Mechanically Checked Proofs of Kernel Specifications*‡

William R. Bevier§ Jørgen F. Søgaard-Andersen¶

Abstract

This paper describes an experiment in the use of the Boyer-Moore logic to specify a non-finite state operating system kernel, and in the use of the Boyer-Moore theorem prover to prove the correctness of an implementation. The kernel specification had first been given in terms of a labeled transition system. It was transcribed into the Boyer-Moore logic so that an attempt could be made to mechanically check correctness proofs.

Keywords: Kernel, mechanical proof checking, Boyer-Moore Theorem Prover, stepwise development, labeled transition systems, safety properties.

1 Introduction

An approach to specifying a multiprogramming kernel is given in [8]. It describes several levels of abstraction in the specification of a kernel implementing occam2–like [4] processes on a single machine. It also explores the question of what it means for one level to be a correct implementation of another. The underlying semantic model used in [8] is the well-known notion of labeled transition systems (see section 2 below).

This paper describes an attempt to use the Boyer-Moore logic to state the kernel specifications, and use the Boyer-Moore theorem prover to mechanically check proofs of correctness of kernel levels. The Boyer-Moore approach was chosen largely because of its use in an earlier kernel specification and implementation project described in [1].

Section 2 of this paper briefly describes the notion of labeled transition systems and what it means for one transition system to be a safe implementation of another. Section 3 describes the kernel specification given in [8]. The translation of the specification into the Boyer-Moore logic is discussed in Section 4. The correctness theorems are presented in Section 5. Section 6 contains some observations on this exercise.

*This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

‡This work was also supported in part at the Technical University of Denmark by the Commision of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Action proj. no. 3104: "ProCoS: Provably Correct Systems" and by The Danish Technical Research Council under the "RapID" programme.

§Computational Logic Inc., 1717 W. 6th St. Suite 290, Austin, Texas 78703, email: bevier@cli.com

¶Department of Computer Science, Building 344, Technical University of Denmark, DK-2800 Lyngby, Denmark, email: jsa@id.dth.dk

This paper contains no introduction to the Boyer-Moore logic or its theorem prover. See [3] for this information. Anyone familiar with Lisp should have little problem following the presentation. We occasionally give what we hope are helpful footnotes.

2 Labeled Transition Systems

In [8] several levels of abstraction in the specification of a multiprogramming kernel are given semantics in the style of Plotkin's Structural Operational Semantics, SOS [7].

The method uses *labeled transition systems* (LTS) as the underlying semantic model. This is a way of describing the steps of a computer program during its execution and captures the intuitive understanding of a program as transitions between states. Formally, an LTS, $\mathcal{S}^\alpha$, where α is the program, is a quadruple $(\Gamma^\alpha_{\mathcal{S}}, \mathsf{I}^\alpha_{\mathcal{S}}, \Lambda^\alpha_{\mathcal{S}}, \xrightarrow[\mathcal{S}^\alpha]{})$, where $\Gamma^\alpha_{\mathcal{S}}$ is the set of states, $\mathsf{I}^\alpha_{\mathcal{S}}$ is the set of initial states, $\Lambda^\alpha_{\mathcal{S}}$ is the set of labels, and $\xrightarrow[\mathcal{S}^\alpha]{}$ is the transition relation ($\xrightarrow[\mathcal{S}^\alpha]{} \subseteq \Gamma^\alpha_{\mathcal{S}} \times \Lambda^\alpha_{\mathcal{S}} \times \Gamma^\alpha_{\mathcal{S}}$). A transition in $\mathcal{S}^\alpha$ is usually written $\alpha \vdash s \xrightarrow[\mathcal{S}]{\lambda} s'$ which denotes a transition from the state s to s' in the context of the program code α. λ denotes the nature of the step. We shall return to this. Below we let α be a given program and omit it as index on LTSs and transitions.

2.1 The Correctness Notion

Since each level of abstraction in the kernel specification is described by a LTS, correctness of each step of development is expressed as a refinement relation between LTSs. In [8] this relation is divided into both a safety and a fairness part. Here we concentrate on the safety part based on simulation between LTSs[1]. We say that a step goes from an *abstract* LTS to a *concrete* LTS (which is then abstract wrt. the next development step).

Part of the safety correctness notion deals with showing correspondence between concrete and abstract transitions. Since we are only interested in investigating concrete transitions emanating from reachable states, it suffices to show the correspondence for transitions emanating from states satisfying an arbitrary concrete invariant[2]. To relate concrete and abstract states the correctness notion requires the existence of an *abstraction function*, $\mathcal{R}$, mapping states of the concrete LTS to states of the abstract LTS.

Definition 1 (SAFE implementation)

A concrete LTS $\mathcal{C}$ is a **safe implementation** *of an abstract LTS $\mathcal{A}$ if an abstraction function $\mathcal{R}$ and a concrete invariant $\mathcal{I}_\mathcal{C}$ exist such that the following conditions hold:*

(i) $(\forall s \in \mathsf{I}_\mathcal{C})(\mathcal{R}(s) \in \mathsf{I}_\mathcal{A})$

(ii) $(\forall s, s' \in \Gamma_\mathcal{C}, \lambda \in \Lambda_\mathcal{C})$
$(\mathcal{I}_\mathcal{C}(s) \wedge s \xrightarrow[\mathcal{C}]{\lambda} s' \Longrightarrow (\lambda \in \Lambda_\mathcal{A} \wedge \mathcal{R}(s) \xrightarrow[\mathcal{A}]{\lambda} \mathcal{R}(s')) \vee (\lambda \notin \Lambda_\mathcal{A} \wedge \mathcal{R}(s) = \mathcal{R}(s')))$

■

[1] Since the presentation in [8] includes a step of compilation, the correctness notion given there is more general than the one used here.

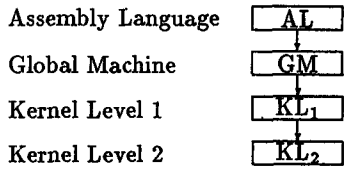
[2] An invariant is a predicate which is satisfied by all states reachable from an initial state.

Informally, the definition states that (i) all initial concrete states must have a corresponding initial abstract state, and (ii) each concrete transition (emanating from a state satisfying the invariant) with a label that exists at the abstract level must have an abstract counterpart, whereas new concrete transitions (indicated by new labels) must not change the abstract state.

Since Definition 1 only requires *simulation* and not *bisimulation* [6] between the LTSs, an LTS with an empty transition relation is a safe implementation of any LTS. This inconvenience can be taken care of by introducing fairness into the correctness notion, as it is done in [8].

3 A Kernel Specification

Several levels of abstraction in the specification of the kernel, implementing multiple processes on a single machine, are described in [8]. The diagram below depicts some of the levels ranging from the most abstract at the top to the most concrete at the bottom.



The Assembly Language level gives the abstract semantics of an assembly language where each process is considered to be running on its own machine. The Global Machine gives the semantics of processes running on one machine but without any explicit scheduling. The two Kernel Levels introduce kernel aspects like *current process*, *ready queue*, etc.

This section describes parts of the GM and KL_1 levels. These are given with a high degree of detail in order to show how the subsequent translation into Boyer-Moore logic corresponds to the specification given here.

We first introduce an assembly language for a stack machine.

3.1 A Sample Assembly Language, SAL

An assembly language program is a (nonempty) list of process codes, each of which is a list of instructions. The instructions are partitioned into *kernel instructions* dealing with communication, and *private instructions* like `jump` etc. The instructions are inspired by the Transputer instruction set [5], i.e. we have synchronous communication on channels.

SAL	=	$Process^+$	(1)
Process	=	Ins^*	(2)
Ins	=	*KernelIns* | *PrivateIns*	(3)
KernelIns	=	`in`(*Ch*) | `out`(*Ch*) | `alt`(*Alts*)	(4)
PrivateIns	=	`ldc`(*Const*) | `stl`(*Addr*) | `jump`(*Label*)	(5)
Alts	=	$(Ch1 \times Label)^+$	(6)
Ch	=	$\mathtt{N}_1$	(7)
*Ch*1, *Const*, *Addr*, *Label*	=	$\mathtt{N}_0$	(8)

We briefly describe a few of the instructions. The `in` instruction is used to input a value to the top position of the stack from a channel. Correspondingly, `out` outputs (and pops) the top value of the stack to a channel. The `alt` (alternation) instruction takes as parameter a list of pairs. The first component of a pair is a channel number (if this number is zero, the guard is considered to be a SKIP guard), and the second component is a label (address) where execution should continue if that alternative is chosen. `ldc` pushes a constant onto the stack, `stl` stores the top value of the stack in workspace, and `jump` changes the flow of control within a process. In the remainder of this paper, α is a given *SAL* program.

3.2 The Global Machine, GM

The private state of each process in GM is given by *Pstate*. It consists of a workspace, a stack, an instruction pointer, and a status.

$$\begin{aligned}
\mathit{Pstate} &= \mathit{Workspace} \times \mathit{Stack} \times \mathit{Ip} \times \mathit{Status} && (9)\\
\mathit{Workspace}, \mathit{Stack} &= \mathtt{N_0}^* && (10)\\
\mathit{Ip} &= \mathtt{N_0} && (11)\\
\mathit{Status} &= \underline{\mathsf{ready}} \mid \underline{\mathsf{error}} && (12)
\end{aligned}$$

A state in the GM LTS is simply a list[3] of private process states. The length of the list is the same as the length of the list of process codes. A process code is then connected with its state via the index into the lists. In an initial state each process has an empty stack and a ready status, and its instruction pointer is zero.

$$\begin{aligned}
\Gamma_{GM} &= \{<s_0,\ldots,s_n> \mid n = \underline{\mathsf{len}}\alpha - 1 \wedge s_i \in \mathit{Pstate}\} && (13)\\
\mathsf{I}_{GM} &= \{<s_0,\ldots,s_n> \in \Gamma_{GM} \mid \underline{\mathsf{s}}\text{-}\mathit{Stack}(s_i) = <> \wedge \underline{\mathsf{s}}\text{-}\mathit{Ip}(s_i) = 0 \wedge \underline{\mathsf{s}}\text{-}\mathit{Status}(s_i) = \underline{\mathsf{ready}}\} && (14)\\
\Lambda_{GM} &= \{\tau(i) \mid 0 \le i < \underline{\mathsf{len}}\alpha\} \cup \{ch : v(i,j) \mid ch \in \mathit{Ch} \wedge v \in \mathtt{N_0} \wedge 0 \le i,j < \underline{\mathsf{len}}\alpha\} && (15)
\end{aligned}$$

The function $\underline{\mathsf{s}}\text{-}\mathit{Stack}$ selects the stack component from a state. Similarly, $\underline{\mathsf{s}}\text{-}\mathit{Ip}$ and $\underline{\mathsf{s}}\text{-}\mathit{Status}$ select the instruction pointer resp. the status field. A label $\tau(i)$ denotes that the ith process is executing a private instruction, whereas $ch : v(i,j)$ denotes that process i sends the value v on channel ch to process j.

We describe some of the transition rules which define the transition relation. Here is the transition rule for the `jump` instruction.

$$\frac{\alpha[i][\underline{\mathsf{s}}\text{-}\mathit{Ip}(s_i)] = \mathtt{jump}(\mathit{lab})}{<s_0,\ldots,s_i,\ldots,s_n> \xrightarrow[GM]{\tau(i)} <s_0,\ldots,s_i',\ldots,s_n>} \quad (16)$$

if $0 \le i \le n \wedge$
  let $(ws_i, st_i, ip_i, stat_i) = s_i$ **in**
    $stat_i = \underline{\mathsf{ready}} \wedge ip_i < \underline{\mathsf{len}}\alpha[i] \wedge$
    $s_i' = (ws_i, st_i, lab, stat_i)$

[3]List notation: A list with elements $a_0,\ldots,a_n$ is written $<a_0,\ldots,a_n>$. $<>$ is the empty list. If l is a list, then $\underline{\mathsf{hd}}l$, $\underline{\mathsf{tl}}l$, and $\underline{\mathsf{len}}l$ denote head, tail, and length of l respectively. $l[i]$ accesses the ith element of l with $l[0]$ being the head element. $l + [i \mapsto v]$ replaces the ith element of l with v. Concatenation of l_1 and l_2 is written $l_1{}^\frown l_2$.

In order for the transition below the line to be possible, the condition above the line and the side condition (after **if**) must both be satisfied.

This rule states that any process whose instruction pointer points to a jump instruction and whose status is ready can make a step in which the instruction pointer is changed to the label in the instruction. Only the state of the chosen process is changed.

The following rule defines synchronous communication between two processes.

$$\frac{\alpha[i][\underline{\text{s-}Ip}(s_i)] = \text{in}(ch) \qquad \alpha[j][\underline{\text{s-}Ip}(s_j)] = \text{out}(ch)}{<s_0,\ldots,s_i,\ldots,s_j,\ldots,s_n> \xrightarrow[GM]{ch:v(j,i)} <s_0,\ldots,s_i',\ldots,s_j',\ldots,s_n>} \tag{17}$$

if $0 \le i \le n \wedge 0 \le j \le n \wedge$
$\quad \underline{\textbf{let}}\ (ws_i, st_i, ip_i, stat_i) = s_i, (ws_j, st_j, ip_j, stat_j) = s_j\ \underline{\textbf{in}}$
$\qquad stat_i = \underline{\text{ready}} \wedge stat_j = \underline{\text{ready}} \wedge$
$\qquad ip_i < \underline{\text{len}}\alpha[i] \wedge ip_j < \underline{\text{len}}\alpha[j] \wedge$
$\qquad \underline{\text{len}}st_j > 0 \wedge v = \underline{\text{hd}}st_j \wedge$
$\qquad s_i' = (ws_i, <v>\hat{}st_i, ip_i + 1, stat_i) \wedge$
$\qquad s_j' = (ws_j, \underline{\text{tl}}st_j, ip_j + 1, stat_j)$

This rule defines transitions from states where two processes are ready to execute an **in** resp. an **out** instruction on the same channel, and the outputting process has a nonempty stack. The resulting states are obtained by incrementing the instruction pointers of the two processes and moving the value from the top of the stack of the outputting process to the top of the stack of the inputting process.

The complete definition of the transition relation at this level consists of nine rules including rules dealing with errors, like a process trying to output with an empty stack.

3.3 The Kernel Level 1, KL_1

At the KL_1 level we introduce explicit process scheduling. To do this we add a *kernel state* consisting of a *current process* identifier and a *global instruction pointer*. The private instruction pointer is then only used to store the global instruction pointer when the process is not current. We also introduce an explicit waiting status denoting that a process is waiting to communicate.

$$Kstate1 = Id \times Ip \tag{18}$$
$$Pstate1 = Workspace \times Stack \times Ip \times Status1 \tag{19}$$
$$Id = \mathbb{N}_0 \tag{20}$$
$$Status1 = Status \mid \underline{\text{waiting}} \tag{21}$$

A state of the KL_1 LTS now includes a kernel state and a list of private process states.

$$\Gamma_{KL1} = \{(ks, psl) \mid ks \in Kstate1 \wedge psl \in Pstate1^+ \wedge \underline{\text{len}}psl = \underline{\text{len}}\alpha\} \tag{22}$$
$$I_{KL1} = \{((id, ip), psl) \in \Gamma_{KL1} \mid id = 0 \wedge ip = 0 \wedge (\forall(ws, st, pip, stat) \in psl)(st = <> \wedge pip = 0 \wedge stat = \underline{\text{ready}})\} \tag{23}$$
$$\Lambda_{KL1} = \{\tau(i), \kappa(i), \kappa \mid 0 \le i < \underline{\text{len}}\alpha\} \cup \tag{24}$$
$$\{ch : v(i,j) \mid ch \in Ch \wedge v \in \mathbb{N}_0 \wedge 0 \le i, j < \underline{\text{len}}\alpha\} \tag{25}$$

The new label $\kappa(i)$ denotes that the kernel is performing a step in the ith process. A κ label denotes a process switch transition (see below).

In the complete definition of the KL_1 transition relation, 14 transition rules are needed. In this section we present only a few. Here is the KL_1 jump transition rule.

$$\frac{\alpha[i][ip] = \texttt{jump}(lab)}{((i,ip),psl) \xrightarrow[KL1]{\tau(i)} ((i,ip'),psl)} \quad (26)$$

if $\underline{\textbf{let}}\ (ws_i, st_i, ip_i, stat_i) = psl[i]\ \underline{\textbf{in}}$
$stat_i = \underline{\text{ready}} \wedge ip < \underline{\textbf{len}}\alpha[i] \wedge$
$ip' = lab$

This rule is similar to rule (16) at the GM level. Note, however, that the jump transition at this level changes the *global* instruction pointer.

At the GM level one transition rule is required to specify the synchronous communication transitions. Here, several transition rules are needed. We only show the rules where the current process wishes to execute an in instruction.

If the current process wishes to execute an in instruction and another process is already waiting to output on the same channel, the communication can be performed. Part of the state change is to give the waiting process a ready status.

$$\frac{\alpha[i][ip] = \texttt{in}(ch)}{((i,ip),psl) \xrightarrow[KL1]{ch:v(j',i)} ((i,ip'),psl')} \quad (27)$$

if $\underline{\textbf{let}}\ (ws_i, st_i, ip_i, stat_i) = psl[i]\ \underline{\textbf{in}}$
$stat_i = \underline{\text{ready}} \wedge ip_i < \underline{\textbf{len}}\alpha[i] \wedge$
$(\exists 0 \leq j < \underline{\textbf{len}}psl)$
$(\underline{\textbf{let}}\ (ws_j, st_j, ip_j, stat_j) = psl[j]\ \underline{\textbf{in}}$
$stat_j = \underline{\text{waiting}} \wedge \alpha[j][ip_j] = \texttt{out}(ch) \wedge$
$v = \underline{\textbf{hd}}st_j \wedge j' = j \wedge$
$ip' = ip + 1 \wedge$
$psl' = psl + [i \mapsto (ws_i, <v>\hat{}st_i, ip_i, stat_i),$
$j \mapsto (ws_j, \underline{\textbf{tl}}st_j, ip_j + 1, \underline{\text{ready}})])$

If, on the other hand, another process is not waiting to output on the same channel, the current process is given a waiting status. Such transitions have no GM counterparts.

$$\frac{\alpha[i][ip] = \texttt{in}(ch)}{((i,ip),psl) \xrightarrow[KL1]{\kappa(i)} ((i,ip),psl')} \quad (28)$$

if $\underline{\textbf{let}}\ (ws_i, st_i, ip_i, stat_i) = psl[i]\ \underline{\textbf{in}}$
$stat_i = \underline{\text{ready}} \wedge ip < \underline{\textbf{len}}\alpha[i] \wedge$
$\neg(\exists 0 \leq j < \underline{\textbf{len}}psl)$
$(\underline{\textbf{let}}\ (ws_j, st_j, ip_j, stat_j) = psl[j]\ \underline{\textbf{in}}$
$stat_j = \underline{\text{waiting}} \wedge$
$\alpha[j][ip_j] = \texttt{out}(ch)) \wedge$
$psl' = psl + [i \mapsto (ws_i, st_i, ip_i, \underline{\text{waiting}})]$

Since we have introduced the notion of current process at the KL_1 level, we need transitions to introduce a new current process. The next transition rule defines such *process switch* transitions. The global instruction pointer is stored in the private state of the old current process and then restored from the private state of the new current process.

$$((i, ip), psl) \xrightarrow[KL1]{\kappa} ((j, ip'), psl') \tag{29}$$

if $0 \leq j < \underline{\text{len}}psl \wedge$
$i \neq j \wedge$
$\underline{\text{s-}}Stat(psl[j]) = \underline{\text{ready}} \wedge$
$ip' = \underline{\text{s-}}Ip(psl[j]) \wedge$
let $(ws_i, st_i, ip_i, stat_i)$ **in**
$psl' = psl + [i \mapsto (ws_i, st_i, ip, stat_i)]$

4 Translation into the Boyer-Moore Logic

In this section we describe the translation into the Boyer-Moore logic of the specification for the GM and KL_1 levels. We follow an approach to modeling finite state machines similar to that described in [2] (even though the specifications described here are non-finite state). A state set is defined by a predicate which recognizes elements of the set. An LTS transition rule is translated into a predicate which determines if a transition defined by the rule is possible in the given state, and a function from the state to a *list* of labeled states, which represents the set of possible resulting states. It is necessary to return a list since a transition rule generally defines several transitions emanating from the same state. In all the examples shown below the lists are, however, of length one.

4.1 The Global Machine, GM

The state of a GM process is defined in the Boyer-Moore logic by two events[4]. The add-shell event `pstate` defines a record structure[5]. It carries the information contained in Equation (9) — that the state of a GM process contains a workspace (`ws`), stack (`st`), instruction pointer (`ip`), and a status field (`stat`). In addition, we have made the process's code (`pr`) a part of its state.

```
(add-shell pstate nil pstate-shell
           ((ws   (none-of) false)
            (st   (none-of) false)
            (ip   (none-of) false)
            (stat (none-of) false)
            (pr   (none-of) false)))
```

The predicate `gm-pstatep` imposes type restrictions on the fields of a `pstate`. A GM process must satisfy the requirements that the workspace and stack are lists of numbers,

[4] We use the term *event* to refer to function definitions, lemmas, etc. which have meaning in the Boyer-Moore logic.

[5] An `add-shell` introduces a new data type. The first argument gives the name of the constructor function for the type. The third argument identifies the recognizer for objects of this type. The fourth argument is a list of the fields. Each field is a triple (*fieldname recognizers defaultvalue*). The `(none-of)` notation used in this example indicates no type restriction.

the instruction pointer is a number, and the status is one of the literals {**ready**, **error**}. The function **gm-statep** captures the requirements contained in Equations (10)–(12) of Section 3.2. In addition, the program must satisfy the predicate **processp**. **processp** expresses the well-formedness of a process's code as described in Equations (2)–(8).

```
(defn gm-pstatep (x)
  (and (pstate-shell x)
       (every-numberp (ws x))
       (every-numberp (st x))
       (numberp (ip x))
       (member (stat x) '(ready error))
       (processp (pr x))))
```

The GM state space is a list of GM processes. This is recognized by the predicate **gm-statep**, which requires its argument to be a non-empty list of **gm-pstateps**. This corresponds to Equation (13).

```
(defn gm-statep (gm)
  (and (every-gm-pstatep gm)
       (listp gm)))
```

In addition to this a predicate **gm-initial-statep** must be defined which captures the meaning of Equation (14).

We define transition rules in the logic with two functions as explained above. One is a predicate which characterizes the enabling condition of a transition rule. The other is a function from a GM state to a list of **cons** pairs. The **car** of each pair is a label and the **cdr** is a GM state.

The translation of rule (17) is described as follows. The predicate **gm-in-out-enabled** recognizes the enabling conditions for a GM in-out transition. The arguments to this predicate are a GM state **gm** and process identifiers **i** and **j**. The predicate requires the **i**th process to be in a ready state with its instruction pointer addressing the **in** instruction, and the **j**th process to be ready to do an output with a non-empty stack. Furthermore, they must be communicating on the same channel.[6]

```
(defn gm-in-out-enabled (gm i j)
  (let ((pi (nth i gm))
        (pj (nth j gm)))
    (and (lessp i (length gm)) (lessp j (length gm))
         (equal (stat pi) 'ready) (equal (stat pj) 'ready)
         (lessp (ip pi) (length (pr pi))) (lessp (ip pj) (length (pr pj)))
         (listp (st pj))
         (let ((instri (gm-fetch pi)) (instrj (gm-fetch pj)))
           (and (equal (opr instri) 'in) (equal (opr instrj) 'out)
                (equal (arg instri) (arg instrj)))))))
```

[6]Here are a few of the primitive functions defined in the Boyer-Moore logic upon which this specification is based. **(length l)** returns the number of (top-level) elements in list **l**. **(nth i l)** fetches the **i**th element (zero based) from **l**. **(put i v l)** replaces the **i**th element of **l** with **v**.

The communication transitions are now defined by gm-in-out-transition. As indicated by rule (17), it returns a list containing only the pair ((tau i) . s'), where s' is the state resulting from the transition. The top value is popped from j's stack and pushed onto i's. The instruction pointer of both processes is advanced by 1. Both processes remain ready.

```
(defn gm-in-out-transition (gm i j)
  (let ((pi (nth i gm)) (pj (nth j gm)))
    (let ((instri (gm-fetch pi)) (instrj (gm-fetch pj)))
      (list (cons (list 'comm j i (arg instrj) (car (st pj)))
                  (put i
                       (pstate (ws pi)
                               (cons (nth 0 (st pj)) (st pi))
                               (add1 (ip pi))
                               (stat pi)
                               (pr pi))
                       (put j
                            (pstate (ws pj)
                                    (nthcdr 1 (st pj))
                                    (add1 (ip pj))
                                    (stat pj)
                                    (pr pj))
                            gm)))))))
```

Compare these definitions with rule (17). The predicate gm-in-out-enabled contains the requirement which occurs above the inference line (that i's current instruction is in, and j's current instruction is out), as well as requirements described in the side condition.

The side condition in (17) is used in part to describe the details of the transitions defined. These aspects of the transition rule occur in the function gm-in-out-transition.

4.2 The Kernel Level 1, KL_1

The kernel level KL_1 is described in the Boyer-Moore logic in the same style as the GM level. The KL_1 state space is defined by a shell kl, and a predicate kl-statep which imposes type restrictions on the KL_1 fields. Recall that KL_1 introduces explicit process scheduling by including a current process id kid, and a global instruction pointer kip for the current process. The psl field is a list of KL_1 private process states. A private state at this level differs from a GM private state only in that a new process status waiting is introduced.

```
(add-shell kl nil kl-shellp
           ((kid (one-of numberp) zero)
            (kip (one-of numberp) zero)
            (psl (none-of) false)))

(defn kl-statep (x)
  (and (kl-shellp x)
       (kl-pstate-listp (psl x))
       (listp (psl x))))
```

Here is the translation of the KL_1 transition rule (27). Such a communication transition is enabled if the current process is ready to receive on a channel, and some other process is waiting to send on the same channel. In place of the existential quantifier, we write a recursive function, here called `some-process-pow-channel`, which recognizes when some process is in the enabling in-out relation with the current process.

```
(defn kl-in-out-enabled (s)
  (let ((p (nth (kid s) (psl s))))
    (and (equal (stat p) 'ready)
         (lessp (kip s) (length (pr p)))
         (let ((instruction (kl-fetch s)))
           (and (equal (opr instruction) 'in)
                (some-process-pow-channel (psl s) (arg instruction)))))))
```

`kl-in-out-transition` describes the transition on the current process and on a process `j` which is sending to the current process.

```
(defn kl-in-out-transition (s j)
  (let ((pi (nth (kid s) (psl s))) (instri (kl-fetch s)) (pj (nth j (psl s))))
    (let ((instrj (nth (ip pj) (pr pj))))
      (list (cons (list 'comm j (kid s) (arg instrj) (car (st pj)))
                  (kl (kid s)
                      (add1 (kip s))
                      (put (kid s)
                           (pstate (ws pi)
                                   (cons (nth 0 (st pj)) (st pi))
                                   (ip pi)
                                   (stat pi)
                                   (pr pi))
                           (put j
                                (pstate (ws pj)
                                        (nthcdr 1 (st pj))
                                        (add1 (ip pj))
                                        'ready
                                        (pr pj))
                                (psl s)))))))))
```

5 The Correctness Theorems

A correctness theorem for each KL_1 transition rule was derived from Definition 1 in Section 2.1. A mapping function map defines the abstraction from a KL_1 state to a GM state. The abstraction changes the status of every waiting process to **ready**. The current KL_1 instruction pointer is installed into the state of the current process. The KL_1 instruction pointer and current process identifier vanish in the mapping.

```
(defn map-pstate (p)
  (pstate (ws p) (st p) (ip p) (if (equal (stat p) 'error) 'error 'ready)
          (pr p)))
```

```
(defn map-states (l)
  (if (listp l)
      (cons (map-pstate (car l)) (map-pstates (cdr l)))
      nil))

(defn map (kl)
  (let ((p (nth (kid kl) (map-pstates (psl kl)))))
    (put (kid kl)
         (pstate (ws p) (st p) (kip kl) (stat p) (pr p))
         (map-pstates (psl kl)))))
```

Using this mapping function, we state a pair of correctness theorems for each transition rule in KL_1. The first theorem requires that a GM transition is enabled if the corresponding KL_1 transition is enabled. Here is an example of this theorem proved for the communication transition rules. This theorem says that for a valid KL_1 state s (as defined by `kl-statep`), a GM communication transition is enabled on the mapping of s if the KL_1 communication transition is enabled on s.

```
(prove-lemma gm-in-out-enabled-map
  (implies (and (kl-statep s)
                (inv-kl s)
                (kl-in-out-enabled s)
                (channel-wo-process (arg (kl-fetch s)) j (psl s)))
           (gm-in-out-enabled (map s) (kid s) j)))
```

A second theorem states the correctness of the transition. Let s be a valid KL_1 state on which a communication transition is enabled, and let `s1` be a possible outcome of such a transition on s. (`s1` is a `(label . state)` pair.) Then the pair `(label . (map state))` is a possible outcome of the GM communication transition performed on `(map s)`.

```
(prove-lemma kl-in-out-correctness
       (implies (and (kl-statep s)
                     (inv-kl s)
                     (kl-in-out-enabled s)
                     (member s1 (kl-in-out-transition s j))
                     (channel-wo-process (arg (kl-fetch s)) j (psl s))
                     (numberp j))
                (member (cons (car s1) (map (cdr s1)))
                        (gm-in-out-transition (map s) (kid s) j))))
```

`inv-kl` is the invariant on reachable KL_1 states which we use throughout the correctness proofs. It contains the fact that any process waiting to output has a non-empty stack, and that the kernel's current process identifier is "valid", i.e. identifies an existing process.

The $KL1_1$ process switch transition (29) has no corresponding transition at the GM level. We prove that this transition is invisible at the GM level. This is contained in the theorem `kl-switch-correctness`.

```
(prove-lemma kl-switch-correctness
  (implies (and (kl-statep s)
                (numberp z)
                (inv-kl s)
                (kl-switch-enabled s z)
                (member s1 (kl-switch-transition s z)))
           (equal (map (cdr s1))
                  (map s))))
```

The correctness theorems above are all derived from (ii) of Definition 1. The theorems are stronger than (ii) since, given a GM transition, we only search for a corresponding KL_1 transition among transitions defined by a certain rule. This is, however, also the way a hand proof would be done.

The correctness theorem for (i) of Definition 1 is simply

```
(prove-lemma initial-correctness
  (implies (kl-initial-statep s) (gm-initial-statep (map s))))
```

For each KL_1 transition which has a corresponding GM transition, we completed a proof of correspondence of the enabling condition and the transition rule. Where a KL_1 transition has no corresponding GM transition we completed a proof that the transition rule is invisible at the GM level. The simple measure of the size of our script, 162 function definitions and 287 proved lemmas, should be taken as an upper bound on the size necessary to complete the project, since we experimented with a macro language for the Boyer-Moore theorem prover.

6 Observations

The purpose of this experiment was to discover if the kernel specifications given in [8] could be translated in a satisfactory way into the Boyer-Moore logic, and to discover how difficult the correctness proofs would be.[7]

We feel that the translation was a success. There was little problem defining functions in the logic which capture the meaning of the specification given in terms of a labeled transition system. Also, the correctness theorems as expressed in the logic were clearly instances of the correctness notion developed for relating LTSs.

The correctness theorems were a good candidate for mechanical checking. Their proofs involve many cases, none of which are very difficult. Hand proofs of these theorems are mistake-prone. In fact, only a few had been attempted by hand because of the tedium involved in writing them down.

A number of similar mapping proofs have been previously checked with the theorem prover [2], all involving much more complicated mappings from concrete to abstract

[7] We hope that the comments about the prover in this section are intelligible. One point worth noting is that one of the central proof strategies used by the theorem prover is term rewriting. The user builds up a database of facts by stating lemmas of the form $H \rightarrow L = R$. An instance of L will be rewritten to the corresponding instance of R if condition H holds. A user can give the theorem prover hints indicating which rewrite rules to apply or which ones to ignore. Alternatively, the user can just let the theorem prover try every applicable rule in the current database.

machines. We expected the KL_1 correctness proofs to be straightforward exercises, particularly considering that one of the authors is an experienced user of the Boyer-Moore prover. This expectation was only partially realized. Some of the proofs, particularly those concerning the I/O transition rules, were far more difficult than we expected. This is disappointing, since the reasoning steps seem elementary when done by hand.

The proofs of the private transition rules were simple and followed a pattern familiar to users of the Boyer-Moore prover. The first proof took some effort, but proofs of subsequent private transition rules were structured in a way similar to the first. These proofs were easily accomplished simply by adjusting the set of supporting rewrite lemmas.

The difficult proofs involved reasoning about specifications where existential quantifiers over process identifiers occur in the LTS version. The existential quantifiers were replaced by recursive functions in the Boyer-Moore translation, thereby introducing an additional level of recursion. We made several attempts before we achieved a formulation that was clear enough to incorporate into the prover's rewrite algorithm. Our solution also involved creation of a more complete set of lemmas for the supporting theories, primarily lists, than we had before.

Is it worthwhile to expend such effort in solving problems of prover control? Our experience is that it is. The base theories which are developed and the insight gained into the problem domain pay off whenever a related problem is addressed.

Acknowledgement: We would like to thank Bill Young of Computational Logic for his contributions in formulating the Boyer-Moore version of the kernel specifications.

References

[1] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.

[2] William R. Bevier, Jr. Warren A. Hunt, J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[4] INMOS Limited. occam2 *Reference Manual.* Series in Computer Science. Prentice Hall, 1988.

[5] INMOS Limited. *Transputer Instruction Set: A compiler writer's guide.* Prentice Hall, 1988.

[6] Robin Milner. *Communication and Concurrency.* Series in Computer Science. Prentice Hall, 1989.

[7] G. D. Plotkin. An operational semantics for CSP. *Formal Description of Programming Concepts - II*, pages 199–225, 1983.

[8] Camilla Østerberg Rump and Jørgen F. Søgaard-Andersen. Specification and verification of kernels. Master's thesis, Department of Computer Science, Technical University of Denmark, August 1990.