Type Theory and Concurrency

Rance Cleaveland
Prakash Panangaden

TR 85-714
December 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

The burgeoning interest in concurrent computation has sparked an increased interest in theoretical models of concurrency. While standard sequential programming has a well-understood semantics and proof theory, the nondeterministic nature of concurrency has made a similar understanding of concurrent programming extremely difficult. Much interesting work in the field has been done, and much remains yet to be done; it is our intention in this paper to present a different kind of model of concurrency, a type-theoretic one, which we hope will shed light on reasoning about concurrency.

We encode the synchronization tree model of Milner's CCS as a type in the Nuprl Type Theory. This is a constructive type theory equipped with a rich collection of inference rules for reasoning about types. We relate the equality in the type of synchronization trees with various behavioral equivalences. We also discuss the relation between the logic induced by our models and various modal logics for reasoning about concurrency.

Introduction

The burgeoning interest in concurrent computation has sparked an increased interest in theoretical models of concurrency. While standard sequential programming has a well-understood semantics and proof theory, the nondeterministic nature of concurrency has made a similar understanding of concurrent programming extremely difficult. Much interesting work in the field has been done, and much remains yet to be done; it is our intention in this paper to present a different kind of model of concurrency, a type-theoretic one, which we hope will shed light on reasoning about concurrency.

The type-theoretic models in this paper revolve around giving a type for processes, and they are developed in a style compatible with the Nuprl type theory, which has evolved at Cornell. The models we present are based on Milner's Calculus of Communicating Systems, henceforth called CCS. This is an extremely general formalism for describing systems comprising autonomous computating agents that may interact with each other through the exchange of synchronizing signals. We have opted for a type-theoretic approach for several reasons. First, the theory is equipped with a an extremely expressive logic, a logic which appears to subsume every modal logic of concurrency we have encountered. This paper details a development of the Hennessy-Milner logic using the type theory and our model, and preliminary investigations indicate that other logics have similar encodings. In particular, Regular Trace Logic and Temporal Logic have straightforward expressions in terms of the model and the Nuprl logic. Aditionally, types exhibit built-in equalities between members of the type, and the logic provides machinery for reasoning about equality. Both of the models presented here have equalities that correspond to interesting behavioral equivalences on processes; one model, in fact, exhibits Milner's strong congruence as the equality between its elements. Using the

Nuprl machinery, then, we can reason in a very natural fashion about process equivalence. The Nuprl logic forms the basis for the Nuprl proof development system which has been implemented at Cornell. The availability of such a system greatly facilitates the development of correct proofs in which some degree of automatic assistance is given to the user. Finally, the Nuprl system has aspects of a program-synthesis system, where programs are extracted from the constructive proofs of theorems. We believe that such automated synthesis has promise in the area of concurrency; using the Nuprl theory to discuss concurrency represents a first step in this direction.

This paper is divided into several sections. The first two present overviews of CCS, the formalism which is a basis for the models presented, and the Nuprl type theory. We then present one type-theoretic model, the acceptance model, and examine its relationship to existing models of concurrent processes. The shortcomings of this model are pointed out, and the second model is presented. Properties of this model are examined, including the relationship between this model's equality and strong equivalence; we then examine some examples of reasoning about traditional properties of interest to designers of concurrent systems in the Nuprl setting, and we conclude with a section which describes an encoding of the Hennessy-Milner logic in type theory.

An Overview of CCS

The Calculus for Communicating Systems (CCS) is an algebraic formalism for reasoning about concurrent and nondeterministic processes. A complete presentation of the calculus may be found in [Milner80]; extensions to the formalism (the extended formalism is called SCCS, for Synchronous Calculus of Communicating Systems) are described in [Milner83]. Informally CCS provides a mechanism for describing *agents,* which perform actions that may include interaction with each other. The nature of these actions is left as unspecified as possible; the only requirement is that each action (except for a distinguished *silent* action, τ) have an action, called an *inverse*, with which it synchronizes. Processes in general may choose from several different actions at any point during their computations, and choosing an action may give rise to nondeterminism in that the same action may lead to different courses of computation. The number of different choices available at any given point in the process execution is finite.

To define CCS formally we first must describe the set *Action*. *Action* must satisfy the following two conditions.

(i) There is a distinguished action, τ, in *Action*.

(ii) If $a \in Action$ and $a \neq \tau$ then so is $\underline{a}$.

τ is called the "silent" action. Intuitively, one may think of τ as being an internal transition or a communication between two different components of a system.
The actions a and $\underline{a}$ are called complements of each other and represent actions which can synchronize. Given *Action*, the CCS terms may be described by the following grammar.

$$
\begin{array}{lll}
P ::= & nil & \\
\mid & aP & a \in Action \\
\mid & P + P & \\
\mid & P|P & \\
\mid & P\backslash S & S \subseteq Action \\
\mid & P[L] & L \text{ a relabelling}
\end{array}
$$

The *nil* process performs no actions and terminates. aP is a process which performs an a action and then goes on to behave like P. $P+Q$ may behave like P or Q; it may perform any of the first actions available to P or Q, after which it behaves like one or the other. $P|Q$ represents the concurrent execution of P and Q. Initially this composite agent may perform any first action available to P or Q, after which it may perform either the first action of the process not selected in the first step or the second action available in the process which was selected in the first step. Additionally, at any step in the compuation, if P may perform an action a and Q may perform $\underline{a}$, the inverse of a, then $P|Q$ may perform τ, a *synchronization*; the next actions available to $P|Q$ after a τ are the next actions available to P and Q. Thus, in CCS, concurrent execution is represented by interleaving the atomic actions. $P\backslash S$ corresponds to the restriction of P with respect to a set S of actions; it behaves like P except that it may not perform any sequence of actions headed by an action in S. Finally, if $Action_1$ and $Action_2$ are two action sets and $L{:}Action_1 \rightarrow Action_2$ is a mapping such that $L(\tau) = \tau$ and $L(\underline{a}) = \underline{L(a)}$ for any $a \in Action_1$, then L is called a relabelling and $P[L]$ is the relabelled process.

Processes may also be defined by *guarded* recursion, which is to say that no head recursive definitions are allowed; every summand in a recursive definition of a process must be headed by an action. Mutual recursion is also allowed. In general processes may run infinitely.

An important property of the concurrent composition operator is captured in the following theorem due to Milner.[Milner80] $\Sigma a_i P_i$ is shorthand for $a_1 P_1 + \ldots + a_n P_n$.

Expansion Theorem: If $P = \Sigma a_i P_i$ and $Q = \Sigma b_i Q_i$ then

$$P|Q = \Sigma a_i(P_i|Q) + \Sigma b_i(P|Q_i) + \Sigma\tau(P_i|Q_j)\{\text{if } a_i = \underline{b_j}\}.$$

This makes manifest the idea that concurrency is being represented by nondeterministic interleaving of the atomic actions.

The formalism of CCS supports an algebraic style of reasoning. Equivalences between processes are defined and axiomatized, and reasoning is carried out equationally. [Milner80] describes two notions of process equivalence which we shall discuss in this paper. The first is *strong* equivalence, which we may define in the following fashion. First, we say that $P \rightarrow^a P'$ if $P = aP' + Q$ for some Q; extending this in an obvious way, if $s = a_1 \ldots a_n$ then $P \rightarrow^s P'$ if $n > 0$ and there is a P''

such that $P \rightarrow^{a_1} P''$ and $P'' \rightarrow^{a_2 \ldots a_n} P'$ or if $n=0$ and $P=P'$. Now we describe a nested hierarchy of relations which in the limit will be strong equivalence.

$P \sim_0 Q$.

$P \sim_{n+1} Q$ if and only if for all $s \in Action^*$, if $P \rightarrow^s P'$ then there is a Q' such that $Q \rightarrow^s Q'$ and $P' \sim_n Q'$ and if $Q \rightarrow^s Q'$ then there is a P' such that $P \rightarrow^s P'$ and $P' \sim_n Q'$.

$P \sim Q$ if and only if for all n, $P \sim_n Q$.

The second equivalence, *observational* equivalence, is essentially an extensional (in the sense of extensional equality between functions) version of strong equivalence. To define it we first introduce the notion of an observable experiment. Let $s \in (Action\text{-}\{\tau\})^*$ with length n. $P \Rightarrow^s P'$ if there is an $s' \in Action^*$ and integers m_0, ..., m_n such that $s' = \tau^{m_0} s_0 \tau^{m_1} \ldots \tau^{m_{n-1}} s_{n-1} \tau^{m_n}$; that is, s' is exactly s with some number of τ's between every action in s. Essentially, $\Rightarrow$ is just $\rightarrow$ with τ's ignored. We may now define a nested hierarchy of relations whose infinite intersection will be observational equivalence.

$P \approx_0 Q$.

$P \approx_{n+1} Q$ if and only if for all $s \in (Action\text{-}\{\tau\})^*$, if $P \Rightarrow^s P'$ then there is a Q' such that $Q \Rightarrow^s Q'$ and $P' \approx_n Q'$ and if $Q \Rightarrow^s Q'$ then there is a P' such that $P \Rightarrow^s P'$ and $P' \approx_n Q'$.

$P \approx Q$ if and only if for all n, $P \approx_n Q$.

Both equivalences are axiomatized (to varying degrees) in [Milner80].

Congruences also play an important role in the proof theory. Briefly, an equivalence is a congruence such that if $C[]$ is a context and $p=q$, then $C[p]=C[q]$. Strong equivalence is a congruence, but observational equivalence is not. Accordingly, in [Milner80] observational congruence, which is the weakest congruence that is strictly finer than observational equivalence, is defined and axiomatized.

We shall have occasion to refer to the SCCS composition operator, $\otimes$, as defined in [Milner83]. $\otimes$ is a synchronous parallel composition; in $P \otimes Q$ the actions in $P \otimes Q$ are required to happen in lock-step synchrony. To define it we first need to strengthen the *Action* set in the following fashion. We require that $(Action, 1, \_, \circ)$ be an Abelian group, with $a \circ \underline{a} = 1$ representing the synchronization of a and $\underline{a}$ and $a \circ b$ representing the lock-step execution of a and b. Now if $P = \Sigma a_i P_i$ and $Q = \Sigma b_j Q_j$ then

$$P \otimes Q = \Sigma(a_i \circ b_j)(P_i \otimes Q_j).$$

Typically, models for CCS describe processes as *synchronization trees*, which are unordered, finitely branching trees of actions. At each node of the tree the branching represents the possible choices to the process being modeled, while the paths in the tree represent possible execution sequences for the process represented by the synchronization tree. The composition of processes corresponds to an operation defined on such synchronization trees. The model we describe in this paper is obtained by encoding synchronization trees as *types*, whose elements reflect the tree structure.

Introduction to Type Theory

The type theory we use in this paper comes from the Nuprl system developed at Cornell. The theory is explained fully in [Prl Staff86]; we will only discuss pertinent points here. Intuitively a type is a collection of elements exhibiting a common structure; typically, one thinks of integers as forming a type and of pairs of integers as forming another type. Functions from one type to another also constitute a type distinct from from either component type, and the disjoint union of two types also defines a type. Propositions also correspond to types, as we shall see, so type theory represents a convenient formalism for discussing *both objects and logics for reasoning about objects*. It is this aspect of type theory that we shall exploit in our model of CCS.

The Nuprl type theory resembles that developed by Martin-Lof in [Martin-Lof82]. The type structure may be defined inductively in the following fashion. The type of integers,*int*, and the empty type, *void*, form the basis for the inductive definition. If A is a type and $B[a/x]$ is a type for every a in T (where substitution for free variable x is defined in the usual sense), then so are the *dependent product* and the *dependent function* types, $x{:}A\#B$ and $x{:}A \rightarrow B$. Intuitively, the dependent product type defines the type of pairs where the type of the second element of each pair depends on the value of the first element of the pair, while the dependent function type denotes the type functions whose codomain type may vary for each possible value of the domain. If x is not free in B then B must be a type, and the two types described above correspond to the standard Cartesian product and function space types; these will be written $A\#B$ and $A \rightarrow B$, respectively. The *disjoint sum* of types A and B, written $A|B$, has $inl(a)$ for a in A and $inr(b)$ for b in B as its elements; this constructor may actually be defined in terms of the dependent product and any two-element type with a decidable equality. Other type constructors will be described later, but a complete understanding of them requires first an understanding of the propositions-as-types principle, which we will now describe.

Just as propositions are either true or not true, so are types either inhabited or not inhabited, and it is the identification between truth and inhabitation which allows one to translate logic into type theory. Given this identification, the types and type constructors presented so far exhibit pleasing logical characteristics. *void* corresponds to *false*. The type $A\#B$, where the absence of a binding variable in front of A indicates that B has no free variables and is hence a type, is inhabited exactly when both A and B are, just as the proposition $A\&B$ is true exactly when both A and B are. Type $A|B$ is inhabited exactly when either A or B is, just as $A \vee B$ is true (constructively) when either A or B is, and $A \rightarrow B$ is inhabited exactly when there is a function from A to B, just as $A \Rightarrow B$ is true exactly when, from the truth of A, one can deduce the truth of B. A is empty exactly when the type $A \rightarrow void$ is inhabited, just as A is false when $\neg A$ (which translates constructively to $A \Rightarrow false$) is true. Even more surprising is the correspondence between dependent products and existentially quantified statements and dependent functions and universally quantified statements. $x{:}A \# B$ is inhabited exactly when there is a pair $<a,b>$ such that a is in A and b is in $B[a/x]$, just as $\exists x{:}A.B$ is true exactly when there is an a in A such that $B[a/x]$ is true. $x{:}A \rightarrow B$ is inhabited exactly when there is a function mapping each element a of A to an inhabiting element of $B[a/x]$, just as $\forall x{:}A.B$ is true exactly when for each a in A $B[a/x]$ is true. In fact, the entire constructive predicate calculus can be translated into type theory.

The correspondence between propositions and types enables now a discussion of three other kinds of types: *equality* types, *set* types and *quotient* types. Equality types have the form $a = b \ in \ T$ and are well-formed when T is a type and a and b are elements of T; they are inhabited (by a token named *axiom*) exactly when a and b are equal in the given type. Equality in the types described so far are as expected; it should be noted that two functions are equal if they are extensionally equal. Set types have the form $\{x{:}A \mid B\}$, where A is a type and $B[a/x]$ is a type for each a in A, and comprise elements of A satisfying B interpreted as a proposition. Quotient types have the form $(x,y){:}A//B$, where A is a type and for all a and b in A, $B[a,b/x,y]$ is a type (proposition) which is an equivalence relation in x and y (reflexive, symmetric and transitive), and allow equality to be redefined for the base type (A in this case). That is, $a = b \ in \ (x,y){:}A//B$ exactly when a and b are of type A and $B[a,b/x,y]$ is true (inhabited).

The Nuprl logic also allows a form of recursive type definition [CM85]. For our purposes recursive types have the syntactic form $rec(x.T)$ and represent "solutions" to equations of the form $x = T(x)$.

The proof theory of the Nuprl logic also mirrors proof theory of the constructive predicate calculus, given the propositions-as-types principle. The entire development will not be presented here (the interested reader is referred to [Prl Staff86]), but it suffices to say that the rules for forming types and demonstrating their inhabitation correspond precisely with rules for forming propositions in the constructive predicate calculus and proving their truth. The Nuprl proof development system also provides a mechanism for implementing derived rules of inference and proof strategies called *tactics*. The programming language ML [GMW 79] serves as the metalanguage for expressing tactics.

A Type of Finite Sets

In order to define synchronization trees in a compact fashion we introduce the *finset* type constructor. Given an arbitrary *base* type, this constructor will represent a type whose elements are the finite sets of elements of the base type. Intuitively these sets will correspond to the images of functions which map finite sets of integers into the base type. Accordingly, we make the following definitions.

$\{1, \ldots, n\} = \{i{:}int \mid 0 < i \leq n\}$

That is, $\{1, \ldots, n\}$ represents the type of all integers between 1 and n, inclusive. In the case that $n = 0$, $\{1, \ldots, n\}$ is an empty type. We also assume the existence of projection functions *first* and *second* for product types; these may easily be defined in the Nuprl theory.

$$\begin{aligned} finset(A) = (f,g){:}(n{:}int \# \{1, \ldots, n\} \rightarrow A) \,//\, & \forall m_1{:}\{1, \ldots, first(f)\}.\exists n_2{:}\{1, \ldots, first(g)\}. \\ & \quad second(f)(m_1) = second(g)(n_2) \text{ in } A \\ & \quad \& \; \forall m2{:}\{1, \ldots, first(g)\}.\exists n_1{:}\{1, \ldots, first(f)\}. \\ & \quad\quad second(g)(m_2) = second(f)(n_1) \text{ in } A \end{aligned}$$

The quotient equivalence says that two elements of *finset(A)* are equal exactly when they have the same images; this is exactly the extensional set equality. We shall have occasion to refer to *first(S)* for S in *finset(A)* as the *cardinality bound* of S and to *second(S)* as the *element function* of S.

We should note here that we have adopted this representation of finite sets rather than the more usual characteristic function representation so that *finset* may be used to construct recursive types. Briefly, for a Nuprl recursive type definition to be valid it must satisfy the positivity requirements set forth in [CM85]; that is, the type being defined must not appear in the antecedent of any arrow.

We now show how to define certain set-theoretic operations in this type. We start with union.

$$P \cup Q = \langle first(P) + first(Q), \lambda x. if\ 0 < x \leq first(P)\ then\ second(P)(x)\ else\ second(Q)(x - first(P)) \rangle.$$

Performing the union of two sets in this scheme involves adding together the bounds on the sets' cardinalities and concatenating their element functions.

$$s \in S = \{i{:}\{1, ..., first(S)\} \mid second(S)(i) = s\ in\ A\}$$

Given the propositions-as-types principle, it suffices to give a type which contains an element when $s \in S$ and is empty otherwise. If s is in the range of S's element function then the type given above will contain an element in the inverse image of s; otherwise, the type will be empty. For this function to be computable the equality in the base type A must be decidable.

Defining intersection in this scheme is tricky but not conceptually difficult. It does involve some rather uninteresting details of the Nuprl theory, however, so rather than present a formal account of intersection we will sketch how intersection may be implemented. Given P and Q which are finite sets over a base type A (with decidable equality), we may systematically check whether elements in P are in Q by iterating through {1, ..., *first(P)*} and asking if $second(P)(i) \in Q$. If so then increase the bound of $P \cap Q$ and update the element function of $P \cap Q$ accordingly.

Other set operations such as set difference may be defined similarly.

The Acceptance Model

Our goal is to give a type-theoretic model of CCS which allows us to reason about properties of concurrency using the Nuprl logic. Our first model somewhat resembles the failures model of CCS and TCSP due to Brookes and the acceptance model for CCS due to [Hennessy83] in that we represent processes as sets of pairs, where the first element of each pair contains a finite execution sequence of the process being modeled and the second element contains a structure which lists a finite number of actions. Unlike the failures model our "action sets" specify potentially allowable actions which the process may perform next, and unlike the acceptance model we demand that our "next action" structure contain more information about the structure of the tree being represented. Following Winskel and and Milner we shall sometimes refer to processes as *synchronization trees*, where the nodes of the tree represent states and the labeled transitions between nodes correspond to the actions the process may perform to change state.

The model translates synchronization trees into sets of pairs, where each pair contains a finite execution history, hereafter called a *trace*, and a set, which we call a *next set*, whose elements consist of all next actions which the process (tree) can perform. Intuitively the elements of a next set represent all the actions which may be chosen after following one of (potentially) many paths labeled by the corresponding trace in the synchronization tree. That there can be more than one such path results from the inherent nondeterminism present in the CCS formalism; depending on the choices made, the same trace can give rised to different sets of next allowable actions.

Expressing the Acceptance Model as a Type

Formally a process P is a subset of *(Action list)* x 2^{Action} satisfying the following conditions:

(1) ε is a trace of P;

(2) xa is a trace of P if and only if x is a trace of P and a belongs to a next set connected to x;

(3) all next sets are finite.

Semantically, we interpret next sets as being complete accounts of the possible actions which a process may perform after executing the associated trace. That is, if a and b are actions such that they both may follow a trace s but the availability of one action means the other is not available then a and b must belong to different next sets attached to s. (In synchronization tree terms, if a and b belong to different subtrees headed by s then a and b must belong to different member sets.). Otherwise a and b belong to the same next set.

Conditions (1) and (2) state that every process must have a starting point and that at any point in a process's computation the only available actions come from the appropriate next set and that all actions in the next set are in fact available, while condition (3) enforces finite choice at every choice point. Our interpretation of next sets is what differentiates this model from other acceptance models. This requirement essentially enables us to keep track of the fact that nondeterministic choices may have been made during the course of a computation and that as a result certain sets of actions may or may not be possible. •

To express the class of processes as a type in this framework we use the *finset* constructor described above to define the intermediary notion of *computation*, where a computation is a trace-next set pair. Accordingly, given a type *Action* with appropriate structure, define *comps* as follows.

comps = Action list # finset(Action)

A single process will be a subtype of *comps* satisfying properties (1)-(3). Therefore, the following definition of *Proc* may be made. (It should be noted here that U_1 denotes the type of all previously described types which do not involve U_1, and that it therefore makes sense, in light of the propositions-as-types principle, to define predicates on a type T as functions from T to U_1. That is, predicate P is true of a exactly when $P(a)$ is an inhabited type.)

$$Proc = \{P{:}comps \to U_1 /$$
$$\exists s{:}finset(Action).P(<\varepsilon, s>)\ \&$$
$$\forall x{:}comps.\ P(x) \Rightarrow \forall a{:}second(x).\exists s{:}finset(Action).P(<first(x).a, s>)\}$$
$$//\ \forall x{:}comps.\ P(x) \Leftrightarrow Q(x)$$

That is, *Proc* comprises predicates on *comps* such that conditions (1) and (2) above are met. (The structure of *comps* guarantees that condition (3) will be met.) Our intention is that *Proc* correspond to the type of sets of *comps* with the appropriate restrictions; therefore, we must impose an extensional equality on the type of predicates. We now define the set-theoretic operations we need in the discussion which follows. Given P and Q in type *Proc*,

$P \cup Q = \lambda x{:}comps.P(x) | Q(x)$; and

$x \in P \Leftrightarrow P(x)$.

$P \cup Q$ is a predicate on *comps* which is true exactly when either P or Q is true; therefore, $P \cup Q$ has the right semantics. $x \in P$ exactly when $P(x)$ is true (in the language of type theory, when $P(x)$ is inhabited), so the membership predicate is properly defined.

We will have occasion to refer to all the traces of a process. Thus we define $Traces(P) = \{s : \text{for some } N <s,N> \in P\}$.

Some simple examples of CCS processes translated into our model follow.

$$a(b+c) \quad = \{<\varepsilon, \{a\}>, <a, \{b,c\}>, <ab, \{\}>, <ac, \{\}>\}$$

$$ab + ac \quad = \{<\varepsilon, \{a\}>, <a, \{b\}>, <a, \{c\}>, <ab, \{\}>, <ac, \{\}>\}$$

$a(b + \tau b) = \{<\varepsilon, \{a\}>, <a, \{b, \tau\}>, <ab, \{\}>, <a\tau, \{b\}>, <a\tau b, \{\}>\}$

We should note that invisible actions are explicitly represented in our model and that a pair whose next set is empty signifies a terminated computation.

Operations as Type Constructions

In this section we describe how Milner's CCS operations on processes may be implemented in our model. We also discuss the representations in our model which arise when additional operations on processes such as those described in [HBR84] and SCCS are allowed. In what follows we shall assume that the set-theoretic operations used have been implemented in the type-theoretic model as described above.

nil: In CCS *nil* represents the process which does nothing except terminate. In our model,

$$nil = \{ <\varepsilon, \{\}> \}$$

aP: Given a process P we may represent the result of prepending an action a onto the beginning of P as follows:

$$aP = \{ <\varepsilon, \{a\}>\} \cup \{ <as, N> : <s,N> \in P\}.$$

P\a: In CCS one may provide for actions to be hidden in a process. Milner calls this hiding "restriction"; in synchronization tree terms, $P\backslash a$ corresponds to the tree P with all subtrees headed by the action a deleted. In our model we represent the restriction of process P with respect to action a as follows:

$$P\backslash a = \{<s,N>:<s,M> \in P \text{ and } s \text{ contains no occurrences of } a \text{ and } N = M\text{-}\{a\}\}.$$

Relabeling: Relabeling provides one with a convenient mechanism for changing the action set (the "sort", in CCS terms) of a process. A *relabeling* from action set A to action set B refers to a function $S{:}A \rightarrow B$ such that $S(\tau) = \tau$ and such that $\underline{S(a)} = S(\underline{a})$. We can also extend S to map strings in A^* to strings in B^* in a natural way.

Therefore, given a process P and a relabeling S, we define the relabeling of P via S in the obvious way.

$P[S] = \{ <s,N> :$ for some $<s',N'> \in P$ $s=S(s')$ and $N=S(N) \}$,

where we extend S to map strings to string and sets to sets in the obvious way.

Nondeterministic choice: $P+Q$ represents a process which may behave like either P or Q; furthermore, the environment initially has some say as to which process gets mimicked. That is, initially the first actions of both P and Q are available; thereafter, either P or Q becomes the process which $P+Q$ behaves like. Given processes P and Q we represent P + Q as follows:

$$P + Q = \{ <\varepsilon, M'> : <\varepsilon,M> \in P \text{ and } <\varepsilon,N> \in Q \text{ and } M'=M \cup N\} \cup$$
$$\{ <s,M> : s \neq \varepsilon \text{ and } <s,M> \in P \text{ or } <s,M> \in Q \}.$$

Composition: Given processes P and Q we define $P|Q$ by first defining some intermediate notions. If s_1 and s_2 are strings let $shuffle(s_1, s_2)$ represent the set of all possible shufflings of s_1 and s_2; we assume that we can tell which actions came from which component string in the members of $shuffle(s_1, s_2)$. We now define a finite family of sets in the following way:

$$S_0 = shuffle(s_1, s_2).$$

$$S_{i+1} = \{x\tau y : xa\underline{a}y \in S_i \text{ for some } a \in Action \text{ and } a, \underline{a} \text{ from different strings}\}$$

Now define

$$Comp(s_1, s_2) = \cup S_i$$

and

$$Synch(M, N) = M \cup N \cup \{\tau\} \qquad \text{if } a \in M \text{ and } \underline{a} \in N \text{ some } a$$
$$= M \cup N \qquad \text{otherwise.}$$

We are now in a position to define $P|Q$ in the following fashion:

$$P|Q = \{ <s, M'> : s \in Comp(s_1, s_2) \text{ and } M' = Synch(M, N) \text{ for some}$$
$$<s_1, M> \in P, <s_2, N> \in Q\}.$$

We have now defined all CCS operations in terms of our model. We now turn our attention to describing the TCSP operations of [HBR84].

Interleaving: $P|||Q$ represents the process which can perform any interleaving of events from P and Q. Given our definition of *shuffle* above we define $P|||Q$ in the obvious way.

$$P|||Q = \{<s,M'>: s \in shuffle(s_1,s_2) \text{ and } M' = M \cup N \text{ for some } <s_1,M> \in P, <s_2,N> \in Q\}$$

Parallel Composition: In TCSP $P||Q$ represents the process which corresponds to the simultaneous execution of P and Q, provided P and Q are performing the same actions. We define P||Q as follows:

$$P||Q = \{<s, M'>: M' = M \cap N \text{ for some } <s,M> \in P, <s,N> \in Q\}$$

Indeterminate choice: $P \oplus Q$ represents a process which may behave like P or Q. It differs from $P+Q$ in that the environment has no say as to which process will be simulated. We define this as

$$P \oplus Q = P \cup Q.$$

So far we have concentrated on formalisms which characterize essentially asynchronous parallelism. Our model, however, is sufficiently expressive that we can capture the notion of synchrony as well. We will exhibit this by giving an account in our model of Milner's SCCS.

The essential difference between CCS and SCCS comes to light in the definition of composition. In CCS events that happen in two processes that are being composed happen asynchronously, except when an action and its complement synchronize. In SCCS, on the other hand, corresponding events in composed processes must execute in lock-step; that is, if $P \rightarrow^a P'$ and $Q \rightarrow^b Q'$ then $P \otimes Q \rightarrow^{a \circ b} P' \otimes Q'$. The notion of complementary action still exists, as $a \circ \underline{a}$ reduces to τ (1 in [Milner83]). With this in mind, we turn our attention to modeling Milner's SCCS product. We first define a label set L which satisfies the abelian group axioms under $\circ$, where $a \circ b$ represents the action corresponding to the synchronous execution of a and b and τ is the unit. Such a definition of L is possible in the Nuprl system[Prl Staff86]. Given two strings s_1 and s_2, we further define

$$
\begin{aligned}
Glue(s_1,s_2) &= \varepsilon && \text{if } s_1=\varepsilon \text{ or } s_2=\varepsilon \\
&= (a\circ b)Glue(x,y) && \text{if } s_1=ax \text{ and } s_2=by
\end{aligned}
$$

With this definition we can define $P \otimes Q$ as follows:

$$
\begin{aligned}
\mathrm{P}\otimes\mathrm{Q} = \{<s,M'> :&\text{ for some } <s_1,M> \in P,\ <s_2,N> \in Q,\ s = Glue(s_1,s_2) \text{ and} \\
&M' = M \times N\},
\end{aligned}
$$

where MxN represents the Cartesian product of M and N.

Type Equality in Relation to Milner's Equivalences

In Martin-Löf's conception of types each type comes equipped with an equality relation between members of the type. This relation is an essential part of the definition of the type. The Nuprl system incorporates rules for reasoning about the equality relation in types. It is of interest to see, therefore, what equality relation on synchronization trees is induced by the type definitions we have given. In particular, we would like to find some correspondence between our equality and the various equivalences that have been defined by Milner to reason about behavioral aspects of processes.

We now turn our attention to the behavioral equivalence which the standard notion of set equivalence induces on our model. It turns out that our equivalence is strictly finer than Milner's $\approx_1$ and is incomparable with the rest of Milner's behavioral equivalences, including observation equivalence.

Theorem 1: $=$ is strictly finer than $\approx_1$.

Proof: If $P=Q$ then $Traces(P) = Traces(Q)$, implying that $P \approx_1 Q$. Since $a(b+c) \neq ab+ac$ but $a(b+c) \approx_1 ab+ac$, strictness follows.

□

In fact the behavioral equivalence corresponding to equality in our model is fine enough for us to give accounts of other models which have been proposed for CCS. From the information contained in next sets, for instance, we can derive the failure sets which form the foundation of the failures model presented in [HBR84] in the following fashion. Given a particular process P and an experiment s, let $N_1, \ldots, N_k$ be the next sets associated with s in our model. The maximal failures set, F, associated with s then is $F = \cup((\cup N_i) - N_i)$, and we are thus in a position to discuss failures equivalence in this model. That is, the equivalence in the acceptance model is at least as fine as failures equivalence. In the spirit of [DH82] we can also talk about *must* sets and *may* sets and the equivalence, called *testing* equivalence, which these sets induce on CCS expressions. Given a trace s in a process P and its associated next sets $N_1,\ldots,N_k$, we define $Must(s,P) = \cap N_i$ and $May(s,P) = \cup N_i$. Using these definitions we can now speak of testing equivalence; acceptance equivalence is at least as fine as testing equivalence.

If our equivalence fit into the Milner observational equivalence hierarchy then our work would jibe very nicely with work done to date. Unfortunately, the next result demonstrates that such is not the case.

Theorem 2: $=$ is incomparable with $\approx_2$.

Proof:

$= \not\supset \approx_2$: Consider the trees $P = a(ab+ac)$ and $Q = aab+aac$. Clearly $P=Q$ and $\neg(P \approx_2 Q)$.

$\approx_2 \not\supset =$: Consider the trees $P = \tau a$ and $Q = a$. $P \approx_2 Q$, and yet since in our model $P = \{<\varepsilon, \{\tau\}>, <,\{\tau\}>, <\tau, \{\}>\}$ while $Q = \{<\varepsilon, \{a\}>, <a,\{\}>\}, P \neq Q$.

□

In fact the following corollary holds.

Corollary: $=$ is incomparable with observational equivalence.

□

The essential problem is that observational equivalence is not a congruence relation. It does turn out, however, that equality in our model represents a behavioral congruence with respect to CCS, which is to say that if two processes exhibit the same representation in our framework then they may be used interchangeably in any context.

Theorem 3: $=$ is a congruence.

Proof: If $P = Q$ then it follows from our definitions that for all processes R, $P+R=Q+R$, $P|R=Q|R$, and $P \otimes R = Q \otimes R$ and that for all $a \in L\ aP = aQ$.

□ .

Additionally, equality in our model fits into the strong equivalence hierarchy developed by Milner.

Theorem 4: $\sim_1 \supset = \supset \sim_2$.

Proof:

$\sim_1 \supset =$: $\sim_1$ is just trace equality, so if $P=Q$ then $P \sim_1 Q$. Strictness follows from the fact that $a(b+c) \sim_1 (ab+ac)$ but $a(b+c) \neq (ab+ac)$.

$= \supset \sim_2$: Suppose $P \sim_2 Q$. We must show that $P=Q$. Assume that $<s,N> \in P$. Now, since $s \in Traces(P)$ and $P \sim_2 Q$, for all P' such that $P \rightarrow^s P'$ there is a Q' such that $Q \rightarrow^s Q'$ and $P' \sim_1 Q'$ (and vice versa). Associated with each P' is a set N' such that $<\varepsilon,N'> \in P'$; furthermore, since in our model of CCS ε may only have one next set associated with it (since a process corresponds to a rooted synchronization tree), if $P' \sim_1 Q'$ then $<\varepsilon,N'> \in Q'$. Now N must be associated with some P', and hence some Q', meaning that $<s,N> \in Q$. Thus $P \subseteq Q$. A symmetrical argument establishes that $Q \subseteq P$, and we therefore have that $P=Q$. Strictness follows from the fact that $a(ab+ac) = (aab+aac)$ but $\neg[a(ab+ac) \sim_1 (aab+aac)]$.

The Recursive Model

The acceptance model is conceptually simple, and the implementations of the CCS operations are straightforward, but the equality in the model does not correspond to an expressive enough behavioral equivalence for our purposes. Furthermore, the inductive definition of CCS operations becomes obscured in this model, and it is difficult to reason inductively about the behavior of processes.

The model we are about to present derives from recent extensions to the Nuprl type theory and exhibits both a pleasing equality and a simple structure. Its only drawback involves its inability to account for nonterminating processes; this drawback is only temporary, however, as implementation continues on the Nuprl project. Formally, we represent the class of all processes as a recursive type, where the type encompasses essentially labeled trees. This type, *ST*, solves the following fixed-point equation:

$$\mathrm{ST} = \mathrm{finset}\,(Action \times \mathrm{ST}),$$

and is written in Nuprl as *rec(ST.finset(Action* $\times$ *ST))*, where *finset* is a defined type constructor which takes a type and constructs the type of all finite sets of the given type and *Action* is the type of all actions. Some examples of specific CCS expressions represented in this model follow.

$$\begin{aligned}
nil &= \{\} \\
a(b+c) &= \{ <a,\ \{<b,\{\}>, \\
&\qquad\qquad <c,\{\}>\} \\
&\quad \} \\
ab + ac &= \{\ \ <a,\{<b,\{\}>\}>, \\
&\qquad <a,\{<c,\{\}>\}> \\
&\quad \}
\end{aligned}$$

CCS Operations

We present the definitions in the model for choice and composition.

Choice: $+ : ST \times ST \rightarrow ST$ is given by

$$P + Q = P \cup Q.$$

Composition: $| : ST \times ST \rightarrow ST$ is given by

$$\begin{aligned} P \mid Q = \quad & \cup\{<a, s'|Q>: \ <a, s'> \in P\} \ \cup \\ & \cup\{<b, P|t'>: \ <b, t'> \in Q\} \ \cup \\ & \cup\{<\tau, s'|t'>: \text{ there is an } <a, s'> \in \mathrm{P} \text{ and } <\underline{a}, t'> \in Q\}. \end{aligned}$$

We may also define SCCS composition, given an *Action* set with the appropriate structure, and the TCSP composition and choice operators. To define SCCS composition we postulate, as before, a group *Action*, where $\circ$ is the group operation and τ is the group identity. With this definition in hand we now are in a position to define SCCS compostion.

$$\begin{aligned} P \otimes Q \ &= \ \{\} \text{ if } P = \{\} \text{ or } Q = \{\} \\ &= \{<a \circ b, P' \otimes Q'> \mid <a, P'> \in P \text{ and } <b, Q'> \in Q\}. \end{aligned}$$

The other CCS operations and the TCSP operations may be defined in a similar fashion. Care must be taken, however, in giving an account of $\oplus$ to ensure that the processes being composed remain completely distinct.

Equivalence

Type equality in this model corresponds exactly to strong equivalence, as the following inductive argument shows. We first note, however, that in this model, unlike in the others we have discussed, the notion of subprocess is explicitly present in the type model.

Theorem 5: Equality in the type model is exactly strong equivalence.

Proof: Assume $P = Q$. The proof that $P \sim Q$ follows by induction on n, the level of n-equivalence.

Base: $n=0$. Trivially, $P{\sim}_0Q$.

Induction: Assume for all $m<n$ that $P{\sim}_mQ$; it remains to show that $P{\sim}_nQ$. Now, if $P{\rightarrow}^sP'$ then there is a Q' such that $Q{\rightarrow}^sQ'$ and $P'=Q'$ *in* the type model. The induction hypothesis guarantees that $P'{\sim}_{n-1}Q'$, and by symmetry the symmetrical result holds if $Q{\rightarrow}^sQ'$. Thus by definition $P{\sim}_nQ$.

Now assume that $P{\sim}Q$; we must show that $P=Q$. The proof will proceed by induction on the structure of P. If P or Q is the null process then $P=Q=Nil$. Now suppose P is non-null. The induction hypothesis allows us to assume that for all subtrees P' of P and Q' of Q that if $P'{\sim}Q'$ then $P'=Q'$. Let $<a,P'> \in P$; we must show that $<a,P'> \in Q$. Since $P{\rightarrow}^aP'$ there is a Q' such that $Q{\rightarrow}^aQ'$ and $P'{\sim}Q'$. By induction, then, $P'=Q'$ and $<a,P'> = <a,Q'> \in Q$. Thus $P{\subseteq}Q$. A similar argument establishes that $Q{\subseteq}P$, and thus $P=Q$.

□

Examples of Synchronization Problems in CCS

This section describes specifications of standard synchronization problems and properties of concurrency in CCS. We assume here full CCS, which is to say that processes may be nonterminating. We begin our study with a presentation of a binary semaphore in CCS. (This example comes from [Milner80].) Suppose process p_1 has critical section $\alpha\beta$ and p_2 has critical section $\gamma\delta$; the implementation of binary semaphore should enforce the atomicity of these critical sections. Consider the following definitions:

$$p_1 = \pi\alpha\beta\phi\, p_1$$

$$p_2 = \pi\gamma\delta\phi\, p_2$$

$$sem = \underline{\pi\phi}\, sem$$

$$q = (p_1|sem|p_2)\setminus\{\pi, \phi, \underline{\pi}, \underline{\phi}\}.$$

Intuitively π and ϕ corresponds to the P and V operations on a semaphore, and *sem* is a process which implements a semaphore. To see that mutual exclusion is enforced, consider the form of process q. Using the expansion theorem and the definition of restriction, it is easy to see that

$$q = (\tau\alpha\beta\tau + \tau\gamma\delta\tau)\, q;$$

in other words, the critical sections of p_1 and p_2 are executed atomically.

Deadlock also has a natural definition in CCS. Intuitively, deadlock happens when two or more processes cannot execute and have not terminated. To capture this formally in CCS we first define a set of *synchronization* actions, $Syn \subseteq Action$. Syn will typically contain events corresponding to *send* and *recieve* or *signal* and *wait*. These events cause the possibility of deadlock in that without these actions deadlock would not be possible. Processes p_1 and p_2 are deadlocked if it is the case that neither process is *nil* and

$$(p_1|p_2) \setminus Syn \sim nil.$$

That is, if we force synchronization events to synchronize then two processes are deadlocked if neither process can perform an action. We can generalize this definition to an arbitrary number of processes in the obvious way.

Our last example details a specification of the readers-writers problem. This problem is paradigmatic of many database problems. The problem is this: given some number of processes which wish to *read* from a database and some number of processes which wish to *write* to a database, enforce the condition that at most one writer may be writing at any time and that any number of readers may be reading at any time, provided that no process is writing at that time. We will define a process *sched* which enforces this condition, provided that readers issue β_r (for "begin read") before each read operation and ϕ_r (for "finish read") after each read and that writers issue corresponding β_w and ϕ_w before and after their writes. Consider the following definitions.

$writer_i = \beta_w\, w_b w_f\, \phi_w\, writer_i$

$reader_i = \beta_r\, r_b r_f\, \phi_r\, reader_i$

$sched = (\underline{\beta_w}\, \underline{\phi_w} + R)\, sched$, where

$R = \underline{\beta_r}\underline{\phi_r} + \underline{\beta_r}R'\underline{\phi_r}$

$R' = \underline{\beta_r}\underline{\phi_r}R' + \underline{\beta_r}R\underline{\phi_r}$.

A specification of the readers-writers problem would then be

$(| \; writer_1 \mid \ldots \mid writer_n \mid sched \mid reader_m \mid \ldots \mid reader_1) \setminus$

$\{\beta_r, \beta_w, \underline{\beta_r}, \underline{\beta_w}, \phi_r, \phi_w, \underline{\phi_r}, \underline{\phi_w}\}$.

Arguing for the correctness of this specification is hard to do withing the CCS formalism; in fact, arguing about the correctness is most naturally carried out in the metasystem in which CCS resides. Informally one can argue that this is correct by noting that *sched* allows a writer to write exactly when there are no other writers writing or readers reading and that *sched* permits reading only if no writes are occurring; furthermore, any number of reads may be occurring simultaneously.

This sort of argument can be unconvincing; in the next section we shall see how one may use the Nuprl logic in conjunction with the abovementioned type-theoretic account of CCS to state and prove formally properties of CCS expressions.

Reasoning Using Nuprl

Using the Nuprl logic one may reason about the properties of synchronization trees. The logic is rich enough to encompass the predicate calculus, and since the type equality associated with the type, ST, of synchronization trees corresponds to Milner's strong equivalence, one may conduct equational reasoning in the spirit of CCS as well as logical reasoning.

Some properties of concurrent programs have natural expressions as equations; an example of such a property is deadlock, as mentioned above. Other properties, however, do not have obvious statements involving equivalence only; an example of such a property is deadlock-freedom, where we wish to say that no subprocesses of two processes may deadlock, or the readers-writers property in the context of the specification given above. Both of these properties, however, do have natural statments in the Nuprl logic, and the deductive apparatus associated with the logic enables these properties to be proven in a straightforward fashion, if such a proof is possible. Consider deadlock-freedom, for example. To define this property in Nuprl we first define the following terms. Let p be of type ST, $a \in Action$, and $s = a_1 a_2 \ldots a_n$ for n $\geq$ 0 and $a_i \in Action$.

$p \rightarrow^{a} p'$ if and only if $\exists y \in p$.first(y) $= a$ & second(y) $= p'$.

$p \rightarrow^{s} p'$ if and only if $\exists y \in p$.first(y) $= a_1$ & second(y)$\rightarrow^{a_2 a_3 \ldots a_n} p'$.

p' is a *subprocess* of p if and only if $\exists s \in Action^*.p \rightarrow^{s} p'$.

All of these definitions are valid statements in Nuprl. With these in hand, and given the above definition of *Syn*, the following expresses deadlock-freedom with respect to *Syn*.

p and q are *deadlock-free* exactly when

$\forall p',q'{:}ST.\ p'$ a subprocess of p and q' a subprocess of $q \Rightarrow (p'|q') \setminus Syn \neq \{\}$.

To prove this property for specific p and q the logic allows one to introduce p' and q' which are subprocesses of p and q, respectively; we then must prove that the composition of p' and q' and the resulting restriction with respect to *Syn* is nontrivial.

Hennessy-Milner Logic

The Nuprl logic provides a very expressive and powerful logic for reasoning about properties of synchronization. To illustrate this we provide an account of the Hennessy-Milner logic using Nuprl and show that the proof rules for Nuprl are complete for the proof rules given in [Stirling85] for the HM logic.

HM logic is a variant of Propositional Dynamic Logic. The atomic proposition is T (for "true"), and in addition to the usual propositional connectives the logic provides modes subscripted by CCS actions for building propositions. Formally, then, propositions may be described syntactically as follows:

$$P ::= T \mid \neg P \mid P \wedge P \mid <a>P$$

where a is an action as described earlier. Semantically the propositional connectives are interpreted in the usual sense; we may also define derived connectives such as F, $\wedge$ and $\Rightarrow$. Intuitively, a process satisfies $<a>P$ if the process can execute an a action and the resulting subprocess satisfies P. We can define the dual modal operator *[a]* as $[a]P = \neg <a> \neg P$; a process satisfies $[a]P$ if whenever it performs an a action the resulting subprocess satisfies P. To specify the semantics formally, we define a satisfaction relation inductively on processes and predicates.

$p \models T$	for all processes p.
$p \models \neg P$	if it is not the case that $p \models P$.
$p \models P \wedge Q$	if $p \models P$ and $p \models Q$.
$p \models <a>P$	if $\exists p'$:process. $p \to^{a} p'$ and $p' \models P$.

As a first step in translating HM into Nuprl we show how the propositions may be described and how the satisfiability relation may be stated. Recalling that according to the propositions-as-types principle, the truth of a proposition corresponds to whether the corresponding type is inhabited, T in Nuprl can be any type which is inhabited; therefore, the type $0=0$ *in int* serves as an adequate definition of T. Following [Prl Staff86] $\neg P$ may be defined as $P \to void$, where *void* is the empty type. Similarly $P \wedge Q$ can be represented as $P\#Q$, where # is the cartesion product type constructor. We now turn our attention to describing the modal operators of HM in the Nuprl logic. Since $<a>P$ makes an implicit statement about processes, our definition of $<a>P$ in Nuprl will have an unbound variable x of type *process* in it; since we have made the connection between processes and synchronization trees, this unbound variable will in fact have type ST. We argue that the following definition captures the semantic content of $<a>P$.

$\exists y \in x.\ \mathrm{first}(y)=a\ \&\ \mathrm{second}(y)\ \mathrm{in}\ P$

This sentence says that tree x has a branch y in it such that y is labeled by a and the subtree associated with y satisfies P.

In constructive logic satisfiability reduces to provability, and since the Nuprl logic is constructive we shall define the semantics of the above type-theoretic account of HM in terms of $\vdash$ in the Nuprl theory. In general we shall say that $p \models P$ in HM is equivalent to $\vdash P[p/x]$ in Nuprl, where $P[p/x]$ denotes the simultaneous substitution of p for x in P with appropriate renaming to avoid capture. Consider the case of T. In HM any process p satisfies T. Similarly, for any process p, $(0=0\ in\ int)[p/x]\ =\ (0=0\ in\ int)$ is provable, since the type is inhabited, so our semantic interpretation mirrors the one given. Now consider $P \wedge Q$. In the Nuprl framework $\vdash (P \wedge Q)[p/x]$ exactly when $\vdash P[p/x]$ and $\vdash P[p/x]$; again, satisfiability has the desired property. In the case of negation $\neg P[p/x]$, or $P[p/x] \Rightarrow void$, is inhabited (provable) only when $P[p/x]$ is not inhabited (unprovable), so negation has the desired semantic translation. Now consider $<a>P$. Using the Nuprl translation of this form we see that $\vdash <a>P[p/x]$ means that $\vdash \exists y \in x.\ first(y)=a\ \&\ P[second(y)/x]$. This is the case exactly when p may execute an a and wind up in a state satisfying P, so this semantic characterization of $<a>P$ captures the original semantic characterization of the formula.

We shall now shift our attention to the axiomatization of a subset of HM due to Stirling. We shall show that his axioms are all true (in a constructive sense) and that his rules of inference all correspond to restricted rules of inference in the Nuprl system. Stirling's system allows one to reason about HM formulas in the context of SCCS expressions; that is, the synchronous product, $\otimes$, rather than the asynchronous product, $|$, is used as a constructor. The formulas which Stirling allows also cannot involve negation. Formally, then, the formulas which Stirling allows can be described by the following grammar.

$$A ::= T \mid F \mid A \wedge A \mid A \vee A \mid <a>A \mid [a]A \quad \text{where } a \in Action$$

The semantics of these formulas are exactly the semantics given above. The SCCS expressions allowed are as follows.

$$p ::= Z \mid nil \mid ap \mid fix\ Z.p \mid p+p \mid p \otimes p \quad \text{where } a \in Action$$

Fix Z.p is a recursion operator which binds free occurrences of Z in p and allows infinite processes to be defined. We shall consider a finitary version of *fix* in our exposition (although our ongoing work will permit this assumption to be dropped).

The paper defines two notions of provability: $\vdash$ and $\vdash_A$, where A is a HM formula. $\vdash$ has the usual meaning; $p \vdash A$ holds if it can be deduced from the axioms and inference rules. $\vdash_A$, on the other hand, has a slightly different meaning; $p \vdash_A B$ if for every q such that $q \vdash B$, $p|q \vdash B$. It is easy to check that $\vdash_A$ is compositional.

We now consider the axioms and rules of inference in our system. We will show that the translation of Stirling's system is valid in Nuprl. To begin with, we must give an account of $\vdash_A$. Using the set type constructor from Nuprl, $p \vdash_A B$ corresponds to the following.

$$\vdash \forall q{:}\{t{:}ST \mid A(t)\}.\ B(p \otimes q)$$

Namely, B must hold of p composed with any process satisfying A.

The axioms will now be considered. The following table summarizes the axioms of Stirling's system and the Nuprl translations of them. We should note that if $c = b \circ a$ then $b \backslash c = a$.

Axiom	Tranlation
$p \vdash Tr$	$\vdash 0=0$ in int
$nil \vdash [a]A$	$\vdash \forall y:\{\}.\ first(y)=a \Rightarrow A(second(y))$
$p \vdash_A Tr$	$\vdash \forall q:\{t:ST \mid A(t)\}.\ 0=0$ in int
$p \vdash_{False} A$	$\vdash \forall q:\{t:ST \mid void\}.\ A(p \otimes q)$
$nil \vdash_A [a]B$	$\vdash \forall q:\{t:ST \mid A(t)\}.\ \forall p':nil \otimes q.\ first(y)=a \Rightarrow B(second(y))$
$ap \vdash [b]A$ if $b \neq a$	$\vdash \forall a:Action.\ b \neq a \Rightarrow \forall p':ap.\ first(p')=b \Rightarrow A(second(p'))$
$ap \vdash_A [b]B$ if b\a does not exist	$\vdash \forall a:Action.\ \forall q:\{t:ST \mid A(t)\}.\forall p':ap \otimes q.\ first(p')=b \Rightarrow B(second(p'))$, assuming that b\a does not exist

All the axioms, in their Nuprl form, are trivially provable.

We now turn our attention to the rules of inference. Stirling's rules are divided into nine categories; in our presentation we will analyze the rules category by category. We will first translate Stirling's rules into Nuprl and then note that the rules in fact are rules in the Nuprl system or can be implemented as derived rules. We also take the liberty of rewriting Stirling's rules in a refinement (i.e., top-down) style.

Inference Rule	Translation
$p \vdash A \vee B$ $p \vdash A$	$\vdash (A\|B)(p)$ $\vdash A(p)$
$p \vdash A \vee B$ $p \vdash B$	$\vdash (A\|B)(p)$ $\vdash B(p)$
$p \vdash_A B \vee C$ $p \vdash_A B$	$\vdash \forall q{:}\{t{:}ST\|A(t)\}.(B\|C)(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST\|A(t)\}.B(p \otimes q)$
$p \vdash_A B \vee C$ $p \vdash_A C$	$\vdash \forall q{:}\{t{:}ST\|A(t)\}.(B\|C)(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST\|A(t)\}.C(p \otimes q)$
$p \vdash_{A \vee B} C$ $p \vdash_A C$ $p \vdash_B C$	$\vdash \forall q{:}\{t{:}ST\|(A\|B)(q)\}.C(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST\|A(q)\}.C(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST\|B(q)\}.C(p \otimes q)$

$\vee$I

The first two rules correspond to the |-introduction rules in the Nuprl logic. The next three rules have straightforward implementations as *tactics*, or derived rules of inference, in the Nuprl system. We will describe informally the implementation of the last rule. Assuming the proofs of the subgoal, the first step in proving the goal involves a step of ∀-introduction; the existence of an arbitrary q of the appropriate type is assumed, and we know that $(A|B)(q)$ is true. If $A(q)$ is true then then proof of the first subgoal (via a step of ∀-elimination) allows one to conclude that $C(p \otimes q)$ holds; similarly, if $B(q)$ is true then the proof of the second subgoal gives the conclusion.

Inference rule	Translation
$p \vdash A \wedge B$ $p \vdash A$ $p \vdash B$	$\vdash (A \wedge B)(p)$ $\vdash A(p)$ $\vdash B(p)$
$p \vdash_{A \wedge B} C$ $p \vdash_{A} C$	$\vdash \forall q{:}\{t{:}ST \mid (A \wedge B)(t)\}.\ C(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST \mid A(t)\}.\ C(p \otimes q)$
$p \vdash_{A \wedge B} C$ $p \vdash_{B} C$	$\vdash \forall q{:}\{t{:}ST \mid (A \wedge B)(t)\}.\ C(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST \mid B(t)\}.\ C(p \otimes q)$
$p \vdash_{A} B \wedge C$ $p \vdash_{A} B$ $p \vdash_{A} C$	$\vdash \forall q{:}\{t{:}ST \mid A(t)\}.\ (B \wedge C)(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST \mid A(t)\}.\ B(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST \mid A(t)\}.\ C(p \otimes q)$

$\wedge$I

The first rule corresponds to the $\wedge$-introduction rule in the logic. The last three rules involve simple sequences of $\forall$-introduction and $\wedge$-introduction and -elimination rules and may be coded as tactics. To implement the second rule, for instance, assume the proof of the subgoal and perform an $\forall$-introduction on the goal. Using the $q{:}\{t{:}ST \mid (A \wedge B)(t)\}$ assumption resulting from the introduction rule, perform an $\forall$-elimination using the subgoal to conclude that $C(p \otimes q)$ holds.

Inference rule	Translation
$ap \vdash \langle a\rangle A$ $p \vdash A$	$\vdash \exists y{:}ap.\mathrm{first}(y) = a \wedge A(\mathrm{second}(y)$ $\vdash A(p)$
$b.p \vdash_{\langle a\backslash b\rangle A} \langle a\rangle B$ $p \vdash_A B$ assuming a\b exists	$\vdash \forall q{:}\{t{:}ST \mid \langle c\rangle B(t)\}.\exists y{:}bp \otimes q.$ $\mathrm{first}(y) = a \wedge B(\mathrm{second}(y))$ $\vdash a = b \circ c$ $\vdash \forall q{:}\{t{:}ST \mid A(t)\}.B(p \otimes q)$

$\langle a\rangle I$

The first rule is simple ∃-intro in the logic. The second can be implemented with a tactic which performs the following Nuprl reasoning. Given the truth of the subgoals, the tactic first performs an ∀-introduction on the goal. From the definition of ⊗ the tactic concludes that $(bp) \otimes q$, where q is the just-introduced member of $\{t{:}ST \mid \langle c\rangle B(t)\}$, comprises one element, a pair whose first element is a and whose second element is equal to $p \otimes q'$, where q' satisfies A. From the second subgoal, the tactic therefore concludes that the seond element of the pair must satisfy B.

Inference rule	Translation
$ap \vdash [a]A$ $p \vdash A$	$\vdash \forall y{:}ap.\mathrm{first}(y) = a \wedge A(\mathrm{second}(y))$ $\vdash A(p)$
$bp \vdash_{[a\backslash b]A} [a]B$ $p \vdash_A B$ assuming a\b exists	$\vdash \forall q{:}\{t{:}ST \mid ([c]A)(t)\}.\forall y{:}bp \otimes q.$ $\mathrm{first}(y) = a \Rightarrow B(\mathrm{second}(y))$ $\vdash a = b \circ c$ $\vdash \forall q{:}\{t{:}ST \mid A(t)\}.B(p \otimes q)$

[a]I

The first rule has a straightforward implementation, given the definition of *ap*. The second can be implemented as a tactic in the following fashion. Assuming the truth of the subgoals, the tactic first introduces an arbitrary q of type $\{t{:}ST \mid ([c]A)(t)\}$; from the definition of $[c]A$ it knows that each pair in q having c as its first element has a second element satisfying A. From the definition of *bp* and $\otimes$, then, the tactic may conclude that each pair in $(bp) \otimes q$ having a ($= b \circ c$) as its first element must have a second element of the form $p \otimes q'$, where q' satisfies A. The second subgoal therefore allows the conclusion of $B(p \otimes q')$.

Inference rule	Translation
$p+q \vdash <a>A$ $p \vdash <a>A$	$\vdash \exists y{:}p+q.\mathrm{first}(y)=a \wedge A(\mathrm{second}(y))$ $\vdash \exists y{:}p.\mathrm{first}(y)=a \wedge A(\mathrm{second}(y))$
$p+q \vdash <a>A$ $q \vdash <a>A$	$\vdash \exists y{:}p+q.\mathrm{first}(y)=a \wedge A(\mathrm{second}(y))$ $\vdash \exists y{:}q.\mathrm{first}(y)=a \wedge A(\mathrm{second}(y))$
$p+q \vdash_A <a>B$ $p \vdash_A <a>B$	$\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\exists y{:}(p+q)\otimes r.$ $\mathrm{first}(y)=a \wedge B(\mathrm{second}(y))$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\exists y{:}p\otimes r.$ $\mathrm{first}(y)=a \wedge B(\mathrm{second}(y))$
$p+q \vdash_A <a>B$ $q \vdash_A <a>B$	$\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\exists y{:}(p+q)\otimes r.$ $\mathrm{first}(y)=a \wedge B(\mathrm{second}(y))$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\exists y{:}q\otimes r.$ $\mathrm{first}(y)=a \wedge B(\mathrm{second}(y))$

$+I<>$

The first two rules can be implemented with very simple tactics which use the definition of $p+q$ to achieve the desired result. Likewise the third and fourth rules have simple implementations based upon an analysis of $+$ and $\otimes$.

Inference rule	Translation
$p+q \vdash [a]A$ $p \vdash [a]A$ $q \vdash [a]A$	$\vdash \forall y{:}p+q.\mathrm{first}(y)=a \Rightarrow A(\mathrm{second}(y))$ $\vdash \forall y{:}p.\mathrm{first}(y)=a \Rightarrow A(\mathrm{second}(y))$ $\vdash \forall y{:}q.\mathrm{first}(y)=a \Rightarrow A(\mathrm{second}(y))$
$p+q \vdash_A [a]B$ $p \vdash_A [a]B$ $q \vdash_A [a]B$	$\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\forall y{:}(p+q)\otimes r.$ $\mathrm{first}(y)=a \Rightarrow B(\mathrm{second}(y))$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\forall y{:}p\otimes r.$ $\mathrm{first}(y)=a \Rightarrow B(\mathrm{second}(y))$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\forall y{:}q\otimes r.$ $\mathrm{first}(y)=a \Rightarrow B(\mathrm{second}(y))$

+I[]

These rules follow in a straightforward fashion from existing Nuprl rules.

Inference rule	Translation
$p \otimes q \vdash B$ $p \vdash A$ $q \vdash_A B$	$\vdash B(p \otimes q)$ $\vdash A(p)$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\ B(q \otimes r)$
$q \otimes p \vdash B$ $p \vdash A$ $q \vdash_A B$	$\vdash B(q \otimes p)$ $\vdash A(p)$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\ B(p \otimes r)$
$p \otimes q \vdash_A C$ $p \vdash_A B$ $q \vdash_B C$	$\vdash \forall r{:}\{t{:}ST \mid A(t)\}.C(p \otimes q \otimes r)$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\ B(p \otimes r)$ $\vdash \forall r{:}\{t{:}ST \mid B(t)\}.\ C(q \otimes r)$
$q \otimes p \vdash_A C$ $p \vdash_A B$ $q \vdash_B C$	$\vdash \forall r{:}\{t{:}ST \mid A(t)\}.C(q \otimes p \otimes r)$ $\vdash \forall r{:}\{t{:}ST \mid A(t)\}.\ B(p \otimes r)$ $\vdash \forall r{:}\{t{:}ST \mid B(t)\}.\ C(q \otimes r)$

xI

The first and second rules correspond to the ∀-elimination rule in Nuprl, using p (in the first rule) and q (in the second rule) as the object of elimination. The third rule can be implemented as a tactic which first performs ∀-introduction on the goal to obtain r in $\{t{:}ST|A(t)\}$ followed by an ∀-elimination on the first subgoal using r and an ∀-elimination on the second subgoal using $p \otimes r$. The fourth rule is symmetric with the third rule.

Inference rule	Translation
$p \vdash A$ $p^n \vdash A$ where $p = \text{fix } Z.q$ and $n = m(A)$	$\vdash A(p)$ $\vdash A(p)$
$p \vdash_A B$ $p^n \vdash_A B$ where $p = \text{fix } Z.q$ and $n = m(B)$	$\vdash \forall q{:}\{t{:}ST \mid A(t)\}.B(p \otimes q)$ $\vdash \forall q{:}\{t{:}ST \mid A(t)\}.B(p \otimes q)$

fix I

These rules are trivially true in this implementation of SCCS in Nuprl, since all processes are terminating and hence all trees are finite. We are presently working on a model which admits nonterminating processes; this model is based on Mendler's *lazy* types [Mendler85], and we expect that the induction principle associated with these types will correspond to Stirling's rules.

Conclusions

We have presented type-theoretic models of concurrency based upon Milner's CCS and SCCS formalisms. We have also shown that the Nuprl logic appears rich enough to express and reason about properties of concurrent processes, especially properties expressible in the Hennessy-Milner logic.

Several interesting problems arise from this treatment of concurrency. One such problem involves the relative expressiveness of the Nuprl logic. There is evidence that logics besides the Hennessy-Milner logic have natural expressions in the Nuprl logic; investigating this connection would yield some useful results. Describing the properties which the Nuprl logic can express in this framework also appears to be a challenging and interesting problem.

The Nuprl system comes equipped with a metalanguage for developing proof tactics. We are currently engaged in implementing the recursive model decribed in the present paper and are developing proof tactics to facilitate reasoning about properties of synchronization trees. This will alleviate the task of managing the intricate interconnection of detail that often arises in reasoning about concurrent computation.

The program-synthesis aspects of the Nuprl system briefly alluded to in the introduction also deserve further investigation. One can certainly imagine writing a "specification" of a CCS expression as a "there exists" formula; proving the existence of a CCS entity with the desired properties will implicitly provide a means for constructing the desired object. This is a very rudimentary form of verified programming, where a proof specifies a "program" (CCS object in this case) that is guaranteed to satisfy the properties set forth in the theorem. Clearly much work remains to be done in this area.

Acknowledgements

We would like to acknowledge Bob Constable, Doug Howe, Scott Smith, Nax Mendler and Richard Koo for their insightful remarks during the preparation of this paper.

References

[CM85] Constable, R.L., and Mendler, N.P. "Recursive Definitions in Type Theory." *Proceedings of the Logics of Programs Conference 1985*, Springer-Verlag, Heidelberg, 1985.

[DH82] deNicola, R., and Hennessy, M.C.B. "Testing Equivalences for Processes." Technical Report CSR-123-82, University of Edinburgh Department of Computer Science, August 1982.

[GMW 79] Gordon, M.J., Milner, A.R.J., and Wadsworth, C.P. *Edinburgh LCF*, LNCS 78, Springer-Verlag, Heidelberg, 1979.

[HBR84] Hoare, C.A.R., Brookes, S.D., and Roscoe, A.W. "A Theory of Communicating Sequential Processes." *Journal of the ACM*, 31(3), July 1984.

[Hennessy83] Hennessy, Matthew. "A Model for Nondeterminstic Machines." Technical Report CSR-135-83, University of Edinburgh Department of Computer Science, July 1983.

[HM85] Hennessy, Matthew, and Milner, Robin. "Algebraic Laws for Nondeterminism and Concurrency." *Journal of the ACM*, 32(1), January 1985.

[Martin-Lof82] Martin-Löf, Per. "Constructive Mathematics and Computer Programming." *6th International Congress for Ligc, Methodology and Philosophy of Science*, North-Holland, 1982.

[Mendler86] Mendler, Nax, Panangaden, Prakash, and Constable, R.C. "Lazy Objects in Type Theory." In preparation.

[Milner80] Milner, Robin. *A Calculus of Communication Systems*. LNCS 92, Springer-Verlag, Heidelberg, 1980.

[Milner83] Milner, Robin. "Calculi for Synchrony and Asynchrony." *Theoretical Computer Science* 25, 1983

[Prl Staff86] The Prl Staff. *Implementing Mathematics with the Nuprl Proof Development System.* To appear.

[Stirling85] Stirling, Colin. "A Complete Modal Proof System for a Subset of SCCS." *TAPSOFT 85*, LNCS 185, Springer-Verlag, Heidelberg, 1985.