$

$$\textbf{Out1}\quad \frac{}{w!_{\pi_1} ps \overset{w'?}{\longmapsto}_{\pi_2} [w!_{\pi_1} ps]}\ \pi_1 \leq \pi_2 \qquad \textbf{Out2}\quad \frac{}{w!_\pi ps \overset{w!}{\longmapsto}_\pi ps}$$

$$\textbf{In1}\quad \frac{}{f \overset{v?}{\longmapsto}_\pi f(v)} \qquad \textbf{In2}\quad \frac{}{f \overset{\tau?}{\longmapsto}_\pi [f]}$$

$$\textbf{Con}\quad \frac{ps[w/s] \xrightarrow{w?}_\pi ps'}{A(s) \overset{w?}{\longmapsto}_\pi ps'}\ A(s) \overset{\text{def}}{=} ps$$

Figure 1: Operational semantics of CBS.

All processes ignore τ at any priority and so idles. Let $v \in \alpha$, $w \in \alpha \cup \{\tau\}$, f be of the form $\lambda x.ps$ and s be of the form $w!_\pi ps$. The process f will hear a value v at any priority and become the list of parallel processes $ps[v/x]$, which means that x is replaced by v for each process in ps. The functional notation allows us to write $ps[v/x]$ as $f(v)$. We avoid the the standard notation of input prefixing $x?p$ not hiding the fact that, in CBS, we identify input prefixing with lambda abstraction. We do, however, assume x ranges over α, so our notation is the sloppy version of: $\lambda(x \in \alpha).[p]$

The process $w!_\pi ps$ can speak w at priority π and evolve to ps. It can hear and ignore any input at priority less than or equal to π. The process $\lambda x.ps$ cannot speak, but it can hear a value from α at any priority. The *Nil* process is the empty list, [], which we write as $\mathbf{0}$.

Semantics. The operational semantics of CBS is given by defining two transition relations, $\longmapsto$ for **Proc** and $\longrightarrow$ for lists of processes. Our definition of $\longrightarrow$ is slightly non-standard in that it identifies transitions $ps \xrightarrow{\alpha}_{\pi} qs$ meaning that the list of parallel processes ps can do an action α and evolve to the list of parallel processes qs. An element process of a list can do an action and evolve to a list of processes identifying transitions $p \overset{w}{\longmapsto}_{\pi} qs$. Formally,

$$\begin{aligned} \longrightarrow &\subseteq [\mathbf{Proc}] \times \alpha \cup \{\tau\} \times \mathbb{N} \times [\mathbf{Proc}] \\ \longmapsto &\subseteq \mathbf{Proc} \times \alpha \cup \{\tau\} \times \mathbb{N} \times [\mathbf{Proc}] \end{aligned}$$

The operational semantics is given in figure 1 where : and $+\!\!+$ are the list *cons* and *append* operators. The rules **Pars1** through **Pars4** describes transitions for lists of parallel processes.

Parallel Broadcast Sort. The broadcast sort explained in the introduction is given in figure 2. The type of the values spoken come from the disjoint union $\texttt{done} \uplus \texttt{num}(\mathbb{N})$ which tags values as either being numbers or the sentinel value `done`. The implementation uses conditional constructs freely (if-then-else and case statements). These are added to the semantics in the obvious way, but as we shall see, need not be as we borrow them from ALF.

The process *Sorter* is the starting process and *Bot* and *Top* are as before. The process *Cell* is split up into two process *InCell* and *OutCell* that describe a cell as being in either an input phase (the input is still being read) or and output phase (the sentinel value has been read). When an *OutCell* hears the (left) number which links it to the previous process it will output it's number with priority 0 if it stores the same number as heard, if not it outputs with priority 1. In this fashion duplicates get higher priority and is spoken before the higher values.

$$
\begin{array}{rcl}
\mathit{Sorter} & \stackrel{\text{def}}{=} & \lambda x.\textbf{case } x \textbf{ of} \\
 & & \quad \texttt{num } x \rightarrow [\mathit{Bot}(x), \mathit{Top}(x)] \\
 & & \quad \texttt{done} \rightarrow [] \\
\mathit{Bot}(n) & \stackrel{\text{def}}{=} & \lambda x.\textbf{case } x \textbf{ of} \\
 & & \quad \texttt{num } x \rightarrow \textbf{if } x \leq n \textbf{ then } [\mathit{Bot}(x), \mathit{InCell}(x,n)] \textbf{ else } [\mathit{Bot}(n)] \\
 & & \quad \texttt{done} \rightarrow n!_1\mathbf{0} \\
\mathit{Top}(n) & \stackrel{\text{def}}{=} & \lambda x.\textbf{case } x \textbf{ of} \\
 & & \quad \texttt{num } x \rightarrow \textbf{if } n < x \textbf{ then } [\mathit{Top}(x), \mathit{InCell}(n,x)] \textbf{ else } [\mathit{Top}(n)] \\
 & & \quad \texttt{done} \rightarrow [] \\
\mathit{InCell}(m,n) & \stackrel{\text{def}}{=} & \lambda x.\textbf{case } x \textbf{ of} \\
 & & \quad \texttt{num } x \rightarrow \textbf{if } m < x \textbf{ and } x \leq n \textbf{ then } [\mathit{InCell}(m,x), \mathit{InCell}(x,n)] \\
 & & \qquad\qquad \textbf{else } [\mathit{InCell}(m,n)] \\
 & & \quad \texttt{done} \rightarrow [\mathit{OutCell}(m,n)] \\
\mathit{OutCell}(m,n) & \stackrel{\text{def}}{=} & \lambda x.\textbf{case } x \textbf{ of} \\
 & & \quad \texttt{num } x \rightarrow \textbf{if } m = x \textbf{ then } \ \textbf{if } m = n \textbf{ then } n!_0\mathbf{0} \\
 & & \qquad\qquad\qquad\qquad\qquad \textbf{else } n!_1\mathbf{0} \\
 & & \qquad\qquad \textbf{else } [\mathit{OutCell}(m,n)] \\
 & & \quad \texttt{done} \rightarrow [\mathit{OutCell}(m,n)]
\end{array}
$$

Figure 2: The Broadcast Sort in CBS.

3 HML

In this section we present the syntax and semantics of our version of HML for CBS. By itself HML is a rather weak logic without recursion or open expressions over the value domain, but is however, powerful in another sense as we can combine HML expressions with ALF expressions and write expressive formulae including a concise sorting specification.

3.1 Syntax & Semantics of HML

Let α be a type, $w \in \alpha_\tau$ and $\vec{w} \in \vec{\alpha_\tau}$ then the syntax *priority abstracted* HML is as follows:

$$f ::= f \wedge f \;\Big|\; f \vee f \;\Big|\; [w?]f \;\Big|\; [w!]f \;\Big|\; \langle w?\rangle f \;\Big|\; \langle w!\rangle f \;\Big|\; \varphi(\vec{w})$$

where φ is a first order logic predicate on $\vec{w} \in \vec{\alpha_\tau}$, such that $\varphi(\vec{w})$ is a closed expression. The satisfiability relation is defined as follows:

$ps \models \varphi(\vec{w})$	$\iff$	$\varphi(\vec{w})$
$ps \models f \wedge g$	$\iff$	$ps \models f$ and $ps \models g$
$ps \models f \vee g$	$\iff$	$ps \models f$ or $ps \models g$
$ps \models [w?]f$	$\iff$	$\forall \pi, qs$ such that $ps \xrightarrow{w?}_\pi qs$ implies $qs \models f$
$ps \models \langle w?\rangle f$	$\iff$	$\exists \pi, qs$ such that $ps \xrightarrow{w?}_\pi qs$ and $qs \models f$
$ps \models [w!]f$	$\iff$	$\forall \pi, qs$ such that $ps \xrightarrow{w!}_\pi qs$ implies $qs \models f$
$ps \models \langle w!\rangle f$	$\iff$	$\exists \pi, qs$ such that $ps \xrightarrow{w!}_\pi qs$ and $qs \models f$

Note that in PCBS all processes can hear (at least with priority 0), hearing is deterministic, and the derived state is independent of the priority at which the transition was made. The query modalities therefore coincide, i.e. $ps \models [w?]f \iff ps \models \langle w?\rangle f$.

3.2 Specifying the *Sorter*

We define the HML *SortSpec* in three steps using two inductively defined formulae. First, we define a formula *InputFormula* (*IF*) parameterized with a list of naturals, xs, and a formula, f. *IF* states that the derived state of a process after hearing xs must satisfy f.

$$
\begin{array}{lcl}
IF & :: & [Nat] \longrightarrow HML \longrightarrow HML \\
IF([], f) & \triangleq & f \\
IF(x : xs, f) & \triangleq & [\mathtt{num}(x)?]\, IF(xs, f)
\end{array}
$$

Second, we define a formula *DeterministicOutputFormula* (*DOF*) parameterized with two lists xs, ys. *DOF* specifies that the output is deterministic. If $xs = ys = []$ then determinism is trivially true. If $xs = x : xs'$ and $ys = y : ys'$ are non-empty a satisfying process must be able to output x and if it can also output y then the two must be equal. Furthermore after outputting the value x any derived process must satisfy *DOF* of xs, ys.

$$
\begin{array}{lcl}
DOF & :: & [Nat] \longrightarrow [Nat] \longrightarrow HML \\
DOF([], []) & \triangleq & \textsc{True} \\
DOF(x : xs', y : ys') & \triangleq & \langle \mathtt{num}(x)! \rangle \textsc{True} \wedge [\mathtt{num}(y)!](x \neq y) \wedge \\
 & & [\mathtt{num}(x)!]\, DOF(xs', ys')
\end{array}
$$

Third, to specify sorting we wrap these two formulae around a sorting function, Sort. The function Sort is a functional version of the relation Sorted given in the introduction and, as we mentioned, we omit the proof that Sort implements Sorted. *SortSpecFormula* (*SSF*) states that after inputting some list xs and the special value `done` a satisfying process must output the sorted version of xs and not some other list ys.

$$
\begin{array}{lcl}
SSF & :: & [Nat] \longrightarrow [Nat] \longrightarrow HML \\
SSF(xs, ys) & \triangleq & IF(xs, [\mathtt{done}?]\, DOF(\textsc{Sort}(xs), ys))
\end{array}
$$

Instead of just checking the *Sorter* $\models$ *SSF(xs,ys)* for two specific lists our goal is to prove that *Sorter* meets our specification for all list:

$$
(*) \qquad \forall xs, ys : Sorter \models SortSpecFormula(xs, ys)
$$

4 Representation in ALF

Before we proceed with the verification we informally present ALF through a few examples on the natural numbers (This short discussion of type theory is taken from [CNSvS95]). We view ALF as the typed lambda calculus extended with dependent types. ALF is a proof editor and all of the ALF code given here appears as it does on the screen. There are two kinds of terms in ALF — types and objects.

Natural Numbers. The type (set) of natural numbers is introduced with the definition $\mathsf{N} \in \mathsf{Set}$, $0 \in \mathsf{N}$, and $\mathsf{s} \in (\mathsf{N})\mathsf{N}$, the latter being constructors for zero and the successor of a natural. Here the type $(\mathsf{N})\mathsf{N}$ is ALF notation for the function type $\mathsf{N} \to \mathsf{N}$. An object (*i.e.*, function) Add that adds two natural numbers and a set (or type) Le representing a relation for $\leq$ are defined by the following (which is how they actually appear in the ALF proof editor).

$\mathsf{Add} \in (\mathsf{N};\mathsf{N})\mathsf{N}$ $\mathsf{Add}(0,y) = y$ $\mathsf{Add}(\mathsf{s}(x),y) = \mathsf{s}(\mathsf{Add}(x,y))$	$\mathsf{Le} \in (m,n \in \mathsf{N})\mathsf{Set}$ $\mathsf{leO} \in (n \in \mathsf{N})\mathsf{Le}(0,n)$ $\mathsf{leS} \in (m,n \in \mathsf{N};\mathsf{Le}(m,n))\mathsf{Le}(\mathsf{s}(m),\mathsf{s}(n))$

The definition of Le follows the normal relational definition. Notice the use of the dependent function type. In Le the constructor leO is a function whose result type $\mathsf{Le}(0,n)$ depends on the object n. This allows us to define Le as would be done in an operational semantics and hints at how the operational semantics of CCS will be specified in ALF, as an inductively defined relation. That is, leO encodes the rule $\frac{}{0 \leq n}$ and leS encodes $\frac{n \leq m}{\mathsf{s}(n) \leq \mathsf{s}(m)}$.

Types as Propositions. A function in ALF can be viewed as a proof of a proposition in first-order logic where the type of the function represents the proposition to be proved. For example, the following function is a proof that Le is transitive. In the definition the first three parameters $i,j,k \in \mathsf{N}$ have been hidden along with the declarations of m and n in the constructors for Le. This feature of ALF makes proofs more readable.

$$\begin{array}{l}\mathsf{LeTrans} \in (\mathsf{Le}(i,j);\mathsf{Le}(j,k))\mathsf{Le}(i,k)\\ \mathsf{LeTrans}(\mathsf{leO},h) = \mathsf{leO}\\ \mathsf{LeTrans}(\mathsf{leS}(h_2),\mathsf{leS}(h)) = \mathsf{leS}(i,k,\mathsf{LeTrans}(h_2,h))\end{array}$$

The type of the function represents the proposition to be proved and the body of the function represents the proof. The function is recursive, which represents a proof by induction.

Equality in ALF. In ALF there is a class of objects denoted by a relation Id where $\mathsf{Id} \in (A \in \mathsf{Set}; A; A)$ that are equivalent up to ALF'S $\alpha\beta\eta$-reduction. This relation plays an important role in our representation as it will allow us to identify closed expressions with their values.

5 PCBS & HML – ALF style

We now describe how we have defined PCBS and HML in ALF and how we use these definitions to implement the *Sorter* and the sorting specification. First we represent PCBS and implement *Sorter*, then we represent HML and give the sorting specification.

5.1 PCBS — ALF style

The syntax of PCBS is represented in a type Proc(A,S), where A is the set of actions Act and we let Act_τ be represented by the lifted domain Lift(A) where bot(A) represents τ. An element $a \in \mathsf{A}$ is now denoted in(a). Instead of giving the ALF syntax as it looks in ALF we give the translation of the standard syntax into that of our implementation.

$w!_k ps$	SAY(triple(w, k, ps))
$x?ps$	LISTEN(fun($[x]ps$))
s	VAR(s)

Here fun($[x]y$) constructs a function of type (A)B where $x \in \mathsf{A}$ and $y \in \mathsf{B}$ and $[x]y$ is ALF notation for the lambda abstraction $\lambda x.y$. Note that the conditional is not part of the CBS syntax. This is not a problem as we borrow it from ALF. We define **If-Then-Else** as a function, Cond.

$$\begin{array}{l}
\mathsf{Cond} \in (b \in \mathsf{Bool}; ps_1, ps_2 \in \mathsf{List(Proc(A,S))})\mathsf{List(Proc(A,S))} \\
\mathsf{Cond}(tt, ps_1, ps_2) = ps_1 \\
\mathsf{Cond}(ff, ps_1, ps_2) = ps_2
\end{array}$$

The transition relation is defined by two separate relations, one for listening and one for speaking. The listening relation $p \xrightarrow{w?}_k qs$ is defined by

$$\begin{array}{rl}
\mathsf{Query} \in & (env \in \mathsf{(S)List(Proc(A,S))}; p \in \mathsf{Proc(A,S)}; \\
& w \in \mathsf{Lift(A)}; k \in \mathsf{Nat}; qs \in \mathsf{Proc(A,S)})\mathsf{Set} \\
\mathsf{PQuery} \in & (env \in \mathsf{(S)List(Proc(A,S))}; ps \in \mathsf{List(Proc(A,S))}; \\
& w \in \mathsf{Lift(A)}; k \in \mathsf{Nat}; qs \in \mathsf{List(Proc(A,S))})\mathsf{Set}
\end{array}$$

where Query and PQuery defines the relations $p \xrightarrow{w?}_k qs$ and $ps \xrightarrow{w?}_k qs$ from the operational semantics. The relations ($p \xrightarrow{w!}_k qs$ and $ps \xrightarrow{w!}_k qs$) are similarly defined in Speak and PSpeak.

To implement *Sorter* from Figure 2 in ALF first we define the set of actions Act and the set of process constants State.

$$\begin{array}{lcl}
\mathsf{Act} & \in & \mathsf{Set} \\
\mathsf{done} & \in & \mathsf{Act} \\
\mathsf{num} & \in & \mathsf{(Nat)Act}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{State} & \in & \mathsf{Set} \\
\mathsf{sorter} & \in & \mathsf{State} \\
\mathsf{botcell} & \in & \mathsf{(Nat)State} \\
\mathsf{topcell} & \in & \mathsf{(Nat)State} \\
\mathsf{incell} & \in & \mathsf{(Nat; Nat)State} \\
\mathsf{outcell} & \in & \mathsf{(Nat; Nat)State}
\end{array}$$

The *Sorter* is then implemented in a function SorterEnv of type Proc(Act,State). SorterEnv maps process constants to PCBS processes. The function sgl maps an object to the singleton list.

$$\begin{array}{l}
\mathsf{SorterEnv} \in \mathsf{(State)List(Proc(Act,State))} \\
\mathsf{SorterEnv(sorter)} = \mathsf{sgl(LISTEN(fun([}a\mathsf{]sorterCase}(a)\mathsf{))))} \\
\mathsf{SorterEnv(botcell}(a_1)\mathsf{)} = \mathsf{sgl(LISTEN(fun([}a\mathsf{]botCase}(a, a_1)\mathsf{))))} \\
\mathsf{SorterEnv(topcell}(a_1)\mathsf{)} = \mathsf{sgl(LISTEN(fun([}a\mathsf{]topCase}(a, a_1)\mathsf{))))} \\
\mathsf{SorterEnv(incell}(a_1, a_2)\mathsf{)} = \mathsf{sgl(LISTEN(fun([}a\mathsf{]incellCase}(a, a_1, a_2)\mathsf{))))} \\
\mathsf{SorterEnv(outcell}(a_1, a_2)\mathsf{)} = \mathsf{sgl(LISTEN(fun([}a\mathsf{]outcellCase}(a, a_1, a_2)\mathsf{))))}
\end{array}$$

We show the two cases incellCase and outcellCase which are direct translation from figure 2.

```
incellCase ∈ (Act;Nat;Nat)List(Proc(Act,State))
incellCase(done, n1, n2) = sgl(VAR(outcell(n1, n2)))
incellCase(num(n1), n2, n3) =
  Cond(bool_and(bool_lt(n2, n1), bool_leq(n1, n3)),
    cons(VAR(incell(n2, n1)), sgl(VAR(incell(n1, n3)))),
    sgl(VAR(incell(n2, n3))))
```

```
outcellCase ∈ (Act;Nat;Nat)List(Proc(Act,State))
outcellCase(done, n1, n3) = sgl(VAR(outcell(n2, n3)))
outcellCase(num(n1), n2, n3) =
  Cond(bool_eq(n1, n2),
    Cond(bool_eq(n2, n3),
      sgl(SAY(triple(in(num(n3)), 0, nil))),
      sgl(SAY(triple(in(num(n3)), s(0), nil)))),
    sgl(VAR(outcell(n2, n3))))
```

In a similar fashion we implement topCase, botCase and sorterCase. We now proceed to implementing the sorting specification in ALF.

5.2 HML — ALF style

The HML syntax from section 3 is mapped to ALF terms by the following translation.

TRUE	True
FALSE	False
$f \wedge g$	And(f, g)
$f \vee g$	Or(f, g)
$[w?]f$	BoxQuery$(w)f$
$\langle w? \rangle f$	DiaQuery$(w)f$
$[w!]f$	BoxSpeak$(w)f$
$\langle w! \rangle f$	DiaSpeak$(w)f$

The modal logic we use in Section 3 is seemingly more powerful as we can use first-order logic predicates in the specifications. Here we are left with only

the propositional variables True and False. That is, every HML expression is closed and each first-order predicate can be identified with either True or False. To inherit the expressive power of ALF we simply move the first order logic expressions, φ to $\models$ in the following way.

$$(**) \qquad p \models [w!]\varphi \quad \Leftrightarrow \quad \text{if } \neg\varphi \text{ then } p \models [w!]\textsc{False}$$

The satisfiability relation $\models$ is defined between lists of PCBS processes and HML formulae given the process' environment.

Sat $\in$ ($env \in$ (S)List(Proc(A,S)); $ps \in$ List(Proc(A,S)); $f \in$ HML(A))Set

We show constructors for two cases, $p \models [w!]f$ and $p \models \langle w! \rangle f$.

SatDiaSpeak $\in$	(PSpeak(env, ps, w, n, qs); Sat(env, qs, f))
	Sat(env, ps, DiaSpeak(w, f))
SatBoxSpeak $\in$	(($n \in$ Nat; $qs \in$ List(Proc(A,S)); PSpeak(env, ps, w, n, qs))
	Sat(env, qs, f))Sat(env, ps, BoxSpeak(w, f))

We can now give our sorting specification in terms of HML. *DOF* from section 3.2 is redefined using $(**)$. That is, we assume a proof $neq \in$ Id(x, y)Empty which means $(x \neq y)$ [1]. To do this we first define a relation *DistinctElemList* in DEL which relates a list vs with a list nvs of equal length where the elements are pairwise distinct. The input formula *IF* is represented in IF, *DOF* in DOF and the entire specification in SSF.

DEL $\in$	($vs, nvs \in$ List(A))Set
distinct0 $\in$	DEL(nil,nil)
distinctR $\in$	($neq \in$ (Id(v, nv))Empty; $dist \in$ DEL(vs, nvs))
	DEL(cons(v, vs), cons(nv, nvs))

In the SSF we have replaced the function Sort with a relation, Sorted from the introduction. This relation and the DistinctElemList provide us with the induction basis when we, in section 6, prove that *Sorter* satisfies *SortSpec*.

[1] Empty $\in$ Set is a set without any constructors, i.e. no objects of type Empty can exist.

$$
\begin{array}{l}
\mathsf{IF} \in (\mathsf{List(Nat)}; f \in \mathsf{HML(Act)})\mathsf{HML(Act)} \\
\mathsf{IF}(\mathsf{nil}, f) = f \\
\mathsf{IF}(\mathsf{cons}(v, vs), f) = \mathsf{BoxQuery}(\mathsf{in}(\mathsf{num}(v)), \mathsf{IF}(vs, f)) \\
\mathsf{DOF} \in (dist \in \mathsf{DEL}(svs, nsvs))\mathsf{HML(Act)} \\
\mathsf{DOF}(\mathsf{distinct0}) = \mathsf{True} \\
\mathsf{DOF}(\mathsf{distinctR}(neq, dist1)) = \\
\quad \mathsf{And}(\mathsf{And}(\\
\quad \mathsf{DiaSpeak}(\mathsf{in}(\mathsf{num}(v)), \mathsf{True}), \\
\quad \mathsf{BoxSpeak}(\mathsf{in}(\mathsf{num}(nv)), \mathsf{False})), \\
\quad \mathsf{BoxSpeak}(\mathsf{in}(\mathsf{num}(v)), \mathsf{DOF}(dist_1))) \\
\mathsf{SSF} \in (ins \in \mathsf{Sorted}(vs, svs); dist \in \mathsf{DEL}(svs, nsvs))\mathsf{HML(Act)} \\
\mathsf{SSF}(ins, dist) = \mathsf{IF}(vs, \mathsf{BoxQuery}(\mathsf{in}(\mathsf{done}), \mathsf{DOF}(dist))
\end{array}
$$

6 Verification in ALF

Recall the HML requirement $(*)$ in Section 3.2. This requirement translates into ALF as follows.

$$
\begin{array}{ll}
\mathsf{SorterSat} \in & (ins \in \mathsf{Sorted}(vs, svs); dist \in \mathsf{DEL}(svs, nsvs)) \\
& \mathsf{Sat}(\mathsf{SorterEnv}, \mathsf{sgl}(\mathsf{VAR}(\mathsf{sorter})), \mathsf{SSF}(ins, dist))
\end{array}
$$

This is not an easy requirement to prove directly in ALF. To help us we will prove a variety of lemmas that characterize the *Sorter*.

6.1 Characterising the *Sorter*

The first step in proving the *Sorter* correct is finding something to do induction on. The *Sorter* has two phases, an input phase and an output phase. In either phase the sorter satisfies similar properties for every state. In the input phase the sorter will input a number and evolve to a process still in the input phase. If it at any state in the input phase inputs the value done it will evolve to a process in the output phase. Likewise, all processes in the output phase have similar behavior. They will all output the minimum value stored. Recognising these facts we make a "syntax for the sorter" in either phase. The following BNF represents the sorter in its input phase.

ISorter nil	::=	$[Sorter]$
ISorter $x : xs$	::=	$Bot(x)$: ITop x xs
ITop x nil	::=	$[Top(x)]$
ITop x $y : ys$	::=	$InCell(x, y)$: ITop y ys

Given a sorted list ISorter describes a derived state of the *Sorter* in the input phase. The first parameter to ITop is the link to the last *InCell*. We use this syntax to implement the following predicate ISorter in ALF, however, we require the list to be sorted. We only give the types for ISorter and ITop.

ISorter ∈	(List(Nat); List(Proc(Act,State)))Set
ITop ∈	(Nat; List(Nat); List(Proc(Act,State)))Set

Similarly, we make a syntax describing the derived states of the *Sorter* in the output phase. However, here we will have to distinguish the set of initial states (the set of states after the *Sorter* has heard a done) and the set of derived states.

OSorterInit nil	::=	nil
OSorterInit $x : xs$	::=	$x!_1\, nil$: OTop x xs
OSorter x nil	::=	nil
OSorter x $x : xs$	::=	$x!_0\, nil$: OSorter x xs
OSorter $_{(x \neq y)} x$ $y : ys$	::=	$y!_1\, nil$: OTop y ys
OTop x nil	::=	nil
OTop x $y : ys$	::=	$OutCell(x, y)$: OTop y ys

Again, given a sorted list the OSorterInit syntax describes a derived state of the *Sorter* after hearing done. The OSorter syntax describes the process after it has spoken the value given by the first parameter. OTop describes the remaining *OutCell*'s where the first parameter is the link to the previously spoken value.

OSortInit ∈	(List(Nat); List(Proc(Act,State)))Set
OSorter ∈	(Nat; List(Nat); List(Proc(Act,State)))Set
OTop ∈	(Nat; List(Nat); List(Proc(Act,State)))Set

These syntax definitions provide the extra structure needed to do induction. We state a few of the lemmas that are used to prove the high-level sorting specification. By Lemma 1 we establish that processes in the input phase stay in the input phase after inputting a number.

Lemma 1 *Let* $ps, qs ::$ [Proc(Act,State)], $v, \pi \in \mathbb{N}$, $vxs, xs ::$ [$\mathbb{N}$]. *If* Sorted(xs), $ps \in$ ISorter xs, insert$(v, xs) = vxs$ *and* $ps \xrightarrow{v}_{\pi} qs$ *then* $qs \in$ ISorter vxs

Lemma 2 states that if a process in the input phase hears the value done the derived process is given by the syntax of initial processes in the output phase.

Lemma 2 *Let* $ps, qs ::$ [Proc(Act,State)], $\pi \in \mathbb{N}$, $xs ::$ [$\mathbb{N}$]. *If* Sorted(xs), $ps \in$ ISorter xs *and* $ps \xrightarrow{Done}_{\pi} qs$ *then* $qs \in$ OSortInit xs

If a process given by the syntax of initial output processes outputs a number the derived state is a process of the OSorter syntax. This is stated in Lemma 3.

Lemma 3 *Let* $ps, qs ::$ [Proc(Act,State)], $v, \pi \in \mathbb{N}$, $xs ::$ [$\mathbb{N}$]. *If* Sorted(xs), $ps \in$ OSortInit xs, *and* $ps \xrightarrow{v!}_{\pi} qs$ *then* $qs \in$ OSorter $head(xs)$ $tail(xs)$

Lastly, by lemma 4, if a process is given by OSorter and it outputs a number, the derived state will also be given by OSorter.

Lemma 4 *Let* $ps, qs ::$ [Proc(Act,State)], $v, \pi \in \mathbb{N}$, $xs ::$ [$\mathbb{N}$]. *If* Sorted$(v :: xs)$, $ps \in$ OSorter v xs, *and* $ps \xrightarrow{v!}_{\pi} qs$ *then* $qs \in$ OSorter $head(xs)$ $tail(xs)$

Note in lemmas 3 and 4 that if xs is empty, then ps must be nil and will therefore not be able to output. These lemmas plus some auxiliary lemmas that the *Sorter* deterministically outputs numbers in make up a verification of the *Sorter*. What remains is taking these lemmas and apply them in the proof of our more high level HML specification.

6.2 $Sorter \models SortSpec$

We can now prove satisfaction in three steps. First we prove that any process satisfies $\mathsf{IF}(xs, f)$ if the derived state after hearing a sequence of numbers satisfies f. ListPQuery is a priority abstracted relation of hearing a list numbers.

$$\begin{array}{l} \mathsf{SatIF} \in \quad ((qs \in \mathsf{List}(\mathsf{Proc}(\mathsf{Act},\mathsf{State})); \mathsf{ListPQuery}(ps, vs, qs)) \\ \qquad\quad \mathsf{Sat}(\mathsf{SorterEnv}, qs, f))\mathsf{Sat}(\mathsf{SorterEnv}, ps, \mathsf{IF}(vs, f)) \end{array}$$

Second, by Lemmas 3 and 4 we know that the derived state of a process in OSortInit or OSorter after outputting a number is a process in OSorter. Also, we have proven auxiliary lemmas stating that output is deterministic. Using these we can prove that processes in the output phase satisfy the DOF.

$$\begin{array}{l} \mathsf{OSortInitSatDOF} \in \quad (\mathsf{OSortInit}(\mathsf{svs},\mathsf{ps}); dist \in \mathsf{DEL}(svs, nsvs)) \\ \qquad\qquad\quad \mathsf{Sat}(\mathsf{SorterEnv}, ps, \mathsf{DOF}(dist, \mathsf{True})) \end{array}$$

$$\begin{array}{l} \mathsf{OSorterSatDOF} \in \quad (\mathsf{OSorter}(v, svs, ps); dist \in \mathsf{DEL}(svs, nsvs)) \\ \qquad\qquad\quad \mathsf{Sat}(\mathsf{SorterEnv}, ps, \mathsf{DOF}(dist, \mathsf{True})) \end{array}$$

Third, we combine the above two proofs. We establish by lemma 1 that the derived state after hearing a sequence of numbers will also be in the input phase. By lemma 2 we know the derived state after hearing done is given by OSortInit with the parameter list being the sorted version of the input list. This yields the desired satisfaction result.

$$\begin{array}{l} \mathsf{SorterSat} \in \quad (ins \in \mathsf{Sorted}(vs, svs); dist \in \mathsf{DEL}(svs, nsvs)) \\ \qquad\qquad\quad \mathsf{Sat}(\mathsf{SorterEnv}, \mathsf{sgl}(\mathsf{VAR}(\mathsf{sorter})), \mathsf{SSF}(ins, dist)) \end{array}$$

7 Conclusions

Proofs done by pencil and paper are often incorrect whereas proofs that are machine checked are guaranteed to be correct modulo the correctness of the implementation of the theorem prover or proof checker. Systems that are finite state can be verified using automatic techniques whereas most infinite state systems cannot be and must be "proof checked". In this paper we presented a machine checked proof of a distributed sorting algorithm. The implementation was done in CBS, the Calculus of Broadcasting Systems and

the specification was written in HML, Hennessy-Milner Logic. The verification itself was done in ALF, an implementation of constructive type theory. To our knowledge this is the first machine checked proof of a parallel sorting algorithm done in either a process calculus such as CBS or in a modal logic such as HML.

One thing worth emphasising about the proof is that proving *directly* that the CBS sorter is correct with respect to a high-level sorting specification is difficult. We first expressed the sorter in an equivalent version of CBS where parallelism is expressed as lists of processes rather than with a binary combinator. This technique allowed us to omit repeated applications of the law for associativity of parallel composition and the law that the *Nil* process is a zero for parallel composition.

Also, the initial sorting specification was too abstract to be used directly and in the verification we used one that was more concrete. Of course, we had to verify that this new specification was correct with respect to the abstract specification (which was done but omitted for lack of space). As one would expect in a large proof[2], the verification was reduced to proving a series of lemmas about the sorter and then combining these lemmas into a complete verification.

The actual ALF–proof is available by contacting *yogi@iesd.auc.dk*

References

[CNSvS95] Thierry Coquand, Bengt Nordström, Jan Smith, and Björn von Sydow. Type theory and programming. Extended Abstracts in Theoretical Computer Science bulletin number 52, February 1995. Also available Chalmers University of Technology technical report 81.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[HPP95] Ed Harcourt, Pawel Paczkowski, and K.V.S. Prasad. A framework for representing parameterized processes. In preparation.

[2]The ALF representation was well over 2300 *plus* imported theory

Department of Computing Science, Chalmers University of Technology, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Pau94] Lawrence Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag, 1994. LNCS 828.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[Pra93] K. V. S. Prasad. Programming with broadcasts. In *CONCUR*, August 1993. Springer Verlag LNCS 715.

[Pra94] K. V. S. Prasad. Broadcasting with priority. In *ESOP*, April 1994. Springer Verlag LNCS 788.

[Pra95] K.V.S. Prasad. A calculus of broadcasting systems. *The Science of Computer Programming*, 1995. To apprear.

[Sti93] Colin Stirling. Modal and temporal logics. In Samson Abramsky, Don Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 478–563. Oxford: Clarendon Press, 1993.

Recent Publications in the BRICS Report Series

RS-96-4 **Jørgen H. Andersen, Ed Harcourt, and K.V.S. Prasad.** ***A Machine Verified Distributed Sorting Algorithm*****. February 1996. 21 pp. Abstract appeared in** ***7th Nordic Workshop on Programming Theory*****, NWPT '7 Proceedings, 1995.**

RS-96-3 **Jaap van Oosten.** ***The Modified Realizability Topos*****. February 1996. 17 pp.**

RS-96-2 **Allan Cheng and Mogens Nielsen.** ***Open Maps, Behavioural Equivalences, and Congruences*****. January 1996. A short version of this paper is to appear in the proceedings of CAAP '96.**

RS-96-1 **Gerth Stølting Brodal and Thore Husfeldt.** ***A Communication Complexity Proof that Symmetric Functions have Logarithmic Depth*****. January 1996. 3 pp.**

RS-95-60 **Jørgen H. Andersen, Carsten H. Kristensen, and Arne Skou.** ***Specification and Automated Verification of Real-Time Behaviour — A Case Study*****. December 1995. 24 pp. Appears in** ***3rd IFAC/IFIP workshop on Algoritms and Architectures for Real-Time Control*****, AARTC '95 Proceedings, 1995, pages 613–628.**

RS-95-59 **Luca Aceto and Anna Ingólfsdóttir.** ***On the Finitary Bisimulation*****. November 1995. 29 pp.**

RS-95-58 **Nils Klarlund, Madhavan Mukund, and Milind Sohoni.** ***Determinizing Asynchronous Automata on Infinite Inputs*****. November 1995. 32 pp.**

RS-95-57 **Jaap van Oosten.** ***Topological Aspects of Traces*****. November 1995. 16 pp.**

RS-95-56 **Luca Aceto, Wan J. Fokkink, Rob J. van Glabbeek, and Anna Ingólfsdóttir.** ***Axiomatizing Prefix Iteration with Silent Steps*****. November 1995. 25 pp.**

RS-95-55 **Mogens Nielsen and Kim Sunesen.** ***Trace Equivalence - Partially Decidable!*** **November 1995.**