A Mechanized Process Algebra for Verification of Device Synchronization Protocols [1]

E. Thomas Schubert [2]
Department of Computer Science
University of California, Davis

Abstract - **We describe the formalization of a process algebra based on CCS, within the HOL theorem-proving system. The representation of four types of device interactions and a correctness proof of the communication between a microprocessor and MMU is presented.**

1 Introduction

This paper describes a methodology to formally verify the correctness of the synchronization and communication within a hardware system. The methodology is based on the formalization of a process algebra within the HOL theorem-proving system. Using this formalization, we show how four types of device interactions can be represented and proven to behave as specified. This paper also includes a correctness proof for the communication between the microprocessor and an memory management unit.

Previous system integration efforts have focused on vertical integration of one layer on top of another [1, 2]. This work examines the "horizontal" integration of communicating devices. Device communication may require only a single message or a series of messages to be passed between two devices. The types of interactions can be loosely described as belonging to one of the following categories: remote procedure call, process creation (fork), message passing, or rendezvous. To demonstrate how system integration can be achieved, we will show how several devices can be composed to form a system that uses these communication protocols.

The remainder of this section discusses background material, Section 2 presents the mechanization of the process algebra within the HOL theorem prover, and Section 3 presents specification and verification examples using the formalized process algebra, including a CPU-MMU composition proof. The final section summarizes this work and describes future extensions.

1.1 Related Work

Hardware verification requires that the design of a system is formally shown to satisfy its specification through a mathematical proof. Using theorem-proving techniques, an expres-

[1]This work was performed under Boeing Contract NAS1-18586, Task Assignment No. 3, with NASA-Langley Research Center.

[2]Author's current address is: Department of Computer Science, Portland State Univeristy

1.1 Related Work

Hardware verification requires that the design of a system is formally shown to satisfy its specification through a mathematical proof. Using theorem-proving techniques, an expression describing the behavior of a device is proven to be equivalent in some sense to an expression describing the implementation structure of the device. These expressions concisely describe the behavior of devices in an unambiguous way. An additional benefit of hardware verification is that the behavioral semantics of the hardware are clearly defined. This provides an accurate basis for building correct software systems [1].

Hardware verification efforts thus far have focused primarily on a microprocessor as the base for computer systems [2, 3, 5]. However, these processors are quite limited. The processors verified have modeled small instruction sets and, generally, have not included modern CPU features such as pipelines, multipled functional units, and hardware interrupt support. Tamarack-3 [2] and AVM-1 [6] do provide sufficient interrupt support to connect with an interrupt controller. However, no system currently verified provides the memory management functions necessary to support a secure operating system.

Process algebras and concurrency theories seek to provide formal models that aid in our understanding of the behavior of such systems. The archetype for the process algebra developed, is the Calculus of Communicating Systems (CCS) [7]. Despite the seemingly simple syntactic definition of CCS, the semantics of its primitive operators are carefully defined and allow complicated communication schemes to be accurately represented.

2 Mechanization of the Process Algebra

This section will describe our work to represent CCS in HOL. This representation has allowed us to easily prove several of the CCS semantic laws. To develop the process algebra in the HOL logic, we take advantage of several type definition mechanisms provided by the logic. Using these facilities we define an initial algebra for *agents*, which are constructed from sequences of *actions*. The purpose of types in higher-order logic is to prevent the inconsistency that higher-order variables can induce. The recursive type definition facility [8] automates the process of defining new data types in terms of already existing types. Both constants of the new type and type operators can be defined. Type operators are constructor functions, used to build compound members of new type from the type constants. The properties of these new types are then derived by formal proof. This guarantees that the new type does not introduce inconsistency into the logic. Additional recursive operators can be defined to operate on the concrete data representation of the type.

2.1 Actions

Using the recursive type definition facility, *actions* are defined to be either internal or external transitions. The internal action represents the τ action of CCS. External transitions require an *label*, which consists of a name (string) and boolean value, that denotes whether the action is a send or receive synchronization operation.

```
new_type_abbrev(`name`, ":string");;
new_type_abbrev(`label`, ":bool#name");;

let ACTION = define_type `action`
   `action = INTERNAL | EXTERNAL label`;;
```

2.2 Agents

Type representations the syntax of a concrete data type and a term algebra is also formed. Good practice suggest that the representation definition of any type, should minimize the number of type operators. This is certainly true if the semantics of some operators can be defined in terms of others. For determining the equivalence of two agents, we would like to define all agents in a normal form with agent terms consisting of only the prefix and summation operators. To describe processes that execute in parallel, we adapt a method described in [9]. This technique allows concurrently executing agent expressions to be represented with only the prefix and summation operators. Three mutually recursive functions replace the composition type constructor:

1. A communication operator, COMM, that declares that two agents will communicate if they have complementary, enabled actions.
2. A left-merge operator, LMERGE, that creates a new agent from two agents, such that the new agent must first behave as though only the left agent (first argument) were present and followed by an agent constructed by the composition of the resulting left agent and original right agent.
3. A compose operator ("merge" in the literature), COMPOSE, that creates an arbitrary interleaving of the two agents (through use of the summation type constructor).

The functions relate through the following laws, where the symbol "||", denotes the COMPOSE operator, the symbol "|" denotes the COMM operator, and "L" denotes the left-merge operation.

1. Given $P = a.P' and Q = b.Q'$

   $P \mid Q = (a = \ COMPLEMENT\ b) \ \rightarrow \ \tau.(P' \mid\mid Q')\ else\ \mathbf{0}$
2. $x \mid\mid y = (xLy) + (yLx) + (x \mid y)$

3. $aLx = a.x$
4. $a.xLy = a.(x \mid\mid y)$
5. $(x + y)Lz = (xLz) + (yLz)$
6. $ax \mid bx = (a \mid b).(x \mid\mid y)$
7. $(x + y) \mid z = (x \mid z) + (y \mid z)$
8. $x \mid (y + z) = (x \mid y) + (x \mid z)$

HOL does not provide a mechanism for defining mutually recursive functions, such as COMPOSE and LMERGE. Instead, two auxiliary compound type operators are present in the type definition with axiomd defining their semantic meaning.

```
let AGENT = define_type 'agent'
   'agent = INACTIVE
           | PREFIX action agent
           | SUMM agent agent
           | COMPOSE agent agent
           | LMERGE agent agent
           | RESTRICT label agent';;

COMM_DEF_AX =  ⊢ ∀ A B a b.  COMM (PREFIX a A) (PREFIX b B) =
         IS_COMPL_ACT a b → PREFIX INTERNAL (COMPOSE A B) |  INACTIVE

COMM_DIST_AX = ⊢ ∀ A B C. (COMM (SUMM A B) C = SUMM (COMM A C) (COMM B C))  ∧
                (COMM A (SUMM B C) = SUMM (COMM A B) (COMM A C))
COMPOSE_AX = ⊢ ∀ A B. COMPOSE A B = SUMM(SUMM(LMERGE A B) (LMERGE B A)) (COMM A B)

LMERGE_AX = ⊢ ∀ A B a. LMERGE (PREFIX a A) B = PREFIX a (COMPOSE A B)

LMERGE_DIST_AX = ⊢ ∀ A B C. LMERGE (SUMM A B) C = SUMM(LMERGE A C)(LMERGE B C)
```

2.3 Transition Semantics

Having defined the form of agents in the previous section, the transition semantics of an evolving agent (e.g., $A \xrightarrow{\alpha} B$) can be captured using the inductive relation definition package. The inductive relation definition package provides a set of theorem-proving tools based on a newly derived principle of definition in HOL for defining relations inductively by a set of rules [10]. Rules consist of a list of premisses and side conditions and a conclusion. Each premiss must make a positive assertion about membership in the relation. Side conditions may be arbitrary propositions not involving the relation being defined (as an example, see the TRANS_RES law definition below). The rules are essentially implications: if the premisses and side conditions hold, then the conclusions hold. The relation is inductively defined by a collection of such rules to be the least relation closed under all the rules.

The TRANS function defined below states that for an agent to evolve to another agent, there must be an immediate transition by a prefixed action or a set of premisses must be satisfied. To embody the CCS laws, symmetric forms are needed for summation laws and the composition laws.

```
let (TRANS_rules, TRANS_ind) =
    let  TRANS = "TRANS:agent->action->agent->bool" in
    new_inductive_definition false `TRANS`  ("^TRANS x a y",[])
    [
 % ACT LAW %
[ ],
 %---------------------------------------%
        "^TRANS (PREFIX a y) a y"
;% SUM1 LAW %
[              "^TRANS x a y"    ],
 %---------------------------------------%
               "^TRANS (SUMM x z) a y"
;% SUM2 LAW %
[              "^TRANS x a y" ],
 %---------------------------------------%
               "^TRANS (SUMM z x) a y"
;% COM1 LAW %
[              "^TRANS x a y" ],
 %---------------------------------------%
  "^TRANS (COMPOSE x z) a (COMPOSE y z)"
;% COM2 LAW %
[              "^TRANS x a y" ],
 %---------------------------------------%
          "^TRANS (COMPOSE z x) a (COMPOSE z y)"
;% COM3 LAW %
[    "^TRANS x (EXTERNAL (T,s)) x'";
     "^TRANS y (EXTERNAL (F,s)) y'" ],
 %---------------------------------------%
 "^TRANS (COMPOSE x y) INTERNAL (COMPOSE x' y')"
;% RES LAW %
[    "^TRANS x a y";
     "~( (LBL a) = (SND (r:label)))" ],
 %---------------------------------------%
          "^TRANS (RESTRICT r x) a (RESTRICT r y) "    ];;
```

2.4 Agent Equivalence

Several notions of equivalence between agents can be defined. The kinds of models that the initial algebra satisfies will be determined by which notion of equivalence is used. Trace semantics can be shown by defining the meaning of an agent as a set of traces where a trace is a list of actions. Trace semantics permit fairly broad equivalence classes to be constructed. We are frequently interested in a narrower definition of equivalence where two agents are equivalent if an external agent cannot distinguish between the visible behavior (traces) of the two agents. When using the notion of strong equivalence, traces consist of both external and internal actions. For our application, a weaker notion of observation equivalence where internal actions cannot be detected by the external agent, is sufficient.

We are actually interested in a weaker form: one-way observation equivalence. Under this definition, P implements Q's behavior if for every action α of Q, every α-derivative of Q is one-way observation equivalent to some α-descendant of P. For the remainder of the paper we will use the term observation equivalence to mean one-way observation equivalence.

Observation equivalence can be defined in terms of an inductive relation definition. Observation equivalence laws are defined for compound terms, which are constructed using only the PREFIX or SUMM operators.

```
let (OE_rules, OE_ind) =
    let OE = "OE:agent->agent->bool" in
    new_inductive_definition false `OE`
    ("^OE A B",[])
    [[
      %1---------------------------------------% ],
                  "^OE A A"
 ;
     [            "^OE A B"
      %2---------------------------------------% ],
               "^OE (PREFIX a A) (PREFIX a B)"
 ;
     [            "~(TRANS_ACT_EXISTS B C) /\
                  ^OE A C"
      %3---------------------------------------% ],
               "^OE (SUMM A B) C"
 ;
     [            "~(TRANS_ACT_EXISTS A C) /\
                  ^OE B C"
      %4---------------------------------------% ],
               "^OE (SUMM A B) C"
 ;
     [
                  "^OE A C";
                  "^OE B C"
      %5---------------------------------------% ],
               "^OE (SUMM A B) C"
 ;
     [
                  "^OE A B";
                  "^OE A C"
      %6---------------------------------------% ],
               "^OE A (SUMM B C)"
    ];;
```

Rules 1 and 2 are straightforward; observation equivalence is reflexive, and two agents prefixed with the same action are observation equivalent if the agents without the prefixed action are observation equivalent.

If the left-hand-side agent is a summation of two agents (i.e. OE (SUMM A B) C) rules 3–5 may apply, and one-way observation equivalence is achieved if either:

1. Both of the summation agents (A and B) satisfy observation equivalence with the right-hand-side agent (C). This is rule number 5, OE_LSUM.
2. The left summation agent (A) satisfies observation equivalence with the right-hand-side agent (C) and there is no action for which a transition for both the right summation agent (B) and the right-hand-side agent (C) exists. This is rule number 3, OE_LSUML.

3. The right summation agent (B) satisfies observation equivalence with the right-hand-side agent (C) **and** there is no action for which a transition for both the left summation agent (A) and the right-hand-side agent (C) exists. This is rule number 4, OE_LSUMR.

The last rule states that if the right-hand-side agent is a summation agent, then the left-hand-side agent must satisfy observation equivalence for both of the right-hand-side summation agents. This is the symmetric case of rule number 5 for a left-hand summation.

Note that symmetric rules for rules 3 and 4 do not exist. If they were present, the OE relation would specify trace semantics. For example,

$$OE\ (a.0 + b.0)\,(b.0 + a.0)$$

requires both:

$$OE\ (a.0 + b.0)\,(a.0) \text{ and } OE\ (a.0 + b.0)\,(b.0)$$

Adding the symmetric rules would allow relations such as $OE\ (b.0)\,(a.0 + b.0)$ to be true. If the semantic meaning of the OE relation is read as "implements", then $(a.0+b.0)$ implements $(a.0)$, but,$(a.0)$ does not satisfy/implement $(a.0 + b.0)$.

3 Mechanization of Device Interactions

3.1 Generic Interactions

Device communication may require only a single message or a series of messages to be passed between two devices. At a lower level, the information is passed over a bus using a hardware protocol (e.g., 4-phase handshaking). The types of interactions can be loosely described as belonging to one of the following categories:

1. Remote Procedure Call,
2. Message Passing,
3. Process Creation (fork), and
4. Rendezvous.

These forms of interaction are described in concurrent programming literature. While the Ada programming language provides primitive support for only remote-procedure-call and rendezvous features, the SR programming language provides primitive statements for each of the above interaction forms.

3.1.1 Remote Procedure Call

A CPU and a memory subsystem interact in a remote-procedure-call manner. The CPU sends a memory request to the subsystem and waits for a response. If the memory subsystem includes a memory management unit (MMU), the MMU and the actual memory may also interact in a remote procedure form. In the example below, we show a proof of a 4-phase handshaking protocol between a CPU and a bus.

```
let cpuRaiseRead = `    -'cpu_addr'.-'cpu_rReq'  `;;
let cpuReadMem   = `    . 'dataAvail'.'data'       `;;
let cpuDropRead = `     . -'cpu_addr'.-'cpu_rReq'  `;;
let cpuReadComplete = `  . SUCCESS `;;

let busGetRead     = `    'cpu_addr'.'cpu_rReq'  `;;
let busReturnMem   = `    . -'dataAvail'.-'data'       `;;
let busDropRead = `     . 'cpu_addr'.'cpu_rReq'  `;;
let busReadComplete = `  . SUCCESS `;;

let cpuRead = cpuRaiseRead^cpuReadMem^cpuDropRead^cpuReadComplete;;
let busRead = busGetRead^busReturnMem^busDropRead^busReadComplete;;

let system_def = `(`^cpuRead^`)|(`^busRead^`)`;;
let system = Agent system_def;;

let system_done = ` SUCCESS | SUCCESS `;;
let success = Agent system_done;;
```

3.1.2 Message Passing

Devices also interact through a message-passing format. For example, consider the interaction between a CPU and an interrupt controller. These devices operate simultaneously, performing independent tasks, and may never interact (this is, of course, unlikely). If an enabled I/O device generates an interrupt, the system's interrupt controller will send a message to the CPU indicating that some device requires attention. At this point the CPU may or may not respond. Depending on the signaling discipline (e.g., *signal-and-continue* or *signal-and-wait*), the interrupt controller may also continue to operate. The example below shows how the interaction between a CPU and a PIC can be modeled.

```
let cpuInt           = ` 'intPending'.CPU_PROCESS_INT`;;
let cpuIntEnabled  =  cpuInt^ `+ tau.FETCH_INSTRUCTION`;;
let cpuIntDisabled = ` FETCH_INSTRUCTION `;;

let picInt           = `  -'intPending'.PIC_PROCESS_INT `;;
let picReq           =  picInt^ ` + tau.PIC_CHECK  `;;
let picNoReq         =  ` PIC_CHECK `;;

let system_def = `(`^pic^`)|(`^cpu^`)`;;
```

3.1.3 Process Creation and Rendezvous

The interaction between a CPU and a Direct Memory Access device (DMA), can be characterized as process creation followed by a rendezvous. The process of initializing the DMA to supervise the transfer of data from an I/O device to memory, is a form of process creation. When the DMA has completed its task, the CPU (through an interrupt controller) will be signaled. In many circumstances, the CPU activity must rendezvous with the completion of the DMA activity. For example, when a new code page must be brought into memory for an executing program to continue, the CPU may switch to other executing programs, but once all other programs have completed, the CPU (really the OS kernel) must wait for the DMA activity to complete.

```
let dmaProcCreate    =   `  cpuWriteReg.cpuInitExec.DMA_EXEC`;;
let dmaSleep         =   dmaProcCreate;;
let dmaDone          =   ` -'dmaInt'.dmaSleep `;;
let dma              =    dmaSleep;;

let cpuDMACreate     = ` -'cpuWriteReg'.-'cpuInitExec`;;
let cpuDMAWait       = ` 'dmaInt'.cpuContinue `;;
```

3.2 CPU/MMU Connection Specification

To determine how a CPU and MMU interact, we examine the external interface of AVM-1 with memory and the interface that the MMU provides to the CPU. Below we show the AVM-1 memory interface specification. If a write request is made at time t the memory will reflect this request at time $t + 1$, otherwise, the memory will remain unmodified. If a read request is made at time t then the data value returned is a function of the memory contents at time t.

```
let MEM = new_definition
   (`MEM`,
    "! (rep:^rep_ty) wr_s rd_s addr data mem.
     MEM rep wr_s rd_s addr data mem =
     !t:time .
        (mem (t+1) =
           (wr_s t => store rep (mem t, address rep (addr t), (data t))
                    | mem t)) /\
        (rd_s t ==> (data t = (fetch rep (mem t, address rep (addr t)))))"
   );;
```

This specification is incomplete for the purposes of connecting the CPU with the MMU verified in [11]. There is no external line to indicate the security status of an executing process (e.g., the supervisor line); nor is there a return code indicating whether the memory request was validated. Additionally, the specification assumes that memory operations occur in a single cycle. This last consideration could be dealt with by an appropriate temporal abstraction.

Below is a modified specification that includes these features. A security line is added to the specification and an acknowledgment variable (`ack`) is added to inform the CPU of invalid memory requests. This variable is set based on a new abstract function `memMgt`. The abstract CPU functions, `store` and `fetch`, now expect the supervisor state as an additional argument (`superV`). These functions should not perform as requested when security/safety requirements are not satisfied.

```
let MEM = new_definition
('MEM',
 "! (rep:^rep_ty) wr_s rd_s addr data mem superV ack.
  MEM rep wr_s rd_s addr data mem superV ack =
  !t:time . (mem (t+1) =
      (wr_s t => store rep (mem t, address rep (addr t), (data t), superV t)
              | mem t)) /\
  (rd_s t ==> (data t=(fetch rep (mem t, address rep (addr t),superV t)))) /\
  (ack t = memMgt rep (mem t, address rep (addr t),data t,superV t,wr_s t))";;
```

With these additions, the process algebra term for this interface can be defined as below.

$$\mathsf{cpu\_write\_request} = (\overline{userMode} + \tau.\overline{superMode}).\overline{write}.\overline{address}.\overline{data}.(ack + nack)$$

$$\mathsf{cpu\_read\_request} = (\overline{userMode} + \tau.\overline{superMode}).\overline{read}.\overline{address}.(ack + nack).data)$$

$$\mathsf{CPUtoMEM} = \mathsf{cpu\_write\_request} + \tau.\mathsf{cpu\_read\_request}$$

$$\mathsf{CPU} = \mathsf{CPUtoMEM.CPU}$$

These terms are abbreviations for their actual representations in HOL. The PREFIX operator is defined to construct an agent from an action and an agent rather than from two agents as presented here.

Note the use of the internal operator to indicate that at given points, the CPU will behave in one of two ways: communicate through the $\overline{userMode}$ action or through the $\overline{superMode}$ action. This choice is made internally by the CPU. Without this use of τ, the terms would mean that communication could occur by either action, depending on what an external agent might choose. This use of the τ action can also be seen in the process algebra terms for the MMU below. To express the notion that the MMU performs some work before responding, the τ action is inserted before the response is returned. Part of this action may be to update the segment-table pointer value. While the system is able to accept either a *userMode* or a *superMode* communication, the MMU chooses to respond with only $\overline{ack}$ or $\overline{nack}$ communication. The MMU specification also states when the MMU segment-table pointer is updated. This action is not relevant to the CPU -MMU interface, so it is expressed as an internal (τ) action. This specification yields the process algebra terms:

$$\mathsf{mmu\_process\_write} = (userMode + superMode).write.address.data.\tau.(\overline{ack} + \tau.\overline{nack})$$

$$\mathsf{mmu\_process\_read} = (userMode + superMode).read.address.\tau.(\overline{ack} + \tau.\overline{nack}).\overline{data}$$

$$\mathsf{MMUtoCPU} = \mathsf{mmu\_process\_write} + \mathsf{mmu\_process\_read}$$

MMU = MMUtoCPU.MMU

System = CPU | MMU

3.3 Proof of Correct Composition

The agents CPU and MMU are recursively defined and exhibit an infinite behavior. To show that the composed CPU and MMU communicate correctly, we must show that the system is always able to return to its initial state. It is also necessary to show that progress is made when the CPU initiates a dialogue with the MMU. The proof goal then can be stated as:

1. If either of the CPU actions, $\overline{userMode}$ or $\overline{superMode}$, are enabled, the memory subsystem will engage in communication.
2. The communication protocol will complete and the system returns to its initial state.

To formalize and prove this goal in HOL, agent terms are defined based on process algebra terms. To reason about the finite protocol communication sequences, the recursive behavior of the agents is removed. The recursive agent reference is replaced with an (undefined) agent constant SUCCESS.

In the box below, we show the construction of the MMU process algebra term. The CPU term is defined in a similar manner. The term is constructed in parts by building up an ML string. The ML function Agent parses the ML string and returns a HOL agent term. ML strings are delimited by backquotes (‘) and the string concatenation operator is the caret symbol (^).

```
let muw = ‘('write'.'addr'.'data'.((-'ack'.SUCCESS)+(T.-'nack'.SUCCESS)))‘;;
let mur = ‘('read'.'addr'.((-'ack'.'data'.SUCCESS)+(T.-'nack'.'data'.SUCCESS)))‘;;

let msw =‘('write'.'addr'.'data'.((-'ack'.SUCCESS)+(T.-'nack'.SUCCESS)))‘;;
let msr =‘('read'.'addr'.((-'ack'.'data'.SUCCESS)+(T.-'nack'.'data'.SUCCESS)))‘;;

let user_mmu  = ‘( 'user'  .(‘^ muw ^‘+‘^ mur ^‘))‘;;
let super_mmu = ‘( 'super' .(‘^ msw ^‘+‘^ msr ^‘))‘;;
let mmu = ‘(‘^ user_mmu ^‘+‘^ super_mmu ^‘)‘;;
let MMU = Agent mmu;;
```

To express the goal in a general form, several auxiliary definitions are defined. A recursive definition (ENABLED) is defined to construct a list of all output actions that an agent can perform. A goal predicate definition BECOMES is defined. The predicate states that for all possible enabled output actions, a complement action exists such that a success agent is reached immediately or reached in a descendent.

```
let ENABLED = new_recursive_definition false AGENT `ENABLED`
    "(ENABLED(INACTIVE)        = []) /\
     (ENABLED(PREFIX a A)      = ( (INTERNAL=a) => ENABLED A |
                                 (( (TYP a) = F) => [a] | [] ))) /\
     (ENABLED(SUMM A B)        = APPEND (ENABLED A) (ENABLED B)) /\
     (ENABLED(COMPOSE A B)     = APPEND (ENABLED A) (ENABLED B) )";;

let BECOMES = new_definition(`BECOMES`,
  "!(system success :agent).  BECOMES system success =
      (EVERY (\x. (TRANS system x success)) (ENABLED sys) )");;
```

The success agent for the composed MMU-CPU is *SUCCESS* | *SUCCESS*. By unwinding the agent definitions and using the TRANS laws, all possible paths are found to reach the success agent through internal transitions. The MMU-CPU composed communication proof shows that:

```
⊢ BECOMES  (CPU | MMU)  (SUCCESS | SUCCESS)
```

4 Conclusions

We have presented a framework to formally verify the correctness of communication between composed devices. Previous system verification research has developed *vertically verified systems.* However, the hardware bases for these systems have been simplistic. Our research is developing a framework to verify a more realistic *horizontally verified system.* This work demonstrates that CCS is a good choice for describing interdevice implementation-level connections within a computer system. Additional research will expand the calculus and address automating the derivation of process algebra expressions from interpreter specifications. Several improvements are being investigated, including an additional type constructor for recursive agents, greater proof support, and automation of the tedious aspects of the proofs.

References

[1] W. R. Bevier, W. A. Hunt, and W. D. Young, "Toward verified execution environments," *IEEE Symposium on Security and Privacy*, 1987.

[2] J. J. Joyce, *Multi-Level Verification of Microprocessor-Based Systems.* PhD thesis, Cambridge University, December 1989.

[3] A. Cohn, "A proof of correctness of the VIPER microprocessor: the first level," in *VLSI Specification, Verification, and Synthesis* (G. Birtwhistle and P. Subrahmanyam, eds.), pp. 27–71, Kluwer Academic Press, 1988.

[4] W. A. Hunt, "A verified microprocessor," Tech. Rep. 47, The University of Texas at Austin, Dec. 1985.

[5] W. A. Hunt, "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, pp. 429–460, 1989.

[6] P. J. Windley, *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.

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

[8] T. Melham, "Automating recursive type definitions in higher order logic," in *Current Trends in Hardware Verification and Automated Theorem Proving* (G. Birtwhistle and P. Subrahmanyam, eds.), pp. 341–386, Springer-Verlag, 1989.

[9] J. C. M. Baeten and W. P. Weijland, *Process Algebra.* Cambridge University Press, 1990.

[10] T. Melham, "A package for inductive relation definitions in HOL," 1991.

[11] E. T. Schubert, "Verification of memory management units using HOL," Tech. Rep. CSE-90-27, University of California, Davis, August 1990.