NASA Contractor Report 189644

Formal Mechanization of Device Interactions with a Process Algebra

E. Thomas Schubert
Karl Levitt
University of California
Davis, California

G. C. Cohen
Boeing Defense & Space Group
Seattle, Washington

NASA Contract NAS1-18586
November 1992

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5525

Preface

This document was generated in support of NASA contract NAS1-18586, Design and Validation of Digital Flight Control Systems Suitable for Fly-By-Wire Applications, Task Assignments 8. Task 8 is associated with formal verification of embedded systems.

The principle emphasis of the work described in this report is to develop a methodology to formally verify correct synchronization communication of devices in a composed hardware system. Previous system integration efforts have focused on vertical integration of one layer on top of another. This task examines "horizontal" integration of peer devices. To formally reason about communication, we mechanize a process algebra in the HOL theorem-proving system. Using this formalization we show how four types of device interactions can be represented and verified to behave as specified. The report also describes the specification of a system consisting of an AVM-1 microprocessor and a memory management unit, which have been verified in previous work. A proof of correct communication is presented and the extensions to the system specification to add a direct memory device are discussed.

The NASA technical monitor for this work is Sally Johnson of the NASA Langley Research Center, Hampton, Virginia.

The work was done at Boeing Military Airplanes, Seattle, Washington, and at the University of California , Davis, California. Personnel responsible for the work include:

Boeing Military Airplanes:
D. Gangsaas, Responsible Manager
T. M. Richardson, Program Manager
G. C. Cohen, principal investigator

University of California:
Dr. K. N. Levitt, chief researcher
E. Thomas Schubert, PhD candidate

TABLE OF CONTENTS

Section			Page
1.0	INTRODUCTION		1
2.0	BACKGROUND		3
	2.1	Hardware Verification	3
	2.2	Related Work	3
	2.3	The HOL Theorem Prover	4
		2.3.1 The Language.	5
		2.3.2 The Proof System.	8
	2.4	Interpreters	9
		2.4.1 Combining Interpreters	10
3.0	PROCESS ALGEBRAS		13
	3.1	Calculus of Communicating Systems	14
		3.1.1 Transitional Semantics	16
	3.2	Semantics	17
		3.2.1 Internal Actions	19
4.0	MECHANIZATION OF THE PROCESS ALGEBRA		23
	4.1	Actions	24
	4.2	Agents	24
	4.3	Transition Semantics	27
	4.4	Pretty Printing Support	29
	4.5	Pretty Parsing Support	30
	4.6	Agent Equivalence	33
		4.6.1 Equivalence Proofs	35
	4.7	Extensions	36
5.0	MECHANIZATION OF DEVICE INTERACTIONS		37
	5.1	Generic Interactions	37
		5.1.1 Remote Procedure Call	37
		5.1.2 Message Passing	38
		5.1.3 Process Creation and Rendezvous	39
	5.2	AVM-1 /MMU Connection Specification	39
		5.2.1 AVM-1	40
		5.2.2 Abstract MMU	40

5.2.3 CPU MMU Interaction 42
5.3 Proof of Correct Composition 45
5.3.1 System Specification with a DMA Device 47

6.0 CONCLUSIONS 49

REFERENCES 50

LIST OF FIGURES

Figure **Page**

3.2-1 Example of Process Algebra Semantic Interpretations 18
3.2-2 Basic Axioms of CCS 20
3.2-3 Internal Action Decision Point 21
3.2-4 Process Algebra Equivalence Relations 22
4.0-5 New Type Isomorphism 23
4.4-1 Pretty Printer Rule Specification 30
4.5-1 Pretty Parser Grammer Specification 31
5.2-1 Abstract MMU Environment/State 41
5.3-1 CPU/MMU/DMA System 48

LIST OF TABLES

Table **Page**

2.3-1 HOL Infix Operators 6
2.3-2 HOL Binders 7
2.3-3 HOL Type Operators 8

1.0 INTRODUCTION

Hardware systems consist of many composed, hardware devices that communicate with one another. This report describes a methodology to formally verify the correctness of the synchronization 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.

The report also includes a specification of a system consisting of an AVM-1 microprocessor and a memory management unit (MMU). These devices have been verified in previous work (refs. 1 and 2). A correctness proof for the communication between the microprocessor and MMU is presented and an extended system specification, which includes a direct memory access device (DMA), is discussed.

Previous system integration efforts have focused on vertical integration of one layer on top of another. 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. At a lower level, the information is passed over a bus using a hardware protocol (e.g., 4-phase handshaking).

To demonstrate how system integration can be achieved, we will show how several devices can be composed to form a system that uses the communication protocols described above. For example, the CPU and MMU interact in a remote procedure call manner. The CPU sends a memory request to the MMU and waits for the response. The CPU and DMA interact in a manner similar to process creation followed by a rendezvous.

The descriptions of devices are frequently large and the proofs of these devices can require significant amounts of CPU time. For example, the proof of a single microprocessor microinstruction can require 24 hours of CPU time. To reason about composed devices, many details must be abstracted away. Further, the modeling and verification of concurrently executing and communicating devices is quite difficult.

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 in this report, is the Calculus of Communicating Systems (CCS) (ref. 3). 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. CCS provides only a single mechanism for processes to synchronize, but Milner asserts that shared memory, pipes, channels, and procedural rendezvous can all be expressed in terms of this simplest form of communication (ref. 4).

This report describes our approach to verifying composed hardware systems and presents a mechanized process algebra for formally reasoning about device interactions. Section 2 discusses formal verification, the HOL theorem proving system, and related work. Section 3 describes process algebras and presents a description of the operational and transitional semantics for CCS. The mechanization of the process algebra within the HOL theorem proving system is presented in Section 4. Section 5 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.

2.0 BACKGROUND

2.1 HARDWARE VERIFICATION

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 (ref. 5).

Verification is expensive and requires a substantial amount of time. While all development efforts would benefit from the use of formal methods, presently only the verification of life-critical and security properties merits the expense. Tools such as HOL are still under development. The theory libraries are not yet sufficient for general use. Further, techniques and methodologies to verify large systems are not available.

Circuits and devices are described in HOL using a mixture of functions and predicates. Universally quantified variables are used to specify input and output device lines while internal device lines are existentially quantified. The specifications are generally defined to model a state transition system. A specification defines the state and environment at time $t+1$, as a function of the state and environment at time t.

2.2 RELATED WORK

Newman proposes a unified hierarchy that accommodates all critical requirements (ref. 6). Responsibility to satisfy each requirement can then be delegated to an appropriate layer of the design. The layers remain interdependent; the more abstract layers relying on the correctness of the lower levels. Formal proofs about the hardware-level discharge some of the assumptions made by higher, software levels. Similarly, hardware-level proofs often make assumptions about the behavior of the software which are discharged when the level is composed (ref. 7).

Hardware verification efforts thus far have focused primarily on a microprocessor as the base for computer systems (refs. 8, 9, 10 and 11). Perhaps the best known verification effort is that of the VIPER microprocessor (refs. 9, 12 and 13). VIPER is the first microprocessor intended for commercial distribution where a formal verification has been attempted. 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 (ref. 8) and AVM-1 (ref. 14) 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.

Previous efforts to verify systems have attempted to construct vertically verified systems with a microprocessor/memory as the system base. Joyce has verified a compiler level whose target machine description is the specification of the verified Tamarack-3 microprocessor (ref. 15). Computational Logic Inc. has verified a "stack" of interpreters where the implementation of a level is the specification of the next lower level (ref. 5). The "stack" consists of a compiler (Micro-Gypsy) an assembler and linking loader, and a microprocessor. The operating system, KIT, is not part of the verified stack. The KIT project designed and verified an operating system that supports multiple processes and asynchronous I/O. User processes are able to communicate only through message passing, which is implemented by the kernel. This ensures that tasks are isolated from one another. However, the hardware base has not been designed nor verified. Bevier assumes extensions to the FM8502 microprocessor (ref. 11) to provide interrupts, asynchronous I/O, memory management and supervisor-mode instructions.

2.3 THE HOL THEOREM PROVER

HOL is a general theorem-proving system developed at the University of Cambridge (refs. 16 and 17) that is based on Church's theory of simple types, or higher-order logic (ref. 18). Church developed higher-order logic as a foundation for mathematics, but it can be used for describing and reasoning about computational systems of all kinds. Higher-order logic is similar to the more familiar predicate logic, but allows quantification over predicates and functions, not just variables, allowing more general systems to be described.

HOL grew out of Robin Milner's LCF theorem prover (ref. 19) and is similar to other LCF progeny such as NUPRL (ref. 20). Because HOL is the theorem-proving environment used in the body of this work, we will describe it in more detail.

HOL's proof style can be tailored to the individual user, but most users find it convenient to work in a goal-directed fashion. HOL is a tactic-based theorem prover. A tactic breaks a goal into one or more subgoals and provides a justification for the goal reduction in the form of an inference rule. Tactics perform tasks such as induction, rewriting, and case analysis. At the same time, HOL allows forward inference and many proofs are a combination of both forward and backward proof styles. Any theorem-proving strategy a user employs in connection with HOL is checked for soundness, eliminating the possibility of incorrect proofs.

HOL provides a metalanguage, ML, for programming and extending the theorem prover. Using ML, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be combined into new theories for later use. The metalanguage makes the HOL verification system extremely flexible.

In HOL all proofs, even tactic-based proofs, are eventually reduced to the application of inference rules. Most nontrivial proofs require large numbers of inferences. Proofs of large devices such as microprocessors can take many millions of inference steps. In a proof containing millions of steps, what kind of confidence do we have that the proof is correct? One of the most important features of HOL is that it is *secure*, meaning that new theorems can only be created in a controlled manner. HOL is based on five primitive axioms and eight primitive inference rules. All high-level inference rules and tactics do their work through some combination of the primitive inference rules. Because the entire proof can be reduced to one using only eight primitive inference rules and five primitive axioms, an independent proof-checking program could check the proof syntactically.

2.3.1 THE LANGUAGE.

The object language of HOL is described in this section. We will discuss HOL's terms and types.

Terms. All HOL expressions are made up of terms. There are four kinds of terms in HOL: variables, constants, function applications, and abstractions (lambda expressions). Variables and constants are denoted by any sequence of letters, digits, underlines, and primes, starting with a letter. Constants are distinguished in the logic; any identifier that is not a distinguished constant is taken to be a variable. Constants and variables can have any finite arity, not just 0, and, thus, can represent functions as well.

Function application is denoted by juxtaposition, resulting in a prefix syntax. Thus, a term of the form "`t1 t2`" is an application of the operator `t1` to the operand `t2`. The term's value is the result of applying `t1` to `t2`.

An abstraction denotes a function and has the form "λ `x. t`". An abstraction "λ `x. t`" has two parts: the bound variable `x` and the body of the abstraction `t`. It represents a function, `f`, such that "`f(x) = t`". For example, "λ `y. 2*y`" denotes a function on numbers that doubles its argument.

Constants can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written "`rand1 op rand2`" instead of in the usual prefix form: "`op rand1 rand2`". Table 2.3-1 shows several of HOL's built-in infix operators.

Table 2.3-1: HOL Infix Operators

Operator	*Application*	*Meaning*
=	`t1 = t2`	`t1` equals `t2`
,	`t1,t2`	the pair `t1` and `t2`
∧	`t1 ∧ t2`	`t1` and `t2`
∨	`t1 ∨ t2`	`t1` or `t2`
⟹	`t1 ⟹ t2`	`t1` implies `t2`

Constants can also belong to another special class called binders. A familiar example of a binder is ∀. If `c` is a binder, then the term "`c x.t`" (where `x` is a variable) is written as shorthand for the term "`c(λ x. t)`". Table 2.3-2 shows several of HOL's built-in binders.

In addition to the infix constants and binders, HOL has a conditional statement that is written `a → b | c`, meaning "if `a`, then `b`, else `c`."

Table 2.3-2: HOL Binders

Binder	*Application*	*Meaning*
$\forall$	$\forall$ x. t	for all x, t
$\exists$	$\exists$ x. t	there exists an x such that t
ε	ε x. t	choose an x such that t is true

Types. HOL is strongly typed to avoid Russell's paradox and others like it. Russell's paradox occurs in a higher-order logic when one can define a predicate that leads to a contradiction. Specifically, suppose that we define P as P(x) = ¬x(x) where ¬ denotes negation. P is true when its argument applied to itself is false. Applying P to itself leads to a contradiction since P(P) = ¬P(P) (i.e. , $true = false$). This kind of paradox can be prevented by typing since, in a typed system, the type of P would never allow it to be applied to itself.

Every term in HOL is typed according to the following recursive rules:

a. Each constant or variable has a fixed type.

b. If x has type α and t has type β, the abstraction λ x. t has the type $(\alpha \rightarrow \beta)$.

c. If t has the type $(\alpha \rightarrow \beta)$ and u has the type α, the application t u has the type β.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (*) followed by a (possibly empty) sequence of letters and digits. Thus, *, ***, and *ab2 are all valid type variables. All type variables are universally quantified implicitly, yielding type-polymorphic expressions.

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\sigma_1, \ldots, \sigma_n$ are types and op is a type operator of arity n, the $(\sigma_1, \ldots, \sigma_n)$op is a type. Note that type operators are postfix while normal function application is prefix or infix. A type operator of arity 0 is a type constant.

HOL has several built-in types, which are listed in Table 2.3-3. The type operators bool, ind, and fun are primitive. HOL has a special syntax that allows (*,**)prod to be written as (* # **), (*,**)sum to be written as (* + **), and (*,**)fun to be written

Table 2.3-3: HOL Type Operators

Operator	*Arity*	*Meaning*
`bool`	0	booleans
`ind`	0	individuals
`num`	0	natural numbers
`(*)list`	1	lists of type *
`(*,**)prod`	2	products of * and **
`(*,**)sum`	2	coproducts of * and **
`(*,**)fun`	2	functions from * to **

as `(* -> **)`.

2.3.2 THE PROOF SYSTEM.

HOL is not an automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

a. Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories contain the five axioms that form the basis of higher-order logic as well as a large number of theorems that follow from them.

b. Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher-order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

c. A collection of tactics. Examples of tactics include: `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition; `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms; and `EQ_TAC` which replaces a goal of the form $P \Longleftrightarrow Q$ with two subgoals, $P \Longrightarrow Q$ and $Q \Longrightarrow P$

d. A proof management system that keeps track of the state of an interactive proof session.

e. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

2.4 INTERPRETERS

The interpreter model, as defined in (ref. 14), can be used to describe state-transition systems. Interpreters can be differentiated from other models by their monolithic treatment of state and their flat control structure. The interpreter selects one of a fixed set of actions based upon the current state and returns a new state. The model incorporates four parts:

- A representation of the state **S**.
- A set of state transition functions defining the denotational semantics for each interpreter action (instruction) $\mathbf{J_i} : \mathbf{S_i} \times \mathbf{Env} \rightarrow \mathbf{S_j}$.
- A next state function that selects an appropriate state transition function given the current state $\mathbf{K} : \mathbf{S_i} \rightarrow \mathbf{J_i}$.
- A function **I**, relating the state at time $t + 1$ to the state and environment at time t using **K** and **J**.

$$\mathbf{I}[s, e] \equiv s(t+1) = (\mathbf{K}(s\ t))(s\ t)(e\ t)$$

Correctness is a relation between a specification and implementing interpreter such that:

$$\mathbf{I}_{impl}[s_m, e_m] \Longrightarrow \mathbf{I}_{spec}[s_n of, e_n of]$$

where f is a temporal abstraction function relating the differing specification and implementation time granularities. The state space s_n and environment e_n are abstractions of the implementing state space s_m and environment e_m, respectively.

Devices that provide a set of instructions such as a CPU or FPU can easily be modeled as interpreters. Other devices, such as a programmable interrupt controller (PIC) or DMA,

can also be modeled as interpreters with instructions. For example, we can model a PIC as an interpreter with instructions to:

- update the state due to an environment request.
- update a local register.
- provide status information.
- initiate communication with CPU.

A memory management unit can also be modeled as an interpreter with the following instructions.

- update a local register.
- provide status information.
- translate/validate memory requests.

2.4.1 COMBINING INTERPRETERS

When composing interpreters, we must show that devices synchronize and share information appropriately. The combined system aggregates the functions of the individual components and shields the communication between the devices. The interaction between interpreters can be made explicit by extending the interpreter model to more fully utilize environments. The selection of a transition function in the current interpreter model is dependent on both the state and (input) environment; however, an output environment is not present. By defining an output environment, it becomes more clear what is being shared and what is private state. Devices then communicate through environments where the output environment of one device is the input environment of another device.

The set of state transition functions defining the denotational semantics for each interpreter action is now defined as:

$$\mathbf{J_i} : \mathbf{S_i} \times \mathbf{Env_{in}} \rightarrow \mathbf{S_j} \times \mathbf{Env_{out}}.$$

The next-state function is also more clearly defined as:

$$\mathbf{K} : \mathbf{S_i} \times \mathbf{Env_{in}} \rightarrow \mathbf{J_i}.$$

As an example, consider the interrupt behavior of a system. The top-level system interpreter would respond to an environment of external world inputs while manipulating a state consisting of the system memory and the set of device registers (CPU, FPU, PIC, MMU, DMA, etc.).

The system implementation may rely on the interrupt function being provided by several interacting devices, including a programmable interrupt controller (PIC) and a CPU. The CPU input environment would include an interrupt request line, which the PIC will assert as part of its output environment when an external request (system environment) is pending.

The next-state function might be defined as below, where external interrupts (environment variables) cause state changes only when interrupts are enabled (state variable). This function definition states that when an interrupt is serviced, the program counter (pc) is pushed onto a stack and the new pc is assigned a value based on the current state and which external interrupt is requesting service.

The variables *intEnabled* and *pc* are part of the CPU state and the variable *PICReg* is part of the PIC state. The variable *interrupt* is part of the CPU's input environment and the PIC's output environment.

$$
\begin{array}{l}
(interrupt\ t\ \wedge\ intEnabled\ t) \\
\quad \rightarrow PushPC\ state\ t \\
\qquad \wedge\ pc(t+1) = IVEC\ (state\ t)\ (whichInt(t+1)) \\
\qquad \wedge\ intEnabled(t+1) = \mathbf{F} \\
\quad \mid\ ExecuteInstruction(state\ t) \\
\wedge \\
interrupt(t+1) = (externalInt\ t\ \vee\ internalInt\ t) \\
\wedge \\
whichInt(t+1) = PICPriority(PICReg\ t) \\
\ldots
\end{array}
$$

We may also wish to guarantee certain properties about the system, such as if an external I/O request is generated in the environment, eventually it will be serviced. This kind of

requirement may be part of an abstract model of fault tolerance that the interpreter must be shown to satisfy.

$$\forall io\, t.\, extIOInt\ io\ (env\, t) \Longrightarrow \exists t'.\, t \leq t' \wedge$$
$$interrupt\, t' \ \wedge\ whichInt\, t' = io$$

3.0 PROCESS ALGEBRAS

Process algebra is the study of concurrent communicating processes in an algebraic framework (ref. 21). Processes, in this context, are mathematical objects (like groups) that obey a set of axioms. CCS is recognized as having establishing the field. CSP is also a well-known example of a process algebra (ref. 22).

Composed systems are made up of a network of devices with a unity of purpose. The behavior of the system is, in some sense, its capability to communicate. From the large body of work on concurrency theory, process algebra has emerged as a framework to study the interaction of communicating, concurrent processes. While processes are generally thought of as executing programs, machines or devices can also be studied under this framework. The behavior of each device can be described as a process and the interaction of the devices studied as a process algebra. Hennessy defines processes as: " computational or algorithmic entities that when executed, affect and are affected by their environment. The environment may be other users, or other computational entities or a combination of both" (ref. 23).

Processes are expressed as terms in an algebra formed by an equational specification. The specification consists of a signature and a set of equations (Σ,E). The signature is a collection of formal function symbols or combinators that define the syntax of terms. The equations are axioms that help define the semantics of the objects. A number of other definitions are important for discussing the semantics:

- A Σ-*algebra* is an interpretation for the combinators over a set called the *carrier*. For example, given a signature with the combinators {*Zero,Succ,Pred,Plus*} , a Σ-algebra can be constructed with the carrier being the natural numbers ($< N, \Sigma_N >$).

- A *term-algebra* is an interpretation constructed from the combinators when the carrier is the set of terms constructed using the combinators.

- A Σ-*homomorphism* is a function between carriers of Σ-algebras that preserves the structure imposed by viewing them as Σ-algebras .

- A Σ-algebra , I, is *initial* for a class of Σ-algebras if for each class member, J, there is exactly one Σ-homomorphism from I to J.

- Given a Σ-algebra and an equational specification (Σ,**E**), the Σ-algebra is said to *satisfy* **E** if the equations are true for the carrier and interpretation. Or inversely, **E** *holds* for the Σ-algebra . The Σ-algebra is also called a *model* for **E**.

3.1 CALCULUS OF COMMUNICATING SYSTEMS

This section will briefly describe CCS without probing or examining the semantics involved in any detail. The semantic issues will be discussed in section 3.2, which describes process algebras in general.

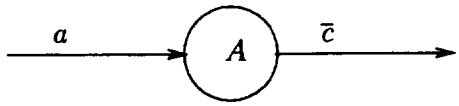
CCS was designed to model communication and concurrency in complex systems (ref. 3). Systems are composed of several parts acting independently of each other, but communicating with one another to achieve a mutual goal. All elements of the system are modeled as *agents*. The medium used to transmit information between a sender and receiver is also modeled as an agent. An agent's behavior is described by *actions*, which may be communications with other agents or independent concurrent actions. Milner suggests that independent actions can also be modeled as (internal) communication.

The calculus provides five primitives to construct expressions describing agents' behavior: **Prefix, Summation, Composition, Restriction and Relabelling.** In the pure calculus, agents do not transmit data values, but synchronize through indivisible actions where a synchronization signal is simultaneously sent by one party and received by another. The summation 'operator' can be applied to duplicate the effect of passing data values between agents. By communicating only through pure synchronizations, the calculus may ignore value variables. Restriction and relabelling operators have the tightest binding followed by prefix, composition, and summation.

An agent identifies the current state of an entity. Transitions from state to state are accomplished by an action. Actions are denoted by labels and describe either synchronization or internal transitions. Labels are taken from an infinite set of names. For two components to synchronize, they must share the same port name. The notation distinguishes send and receive synchronization labels by placing a bar over the name of send labels (e.g., $\overline{c}$). Internal transitions are labeled by the reserved label τ, which has no complement.

For example, in the figure below, agent A has an input synchronization port (`in`) and,

an output synchronization port ($\overline{\texttt{out}}$).



The behavior of the agent might be defined as follows:

$$A =_{def} \overline{p}.A'$$

$$A' =_{def} v.A$$

These agents describe a semaphore and show the use of the **Prefix** (".") operator. Agent A waits to synchronize with some other agent and, upon completion, behaves as A'.

Composition hooks together two independent agents and is denoted with a vertical bar. For example, consider the following agents:

$$A =_{def} a.A'$$

$$A' =_{def} \overline{c}.A$$

$$B =_{def} c.B'$$

$$B' =_{def} \overline{b}.B$$

The composite agent $(A \mid B)$ may be connected as follows:

We can also write $A =_{def} a.A'$ as $A \xrightarrow{a} A'$. The transitional semantics of the language (described below) allow us to infer the following:

a. From $A \xrightarrow{a} A'$, we infer $A \mid B \xrightarrow{a} A' \mid B$.

b. From $A' \xrightarrow{\bar{c}} A$, we infer $A' \mid B \xrightarrow{\bar{c}} A \mid B$ (this represents communication between A' and some agent other than B).

c. Since $A' \xrightarrow{\bar{c}} A$ and $B \xrightarrow{c} B'$ we can infer $A' \mid B \xrightarrow{\tau} A \mid B'$ (internal transitions).

d. From $B' \xrightarrow{\bar{b}} B$, we infer $A \mid B' \xrightarrow{\bar{b}} A \mid B$.

By imposing **Restriction** on the composition of A and B (i.e., $A \mid B\backslash c$), the action c is no longer externally available. This implies item (2) would then be excluded.

In the above examples, each agent was defined as conducting only one action. The **Summation** operator (+) is used to define agents that may make one of several (possibly infinite) transitions. For example, an agent similar to A could be defined to either accept input signals or send output signals as follows:

$$C =_{def} a.C + \bar{c}.C$$

Relabelling allows agent definitions to be reused. Using relabelling we could use the same agent definition for A and B above. Given a relabelling function f, $A[f]$ denotes a copy of agent A with appropriately modified action names.

3.1.1 TRANSITIONAL SEMANTICS

CCS provides several rules describing the semantics of the expression constructors:

Act

$$\overline{a.E \xrightarrow{a} E}$$

Sum$_j$

$$\frac{E_j \xrightarrow{a} E'_j}{\sum_{i \in I} E_i \xrightarrow{a} E'_j} (j \in I)$$

Com$_1$

$$\frac{E \xrightarrow{a} E'}{E \mid F \xrightarrow{a} E' \mid F}$$

Com$_2$

$$\frac{F \xrightarrow{a} F'}{E \mid F \xrightarrow{a} E \mid F'}$$

Com$_3$

$$\frac{E \xrightarrow{a} E' \; and \; F \xrightarrow{\bar{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

Res

$$\frac{E \xrightarrow{a} E'}{E\backslash L \xrightarrow{a} E'\backslash L}(a, \bar{a} \notin L)$$

Rel

$$\frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$$

Con

$$\frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'}(A =_{def} P)$$

3.2 SEMANTICS

Our primary concern regarding semantics pertains to the meaning of agent equivalence. Certainly different Σ-algebras will identify different machines and the sequence of actions a machine may perform. We want the equation axioms to restrict the set of models that satisfy the equational specification to those models that are real processes (devices).

As processes are inherently non-deterministic, the behavior of a process could be characterized by a set of action sequences (traces). One of the properties of a CCS term is to abbreviate this set. CCS terms also have an important dynamic aspect. Communication links and capabilities as well as the number of components, can all change as the system evolves.

For example, the agent $a.(b + c)$ can produce the trace set { (a,b), (a,c) }. Likewise, the agent $a.b + a.c$ produces the same trace [1]. The question then arises, are these agents the equivalent?

[1] For purposes of reducing syntax, we will frequently write $a.b$ as simply ab.

$$a(b+c) = ab + ac \tag{1}$$

Using a trace semantic definition of equivalence, the two agents are equal. However, this equivalence does not hold for all process algebras.

Of the five combinators, the summation function is the most open to different semantic interpretations: the restrict and relabel operators are clearly more of a convenience than a necessity; the composition operator can be defined in terms of the summation operator (see section 4.2); and the primitive nature of the prefix combinator provides little freedom for varying its meaning. To define the meaning of the composition operator, however, the timing of choices a system makes must be carefully considered. The agents in equation (1) illustrate this idea.

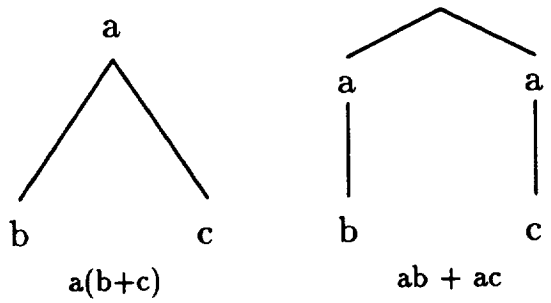


Figure 3.2-1: Example of Process Algebra Semantic Interpretations

The agent $ab + ac$ represents an agent that makes a choice to behave as the agent ab or the agent ac. If a specification equation were to declare that the prefix operator left-distributes over the summation operator, then the two agents would be equivalent. When this left-distribution axiom is not present, the agent $a(b+c)$ postpones the choice of a second action until after the action a occurs. Depending on when the choice between two traces is made, the behavior of two agents of can be quite different as viewed from an observer agent. In Figure 3.2-1 this difference in the moment of choice is exhibited by the difference in branching structure.

Adding the axiom *prefix left-distributes over summation* would allow many unrealistic process models to satisfy the equational specification, so it will not be present in the set of equations.

The notion of equivalence we will use is that of *bisimulation* (ref. 3). Two agents are equivalent by bisimulation, if an outside observer cannot distinguish between them. More formally, $P =_{bisimulate} Q$:

a. If $P \xrightarrow{\alpha} P'$ then for some Q', $Q \xrightarrow{\alpha} Q'$ and $P' =_{bisimulate} Q'$.

b. If $Q \xrightarrow{\alpha} Q'$ then for some P', $P \xrightarrow{\alpha} P'$ and $P' =_{bisimulate} Q'$.

Proofs of equivalence are done by constructing bisimulations. However, state explosion can quickly occur, making these proofs difficult. The solution is to use equational reasoning. This requires a relation where equivalence is a congruence (reflexive, symmetric, and transitive). Congruence on CCS agents is an equivalence relation satisfying the following set of properties:

if $A =_{equiv} A'$ and $B =_{equiv} B'$ then

a. $A + B =_{equiv} B + A$

b. $A|B =_{equiv} B|A$

c. $a.A =_{equiv} a.A'$

d. $A\backslash l =_{equiv} A'\backslash l$

e. $A[f] =_{equiv} A'[f]$

f. $A = E[A] \ \wedge \ B = F[B] \ \wedge \ \forall P \ E[P] = F[P] \Longrightarrow A =_{equiv} B$

The basic axioms of CCS are presented in Figure 3.2-2. Most variations of CCS incorporate these laws and differ in what equational laws regarding the internal action are included. A brief summary of six such process algebra variations is presented in (ref. 24).

3.2.1 INTERNAL ACTIONS

The internal action τ, also requires special attention to define when an agent chooses to take an internal action. As the semantics of τ vary, so does the meaning of equivalence relation and, thus, what interpretation models satisfy the process algebra defined. The basic issue is

Figure 3.2-2: Basic Axioms of CCS

summation (monoid) laws

$$P + Q = Q + P$$
$$P + (Q + R) = (P + Q) + R$$
$$P + P = P$$
$$P + 0 = P$$

composition laws

$$P|Q = Q|P$$
$$P|(Q|R) = (P|Q)|R$$
$$P|0 = P$$

restriction laws

$$(\alpha.Q)\backslash L = \left\{ \begin{array}{l} 0 \ \ if \ \alpha \in L \\ \alpha.Q\backslash L \ otherwise \end{array} \right.$$
$$(Q + R)\backslash L = Q\backslash L + R\backslash L$$
$$0\backslash L = 0$$

relabelling laws

$$(\alpha.Q)[f] = f(\alpha).Q[f]$$
$$(Q + R)[f] = Q[f] + R[f]$$
$$0[f] = 0$$

The Expansion Law

$$\alpha.P \mid \beta.Q = \alpha(P \mid \beta.Q) + \beta(\alpha.P \mid Q) + \left\{ \begin{array}{ll} \tau(P \mid Q) & if \ (\alpha = \overline{\beta}) \\ 0 & otherwise \end{array} \right.$$

how observable are internal actions to an outside observer. For example, we must consider whether the agent $a.P + \tau.b.Q$ is equivalent to the agent $a.P + b.Q$ (Figure 3.2-3 displays the branching structure of these agents).

A strict interpretation of these agents (using *strong equivalence*) would suggest the agents are not equivalent. The agent $a.P + b.Q$ is able to communicate with its environment through actions a or b. The agent $a.P + \tau.b.Q$, however, will either 1) only communicate through action a, or 2) only communicate through action b. Which behavior is observed depends on whether or not the internal action is taken.

A hierarchy of equivalences is presented in the Figure 3.2-4. The weakest equivalence is the trace equivalence definition where two processes are equal if the set of all possible traces that the two may execute is the same. Strong equivalence requires strict equality in what internal actions do.

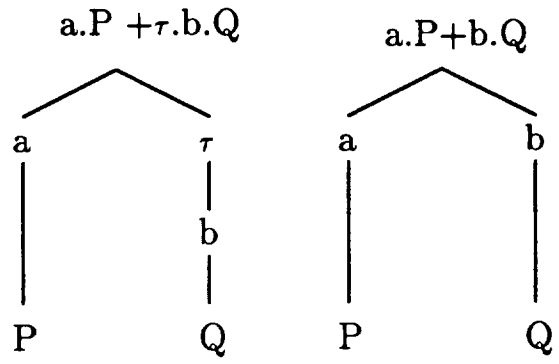


Figure 3.2-3: Internal Action Decision Point

The arrows in Figure 3.2-4 are meant to indicate that if two processes are strongly equivalent, then they are observationally congruent, Likewise, if two processes are observationally congruent, then they are observationally equivalent. Observational equivalence is a weaker form of bisimulation. Observational equivalence equates the agents $\tau.B$ and B while observational congruence makes a distinction between them. The τ laws for observational congruence that we will use are:

a. $\alpha.\tau.P =_{equiv} \alpha.P$

b. $P + \tau.P =_{equiv} \tau.P$

c. $\alpha(P + \tau.Q) + \alpha.Q =_{equiv} \alpha(P + \tau.Q)$

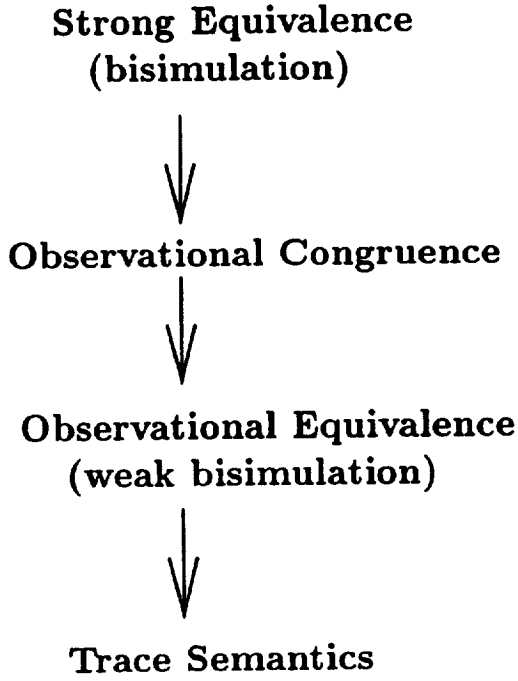


Figure 3.2-4: Process Algebra Equivalence Relations

4.0 MECHANIZATION OF THE PROCESS ALGEBRA

This section will describe preliminary work to represent CCS in HOL. There has been significant recent interest in mechanizing process algebras in HOL (refs. 25, 26 and 27). The development described here takes a purely definitional approach. This representation has allowed us to directly prove several of the CCS semantic laws. We are also interested in a vehicle to investigate modifications to the CCS logic to handle data passing as well as synchronization.

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 (ref. 28) 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 (ref. 29) 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.

As shown in Figure 4.0-5 from Melham's paper (ref. 29), the new type is defined by adding an axiom to the logic that asserts that the new type is isomorphic to an appropriate subset of an existing type. The predicate $P : ty \rightarrow bool$ defines some useful subset of the type ty.

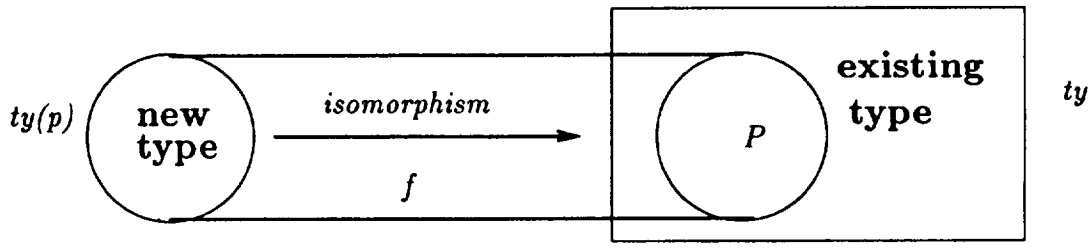


Figure 4.0-5: New Type Isomorphism

4.1 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 a *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';;
```

Several auxiliary recursive function definitions are defined for the *action* data type. An induction theorem for the type is provided and a simple theorem is proved showing that all actions must be either internal or external and not both.

```
⊢def (Is_Internal_Act INTERNAL = T)  ∧  (∀ l. Is_Internal_Act(EXTERNAL l) = F)
⊢def (Is_External_Act INTERNAL = F)  ∧  (∀ l. Is_External_Act(EXTERNAL l) = T)
⊢def ∀ l. LBL(EXTERNAL l) = SND l
⊢def ∀ l. TYP(EXTERNAL l) = FST l

ACTION_INDUCT =
    ⊢ ∀ P. P INTERNAL  ∧  (∀ p. P(EXTERNAL p)) ⟹ (∀ a. P a)

action_cases =
    ⊢ ∀ a. (Is_External_Act a  ∨  Is_Internal_Act a)  ∧
          ¬ (Is_External_Act a  ∧  Is_Internal_Act a)

IS_COMPL_ACT =
⊢ ∀ a b. IS_COMPL_ACT a b = (LBL a = LBL b)  ∧  (TYP a = ¬ TYP b)
```

4.2 AGENTS

The type representation covers the syntax of the concrete data type. Note that a term algebra is also formed. It would seem good practice for the representation definition of any type to 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 (ref. 21). This technique allows composed, concurrently executing agent expressions to be converted to an equivalent agent expressed only with the prefix and summation operators. We define three mutually recursive functions to replace the composition type constructor:

a. A communication operator, COMM, which declares that two agents will communicate if they have complementary, enabled actions.

b. A left-merge operator, LMERGE, which 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.

c. A compose ("merge" in the literature), COMPOSE, which creates an arbitrary interleaving of the two agents (through use of the summation type constructor).

The functions relate through the following laws. We will use the symbol "||", for the COMPOSE operator, the symbol "|" for the COMM operator and "L" for the left-merge operation.

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

   $P \mid Q = (a = \ COMPLEMENT\ b) \ \rightarrow \ \tau.(P' \parallel Q')\ else\ \mathbf{0}$

b. $x \parallel y = (xLy) + (yLx) + (x \mid y)$

c. $aLx = a.x$

d. $a.xLy = a.(x \parallel y)$

e. $(x + y)Lz = (xLz) + (yLz)$

f. $ax \mid bx = (a \mid b).(x \parallel y)$

g. $(x + y) \mid z = (x \mid z) + (y \mid z)$

h. $x \mid (y + z) = (x \mid y) + (x \mid z)$

The functions COMPOSE and LMERGE, are mutually recursive. Regrettably, HOL does not provide a mechanism for defining mutually recursive functions. To handle this prob-

lem, we include two auxiliary compound type operators in the type definition and axiomatize their semantic meaning (see the HOL code below).

Using the CCS restrict operator, a CCS agent can be constrained so that (a set of) specified actions cannot form links with other agents. The restrict operator in our grammer restricts an agent for a single action. This does not result in a significant loss of function; for example, the agent:

$$(a.A + b.B)\backslash\{a, c\}$$

can be defined by:

$$((a.A + b.B)\backslash a)\backslash c$$

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

let AGENT_INDUCT = save_thm('AGENT_INDUCT', prove_induction_thm AGENT);;
let AGENT_CASES = prove_cases_thm AGENT_INDUCT;;
let AGENT_CONSTRUCTORS_DISTINCT = prove_constructors_distinct(AGENT);;
let AGENT_one_one = prove_constructors_one_one 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)
```

While HOL allows the embedding of other logics, the native HOL syntax must be used. This makes CCS terms stated in HOL difficult to understand and awkward to use (as the above theorems show). For example, even the fairly simple agent expression: $(a.E+\overline{b}.0)|(\overline{a}.F)$ will be output as (and must be input as):

```
(COMPOSE (SUMM (PREFIX (EXTERNAL(T,'a')) E) (PREFIX)
(COMPOSE (SUMM (PREFIX (EXTERNAL(T,'a')) E)
                    (PREFIX (EXTERNAL(F,'b')) INACTIVE) )
          (PREFIX (EXTERNAL(F,'a')) F) )
```

To reduce the difficulty in manipulating embedded languages, new library facilities are available in HOL88 *Version* 2.0. The new library facilities include a "pretty printing" library and a "pretty parser" library for the HOL theorem-proving system. The use of these tools will be described below in sections 4.4 and 4.5.

4.3 TRANSITION SEMANTICS

Having defined the form of agents in the previous section, we define a relation between two agents that captures the semantic definition of labeled transitions (e.g., A $\xrightarrow{\alpha}$ B) This relation can be defined 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 (ref. 30). 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 all the laws described in 3.1, 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) "
    ];;
```

```
let TRANS_ACT =  (el 1 TRANS_rules);;
let TRANS_SUM1 = (el 2 TRANS_rules);;
let TRANS_SUM2 = (el 3 TRANS_rules);;
let TRANS_COM1 = (el 4 TRANS_rules);;
let TRANS_COM2 = (el 5 TRANS_rules);;
let TRANS_COM3 = (el 6 TRANS_rules);;
let TRANS_RES =  (el 7 TRANS_rules);;

let TRANS_INDUCT_TAC =  RULE_INDUCT_THEN TRANS_ind ASSUME_TAC ASSUME_TAC;;

let TRANS_CASES =  derive_cases_thm(TRANS_rules,TRANS_ind);;

let TRANS_ACT_EXISTS = new_definition('TRANS_ACT_EXISTS',
   "!P Q. TRANS_ACT_EXISTS P Q = ?P' Q' a. (TRANS P a P') /\ (TRANS Q a Q')");;
```

Using the definition of TRANS, we can prove that agents can evolve to other agents (e.g., liveness properties). The relation FBY, describes the transitive closure of the TRANS relation.

```
let (FBY_rules, FBY_ind) =
    let FBY = "FBY:agent->agent->bool" in
    new_inductive_definition false 'FBY'
    ("^FBY A B",[])
    [[
      %1--------------------------------------% ],
                "^FBY A A"
;
     [          "TRANS A a C";
                "^FBY C B"
      %2--------------------------------------% ],
                "^FBY A B"                          ];;

let FBY_REFL       = (el 1 FBY_rules);;
let FBY_TRANS      = (el 2 FBY_rules);;
```

4.4 PRETTY PRINTING SUPPORT

A set of rules can be defined so that the CCS terms are presented in a form resembling their form in the literature. Each rule consists of two parts, which are separated by the operator ->: a *pattern* and a *format*. Each *pattern* defines a context in which the rule is relevant and a template that a HOL term much match for the rule to be fired. The template may also contain optional type information. The *format* defines horizontal spacing parameters (e.g., where line breaks should appear) and how the HOL term should be printed. Frequently, this information informs the pretty printer that a new context should be entered. For example, the following rule will cause the pretty printer to to print the string `tau` for the internal action constant `INTERNAL`.

```
'term'::CONST(INTERNAL(),**) -> [<h 0> "tau"];
```

To activate a pretty printer specification, the follow HOL code must be executed. This code redefines the HOL system print function (`top_print`) to first attempt to print terms using the ccs_rules and if they fail, to use the built-in rules. The variable *assignable_print_term* must be reassigned so that the subgoals package also prints using the new rules.

```
top_print (\t. pp (ccs_rules_fun then_try
                   hol_term_rules_fun then_try
                   hol_type_rules_fun) 'term' [] (pp_convert_term t));;

let ccs_term_print t =
               pp (ccs_rules_fun then_try
                   hol_term_rules_fun then_try
                   hol_type_rules_fun) 'term' [] (pp_convert_term t);;

assignable_print_term := ccs_term_print;;
```

```
'lbls'::COMB(COMB(CONST(#'#(),**),*ch),CONST(#'#(),**))
              -> [<h 0> *ch ];
'lbls'::VAR(*str,**) -> [<h 0> "<" *str ">"];
'lbls'::CONST(*str,**) -> [<h 0>  *str];
'lbls'::COMB(COMB(CONST(STRING(),**),*ch),*str)
              -> [<h 0> *ch 'lbls'::*str];

'lbl'::COMB(COMB(CONST(#,#(),**),CONST(F(),**)),*str)
       -> [<h 0> "-" 'lbls'::*str];
'lbl'::COMB(COMB(CONST(#,#(),**),CONST(T(),**)),*str)
       -> [<h 0> 'lbls'::*str];

'term'::CONST(INTERNAL(),**) -> [<h 0> "tau"];
'term'::CONST(INACTIVE(),**) -> [<h 0> "0"];

'term'::COMB(CONST(EXTERNAL(),**),*label) -> [<h 0> 'lbl'::*label];
'term'::COMB(COMB(CONST(PREFIX(),**),*act),*ag) -> [<h 0> *act "." *ag];

'term'::COMB(COMB(CONST(COMPOSE(),**),*ag1),*ag2)
                  -> [<h 0> "(" *ag1 " || " *ag2 ")" ];
'term'::COMB(COMB(CONST(LMERGE(),**),*ag1),*ag2)
                  -> [<h 0> "(" *ag1 " L " *ag2 ")" ];
'term'::COMB(COMB(VAR(COMM(),**),*ag1),*ag2)
                  -> [<h 0> "(" *ag1 "|" *ag2 ")" ];

'term'::COMB(COMB(CONST(SUMM (),**),*ag1),*ag2) -> [<h 0> *ag1 "+" *ag2];
'term'::COMB(COMB(CONST(RESTRICT(),**),*lset),*ag) -> [<h 0> *ag "\" *lset];
'term'::COMB(COMB(CONST(RELABEL(),**),*ag),*rfun)
                  -> [<h 0> *ag "[" *rfun "]" ];
'term'::COMB(COMB(COMB(CONST(TRANS(),**),*ag1),*act),*ag2)
                  -> [<h 0> *ag1 " -<" *act ">--> " *ag2 ];
```

Figure 4.4-1: Pretty Printer Rule Specification

4.5 PRETTY PARSING SUPPORT

A generic parser-generator is provided as a library in HOL88 *Version* 2.0. The input to the generator is a form of modified BNF notation consisting of terminals, non-terminals, and action symbols. The parser-generator can only accept a syntax for a non left-recursive context-free grammer. Action symbols are embedded in the grammer to construct the intended semantics for the syntax.

For pretty parsing to work, a set of ML functions are defined, which are activated when the parser completes a grammer rule. These rules convert CCS terms into HOL terms.

Figure 4.5-1: Pretty Parser Grammer Specification

```
FIRST_CHARS 'a b c d e f g h i j k l m n o p q r s t u v w x y z
            A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
            * '.

CHARS 'a b c d e f g h i j k l m n o p q r s t u v w x y z
      A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
      1 2 3 4 5 6 7 8 9 0 *'.

USEFUL [('''', '''')].

MAIN_LOOP --> term [EOF].

term --> agent more_agent.

more_agent --> agent_op  more_agent | [].

agent --> [(] agent more_agent [)]
        | [0] {mk_const('INACTIVE', ":agent")}
        | action prefix
        | {mk_var(TOKEN,":agent")}.

action -->  [T]  { mk_const('INTERNAL',":action") }
          |      inlabel { mk_ccs_action( POP) }
          | [-] outlabel { mk_ccs_action( POP) }
          | {mk_var(TOKEN,":action")}.

prefix --> [.] {mk_ccs_act_comb('PREFIX',POP,agent)}.

agent_op --> [+] {mk_ccs_comb('SUMM',POP,agent)}
           |  [|] {mk_ccs_comb('COMPOSE',POP,agent)}
           |  [=] action [=] [>] {mk_ccs_trans(POP,POP,agent)}.

inlabel  -->  ['] { mk_ccs_label("T:bool", WORD) } ['].

outlabel -->  ['] { mk_ccs_label("F:bool", WORD) } ['].

inactive --> [0] {mk_const('INACTIVE', ":agent")}.
```

```
let mk_ccs_act_comb (op, a, b) = mk_comb(mk_comb(
                            mk_const(op,":action->agent->agent"),a), b);;

let mk_ccs_comb (op, a, b) = mk_comb(mk_comb(
                            mk_const(op,":agent->agent->agent"),a), b);;

let mk_ccs_label( b, w )=  mk_pair( b, mk_const('\''^w^'\'',":string") );;

let mk_ccs_action( l ) = mk_comb( mk_const('EXTERNAL',":label->action"),l);;

let mk_ccs_trans (A, act, B) = mk_comb(mk_comb(mk_comb(
                            mk_const('TRANS',":agent->action->agent->bool")
                           ,A), act), B);;

let SEPS = [('(',[]);(')',[]); ('.',[]);('-',[]);
            ('=',[]);('>',[]); ('+',[]);('T',[])];;

let parse thing = (PARSE_text(thing,[],SEPS));;

new_syntax_block('<<','>>','parse');;
```

When the HOL term parser finds a new syntax block (as defined above), the parser constructed is given the complete string within the syntax block (delimited by << and >>). The parser is expected to return the corresponding HOL term.

The following example demonstrates such a proof and show the use of the pretty parser and pretty printer.

```
let X = '('a'.E + -'b'.0) | (-'a'.F)';;
let Y = '(E | F)';;
let T = (X^'=T=>'^ Y);;
let Z = (PARSE_text( T,[],SEPS));;

X = '('a'.E + -'b'.0) | (-'a'.F)' : string
Y = '(E | F)' : string
T = '('a'.E + -'b'.0) | (-'a'.F)=T=>(E | F)' : string
Z = "('a'.E+-'b'.0 || -'a'.F) -<tau>--> (E || F)" : term

set_goal( [],"^Z");;

"('a'.E+-'b'.0 || -'a'.F) -<tau>--> (E || F)"

() : void
Run time: 0.3s

e(   RULE_TAC TRANS_COM3
     THEN EXISTS_TAC "'a'"
     THEN CONJ_TAC
     THENL [  RULE_TAC TRANS_SUM1  THEN RULE_TAC TRANS_ACT
            ; RULE_TAC TRANS_ACT ]);;

OK..
goal proved

|- TRANS
   (COMPOSE
    (SUMM(PREFIX(EXTERNAL(T,'a'))E)(PREFIX(EXTERNAL(F,'b'))INACTIVE))
    (PREFIX(EXTERNAL(F,'a'))F))
   INTERNAL
   (COMPOSE E F)

Run time: 0.1s
Intermediate theorems generated: 42


pp_thm (top_thm() );;

"('a'.E+-'b'.0 || -'a'.F) -<tau>--> (E || F)" : term
Run time: 0.0s
```

4.6 AGENT EQUIVALENCE

As described in section 3.0, 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.

```
new_type_abbrev('trace', ":action list");;

new_type_abbrev ('traces', ":trace set");;

let TR = new_recursive_definition false AGENT 'TR'
    "(TR(INACTIVE)      = {}:traces) /\
     (TR(PREFIX a A)    = { t | (HD t = a) /\ (TL t IN (TR A)) } ) /\
     (TR(SUMM A B)      = { t | (t IN TR(A)) \/ (t IN (TR B)) } )    ";;
```

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)"
    ];;

let OE_REF       = (el 1 OE_rules);;
let OE_PRE       = (el 2 OE_rules);;
let OE_LSUML     = (el 3 OE_rules);;
let OE_LSUMR     = (el 4 OE_rules);;
let OE_LSUM      = (el 5 OE_rules);;
let OE_RSUM      = (el 6 OE_rules);;

let OE_INDUCT_TAC =  RULE_INDUCT_THEN OE_ind ASSUME_TAC ASSUME_TAC;;

let OES_CASES =  derive_cases_thm(OE_rules,OE_ind);;
```

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:

a. 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.

b. 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.

c. 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)
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)

4.6.1 EQUIVALENCE PROOFS

Based on the above definitions of actions, agents, transitions, and equivalence, many of the CCS axioms can be proved. For example, in the box below, we show the proof script for the summation law: P + 0 = P

```
let SUM_INACTIVE = prove_thm('SUM_INACTIVE',
        "!P:agent. OE (SUMM P INACTIVE) P",
   GEN_TAC
   THEN RULE_TAC OE_LSUML
   THENL [ REWRITE_TAC [TRANS_ACT_EXISTS]
           THEN CONV_TAC NOT_EXISTS_CONV THEN GEN_TAC
           THEN CONV_TAC NOT_EXISTS_CONV THEN GEN_TAC
           THEN CONV_TAC NOT_EXISTS_CONV
           THEN REWRITE_TAC [DE_MORGAN_THM]
           THEN GEN_TAC
           THEN DISJ1_TAC
           THEN MP_TAC ( SPECL ["INACTIVE";"a:action"; "P'"] TRANS_CASES )
           THEN REWRITE_TAC [AGENT_CONSTRUCTORS_DISTINCT ]
         ;
           RULE_TAC OE_REF
         ] );;
```

4.7 EXTENSIONS

For our purposes, neither the recursive operator nor the relabel operator have been necessary. However, work is currently underway to add a repeat operator to the type definition. This operator is available in the π calculus developed by Melham (ref. 27). The relabel operator would be easy to add if warranted by further application of the calculus.

5.0 MECHANIZATION OF DEVICE INTERACTIONS

5.1 GENERIC INTERACTIONS

In this section, we will show how the process algebra formalism developed in section 4.0 can be used to specify various types of device 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:

a. Remote Procedure Call,

b. Message Passing,

c. Process Creation (fork), and

d. 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.

5.1.1 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. This example is somewhat simplified as the bus returns a data value rather than actually reading a memory location.

```
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;;
```

5.1.2 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.

Another example of the message passing form is the instruction-buffer management interaction between an 8088 CPU and its floating-point-instruction coprocessor (FPC), the 8087 (ref. 31). The CPU and FPC both maintain an instruction prefetch buffer. Instead of both devices duplicating all the instruction decode logic, the CPU sends messages to the FPC indicating how each byte should be interpreted (e.g., remove first byte, interpret byte as FPC instruction, flush the buffer, no change).

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^')';;
```

5.1.3 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. A DMA must be programmed to supervise the transfer of data from an I/O device to memory. The process of initializing and activating the DMA is a form of process creation. When the DMA has completed its task, the CPU (probably through the 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 ';;
```

Another example of rendezvous exists where a FPU must rely on the CPU for stores to and fetches from memory. Given an interleaved instruction stream, simultaneous execution by the two devices can occur; however, a rendezvous exists when results from one are needed by the other device.

5.2 AVM-1 /MMU CONNECTION SPECIFICATION

This section will describe the specification of a system consisting of the AVM-1 microprocessor and the MMU device verified in previous work (refs. 2 and 1).

5.2.1 AVM-1

The AVM-1 processor is described in detail in (ref. 14) and will only briefly be described here. AVM-1 was designed to demonstrate the use of generic interpreters in verifying hierarchically decomposed microprocessor specifications.

The design was motivated by the desire to provide hardware features that were not present in previous microprocessor verification efforts and would support a realistic operating system. In addition to vectored interrupts, AVM-1 supports a supervisor mode. AVM-1 has a load-store architecture based on a register file consisting of:

a. Register 0, which is read-only and contains the constant 0.

b. Seven supervisor-mode registers including a supervisor stack pointer. These registers are read-only unless the CPU is in supervisor-mode.

c. Twenty-four general purpose registers.

A program status word register maintains the status of the last ALU operation, whether interrupts are enabled, and the privilege level of the currently executing process. The program counter is also visible at the assembly language programmer level.

The instruction set contains 30 instructions and is similar to that of RISC I. For example the MOVE instruction is synthesized using an ALU operation. The instruction set is designed to support coprocessors. Six bits are reserved for the opcode. If the top opcode bit is high, the processor assumes the instruction is intended for a coprocessor.

5.2.2 ABSTRACT MMU

A description of a number of memory management units which form a complexity hierarchy appears in (ref. 2). By developing a sophisticated MMU in steps, the construction of the final proof appears to be more tractable. The simpler devices validate access to fixed length memory pages while the more complex devices authorize read, write or execute access to variable length segments and translate virtual addresses to real addresses. Many of these devices were designed and verified to the gate-level. However, as the complexity increases,

the emphasis of the verification shifts from gate level connections to the correctness of the operating-system support features.

The abstract MMU validates memory requests based on information maintained in a memory-resident segment descriptor table. The location of the table is determined by a segment-table pointer register, which is accessible only during supervisor operations. Each descriptor consists of two words: the first contains access control information (present bit, read/write/execute permissions, segment size) and the second serves as the base address for the segment's real location in memory. To translate from a virtual address to a real address, the MMU adds the segment offset to the segment base address. The MMU assumes the table provides an entry for all possible segment descriptors.

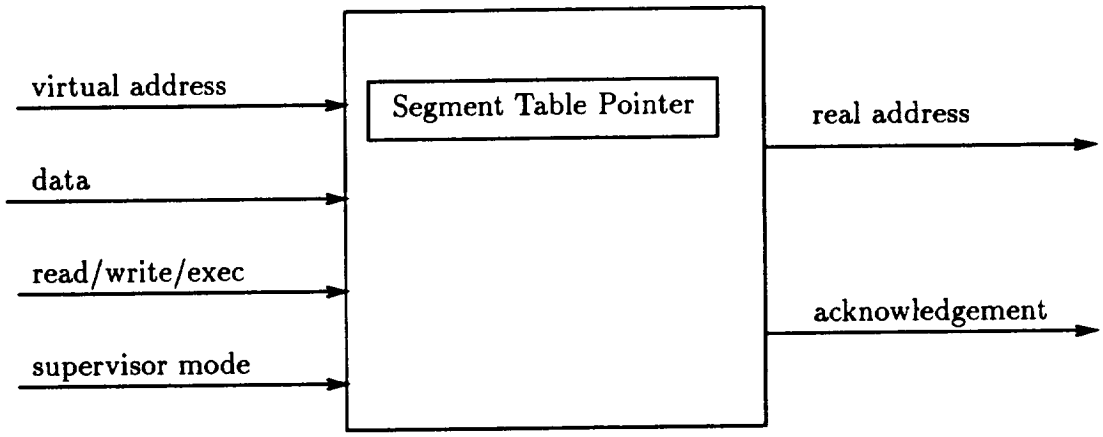
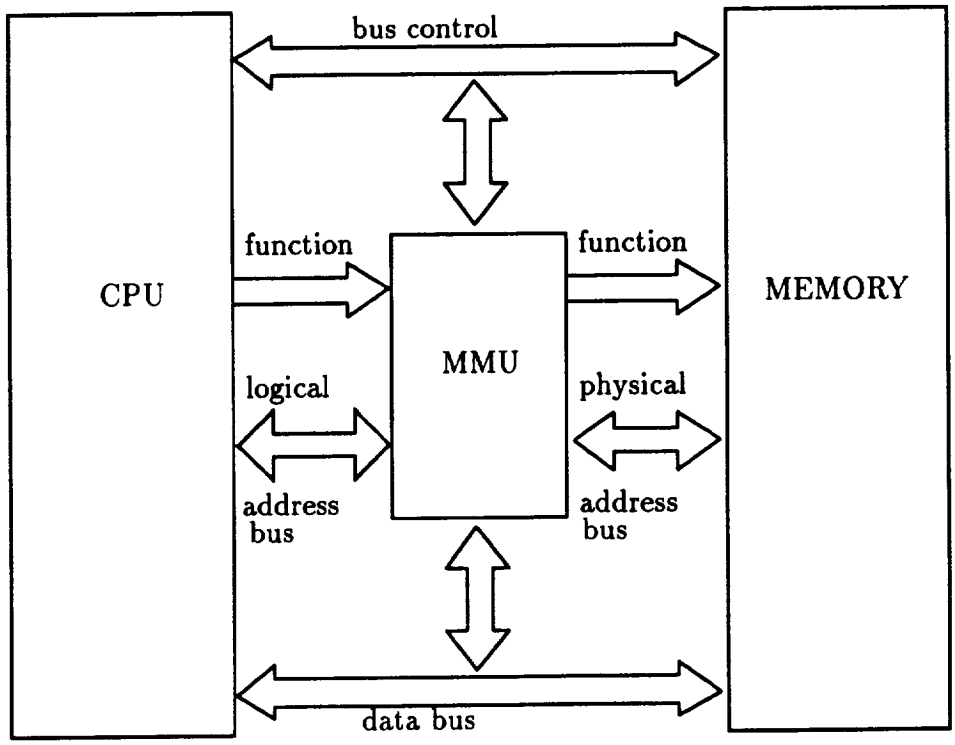


Figure 5.2-1: Abstract MMU Environment/State

The abstract MMU representation generalizes traits particular to concrete implementations. A generic theory for a class of MMU devices is defined where several functions and data types are left abstract. Using an abstract representation, details such as word length can be omitted and the verification focuses only on the correctness of higher level abstraction (e.g., electronic block level rather than gate level). Other properties such as the exact security policy and division of a virtual address into a segment identifier and offset are left unspecified. At a later point, the abstract representation can be instantiated with components that implement concrete behavior.

5.2.3 CPU MMU INTERACTION



To determine how the 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. 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.

The MMU specification is in several parts as listed below.

```
⊢def legalAccess r vAddr tblPtr rwe mem =
  let a = (fetch r)(mem,(address r)((add r) (segIdshf r vAddr,tblPtr))) in
  ((validAccess r) (vAddr,a,rwe)    ∧   (ofsLEq r) (vAddr,a))

⊢def vToR r vAddr tblPtr mem =
  let a = (fetch r) (mem, (address r) ((add r)( (wordn r 1),
              (add r)(segIdshf r vAddr,tblPtr) ))) in
 (address r) ((add r) (segOfs r vAddr, a))

⊢def superMode r vAddr rwe tblPtrADDR tblPtr data mem =
  ((wBIT rwe)  ∧  (addrEq r (vAddr,tblPtrADDR)))
     → ( T, vAddr, data ) |  ( T, vAddr, tblPtr )

⊢def userMode r vAddr rwe tblPtrADDR tblPtr data mem =
  legalAccess r vAddr tblPtr rwe mem
     → ( T, (vToR r vAddr tblPtr mem), tblPtr )
     |  ( F, vAddr,                    tblPtr )

⊢def mmu_spec r vAddr rwe tblPtrADDR tblPtr data mem superv =
  superv → superMode r vAddr rwe tblPtrADDR  tblPtr data mem
         | userMode  r vAddr rwe tblPtrADDR  tblPtr data mem
```

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:

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

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

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

$$\mathsf{MMU} = \mathsf{MMUtoCPU.MMU}$$

$$\mathsf{System} = \mathsf{CPU} \mid \mathsf{MMU}$$

5.3 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:

a. If either of the CPU actions, $\overline{userMode}$ or $\overline{superMode}$, are enabled, the memory subsystem will engage in communication.

b. 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 the process algebra terms described in section 5.2.3. 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[2]. 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;;
```

[2]As with many such definitions, it is perhaps easier to read the code from the bottom upwards. This gives the reader a top-down viewpoint.

To express the goal in a general form, several auxiliary definitions are defined.

a. A recursive definition (ENABLED) is defined to construct a list of all output actions that an agent can perform.

b. A new relation FLUX states that two composed systems are members of the relation only if a sequence of internal actions can be found to relate the first to the second. FLUX is similar to TRANS except that the laws TRANS_COM1 and TRANS_COM2 are not present.

c. 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 (FLUX_rules, FLUX_ind) = let  FLUX = "FLUX:agent->action->agent->bool" in
    new_inductive_definition false 'FLUX' ("^FLUX x a y",[])    [
 % ACT LAW %
[ ],
   %--------------------------------------%
            "^FLUX (PREFIX a y) a y"
;% SUML LAW %
[           "^FLUX x a y"
   %--------------------------------------%
            "^FLUX (SUMM x z) a y"
;% SUMR LAW %
[       "^FLUX x a y" ],
   %--------------------------------------%
             "^FLUX (SUMM z x) a y"
;% COM LAW %
       [   "^FLUX x (EXTERNAL (T,s)) x'";
           "^FLUX y (EXTERNAL (F,s)) y'" ],
   %--------------------------------------%
    "^FLUX (COMPOSE x y) INTERNAL (COMPOSE x' y')"
;% RES LAW %
       [   "^FLUX x a y";
           "~( (LBL a) = (SND (r:label)))" ],
   %--------------------------------------%
         "^FLUX (RESTRICT r x) a (RESTRICT r y) "
;% TRANSITIVE LAW %
    [        "^FLUX A a C";
             "^FLUX C a B"
   %--------------------------------------% ],
             "^FLUX A a B"                   ];;

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

The success agent for the composed MMU-CPU is $SUCCESS \mid SUCCESS$. By unwinding the agent definitions and using the FLUX 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)
```

5.3.1 SYSTEM SPECIFICATION WITH A DMA DEVICE

The DMA we present is based on the device verified in (ref. 32), which provides four channels between memory and I/O devices. The behavior of each channel is determined by a set of registers. For present purposes, we will consider a DMA with a single write-to-memory channel. The register set for a channel consists of:

a. A counter register, which, when reaching zero, interrupts the CPU to indicate the channel has completed its task.

b. A memory address counter, which must be initialized to the base address in memory for the block transfer. After each I/O transfer, this register is incremented.

c. A control/status register, set by the CPU to activate a channel and unset by the DMA when the channel has completed its task.

Two system configurations are possible. The DMA could reside between the CPU and the MMU or between the MMU and memory. The proposed system places the DMA before the MMU (see Figure 5.3-1). Thus, DMA writes to memory are validated by the MMU.

The DMA will respond to three events: a read DMA register request, a write DMA register request, and an I/O device service request (once enabled). An I/O device service request requires that a value be written to memory and that when the channel counter reaches zero, the CPU be interrupted. Below we present the process algebra terms for this behavior. A more complete CPU agent is also defined to accept interrupts from the DMA.

writeReg = $write.address.data$

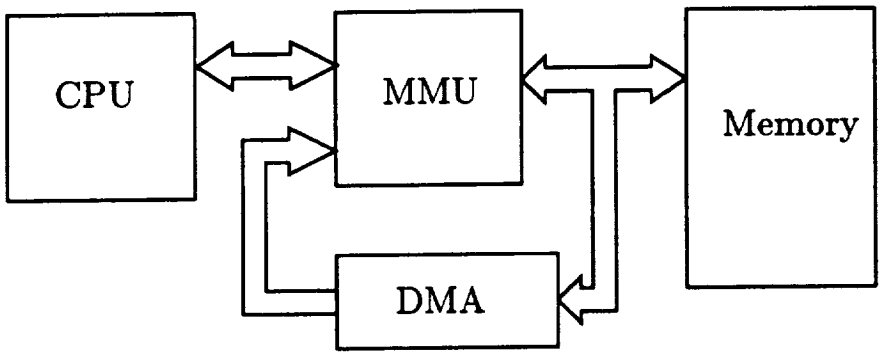


Figure 5.3-1: CPU/MMU/DMA System

$\mathsf{readReg} = read.address.\overline{data}$

$\mathsf{activeIO} = \mathsf{writeReg.DMA} + \mathsf{readReg.DMA} +$
$(ioInt.\overline{superMode}.\overline{write}.\overline{address}.\overline{data}.(ack + nack).(\mathsf{DMA} + \tau.\overline{cpuInt}.\mathsf{DMA})$

$\mathsf{DMA} = writeReg + readReg + \tau.\mathsf{activeIO}$
$\mathsf{CPU} = (\mathsf{CPUtoMEM} + \tau.cpuInt).\mathsf{CPU}$

$\mathsf{System} = \mathsf{CPU} \mid \mathsf{MMU} \mid \mathsf{DMA}$

6.0 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:

a. An additional type constructor for recursive agents. This constructor will be necessary to show the equivalence of infinite agents.

b. Extending the calculus so that reasoning about the higher-level semantics of composed agents can be performed.

c. Greater proof support. Many of the proofs performed appear fairly easy to understand outside of the formal model. As with most formal proofs, many details that appear trivial require a fair amount of work to verify. The pretty-parser and pretty-printer support are helpful; however, due to the underlying representation for agents, large terms must be manipulated by hand. The automation of more of the tedious aspects of the proof will allow formal reasoning about much larger examples to take place.

d. Several extensions to the calculus are being examined, including the addition of a data field to all actions. With this addition, conditional operator constructs could be formed (e.g., $if\ (val > 5)\ then\ Agent = A\ else\ Agent = B$). The addition of a labeled τ action might also help in understanding the semantics of an agent.

REFERENCES

1. P. J. Windley, "Formal Proof of the AVM-1 Microprocessor Using the Concept of Generic Interpreters," NASA Contractor Report 187491, March, 1991.

2. E. T. Schubert, "Verification of Memory Management Units using HOL," Tech. Rep. CSE-90-27, University of California, Davis, August 1990.

3. R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

4. A. Pitts, "Concurrency Theory Lectures, *Cambridge University,*" 1990.

5. W. R. Bevier, W. A. Hunt, and W. D. Young, "Toward Verified Execution Environments," *IEEE Symposium on Security and Privacy*, 1987.

6. P. G. Neumann, "On Hierarchical Design of Computer Systems for Critical Applications," *IEEE Transaction on Software Engineering*, vol. SE-12, No. 9, September 1986.

7. J. D. Guttman and H.-P. Ko, "Verifying A Hardware Security Architecture," *IEEE Symposium on Research in Security and Privacy*, 1990.

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

9. 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.

10. W. A. Hunt, "A Verified Microprocessor," Tech. Rep. 47, The University of Texas at Austin, Dec. 1985.

11. W. A. Hunt, "Microprocessor Design Verification," *Journal of Automated Reasoning*, vol. 5, 1989.

12. W. J. Cullyer, "Implementing Safety Critical Systems: The VIPER Microprocessor," in *VLSI Specification, Verification, and Synthesis* (G. Birtwhistle and P. Subrahmanyam, eds.), pp. 1–25, Kluwer Academic Press, 1988.

13. A. Cohn, "A Proof of Correctness of the VIPER Microprocessor: the Second Level," in *Current Trends in Hardware Verification and Automated Theorem Proving* (G. Birtwhistle and P. Subrahmanyam, eds.), pp. 1–91, Springer-Verlag, 1989.

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

15. J. J. Joyce, "Totally Verified Systems: Linking Verified Software to Verified Hardware," *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, July 1989.

16. M. Gordon, "HOL: A Proof Generating System for Higher-Order Logic," in *VLSI Specification, Verification, and Synthesis* (G. Birtwhistle and P. Subrahmanyam, eds.), pp. 73–128, Kluwer Academic Press, 1988.

17. A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification using Higher Order Logic," in *From HDL Descriptions to Guaranteed Correct Circuit Designs* (D. Borrione, ed.), Elsevier Scientific Publishers, 1987.

18. A. Church, "A Formulation of the Simple Theory of Types," *Journal of Symbolic Logic*, vol. 5, 1940.

19. M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF.* Lecture Notes in Computer Science No. 78, Springer Verlag, 1979.

20. R. L. Constable, *Implementing Mathematics with the NUPRL Proof Development System.* Prentice Hall, 1986.

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

22. C. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

23. M. Hennessy, *Algebraic Theory of Processes.* MIT Press, 1988.

24. R. DeNicola, P. Inverardi, , and M. Nesi, *Using the Axiomatic Presentation of Behavioral Equivalences for Manipulating CCS Specifications.* Lecture Notes in Computer Science No. 407, Springer Verlag, 1989.

25. A. J. Camilleri, "Mechanizing CSP Trace Theory in Higher Order Logic," *IEEE Transactions on Software Engineering*, vol. 16, September 1990.

26. A. Camilleri, P. Inverardi, and M. Nesi, *Combining Interaction and Automation in Process Algebra Verification.* Lecture Notes in Computer Science No. 494, Springer Verlag, 1991.

27. T. Melham, "A Mechanized Theory of The π-Calculus in HOL," Tech. Rep. 244, Computer Lab, University of Cambridge, 1992.

28. J. Goguen and J. Meseguer, "An Initiality Primer," 1983.

29. 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.

30. T. Melham, "A Package for Inductive Relation Definitions in HOL," HOL System Documentation, 1991.

31. E. T. Schubert, "Towards Composition of Verified Hardware Devices," Tech. Rep. CSE-91-16, University of California, Davis, January 1991.

32. S. Kalvala, "Design and Verification of a DMA Processor," Tech. Rep. CSE-90-27, University of California, Davis, August 1990.