A CCS semantics for NIL

by Scott A. Smolka
Robert E. Strom

We present a syntax-directed translation of NIL, a high-level language for distributed systems programming, into CCS, Milner's Calculus of Communicating Systems. This translation presents unique problems because of NIL's highly dynamic nature, and makes full use of CCS's descriptive facilities. In particular, we consider NIL constructs for dynamic creation and deletion of processes and communication channels, queued synchronous and asynchronous message passing, nondeterministic message selection, and exception handling. A NIL implementation of a simple command shell is used to illustrate the translation procedure. We discuss various issues and open problems concerning the suitability of CCS as an abstract semantics for NIL.

1. Introduction

In this paper, we present a syntax-directed translation of NIL into CCS. (A preliminary version of the paper appeared in the proceedings of the IFIP Conference on the Formal Description of Programming Concepts—III, Ebberup, Denmark, August 1986.) NIL [1, 2] is a high-level language for distributed systems programming developed at IBM Research, Yorktown Heights, New York. CCS, Milner's Calculus of Communicating Systems [3] is a calculus for the description and algebraic manipulation of concurrent communicating systems. Because of the existence of a formal (operational) semantics for CCS, the translation of NIL into CCS effectively gives NIL a formal semantics.

The primary motivation for this work is to provide a formal definition of the semantics of the NIL language, the first such definition. We also believe that the translation will enable people who know CCS to learn NIL and, conversely, enable people who know NIL to learn CCS.

Milner, in Chapter 9 of [3], illustrated the feasibility of using CCS as a semantics for programming languages by presenting a CCS translation of a simple parallel programming language. Other programming languages that have been given CCS semantics include Hoare's CSP [4, 5], a subset of Ada [6], and Sticks and Stones [7]. Furthermore, CCS has been used as a semantics for several programming systems including the ISO OSI [8] and the system-calls level of UNIX [9]. The sum of these efforts demonstrates the utility of CCS as a formal model for concurrent programming languages and systems.

Our translation of NIL into CCS differs from the above work in several respects. Regarding the Ada translation [6], we were required to model NIL's dynamic binding of communication ports to processes; such binding is static in Ada. Also, the particular code body being executed by a NIL process is determined dynamically (see [10] for a much more elaborate comparison of Ada and NIL). Our translation of NIL is inherently different from the UNIX translation [9]. In UNIX, process creation is at the "fork" level; its translation into CCS is thus primarily concerned with the copying and creation of memory segments. Interprocess communication is at the file level (i.e., pipes). The corresponding NIL constructs are at a much higher level of abstraction.

The structure of this paper is as follows: Section 2 provides an overview and comparison of NIL and CCS. In Section 3, we present our translation of NIL into CCS. Some highlights of our translation are also given in the beginning of Section 3. The translation procedure is illustrated in Section 4 using a NIL implementation of a simple command shell. Finally, Section 5 concludes with a discussion of some of the issues in using CCS as a semantics for NIL.

2. An overview of NIL and CCS

In this section we first present an overview of NIL; we follow with an overview of CCS; and conclude with a capsule summary of the main differences between the two.

A CCS semantics for NIL

by Scott A. Smolka
Robert E. Strom

We present a syntax-directed translation of NIL, a high-level language for distributed systems programming, into CCS, Milner's Calculus of Communicating Systems. This translation presents unique problems because of NIL's highly dynamic nature, and makes full use of CCS's descriptive facilities. In particular, we consider NIL constructs for dynamic creation and deletion of processes and communication channels, queued synchronous and asynchronous message passing, nondeterministic message selection, and exception handling. A NIL implementation of a simple command shell is used to illustrate the translation procedure. We discuss various issues and open problems concerning the suitability of CCS as an abstract semantics for NIL.

1. Introduction

In this paper, we present a syntax-directed translation of NIL into CCS. (A preliminary version of the paper appeared in the proceedings of the IFIP Conference on the Formal Description of Programming Concepts—III, Ebberup, Denmark, August 1986.) NIL [1, 2] is a high-level language for distributed systems programming developed at IBM Research, Yorktown Heights, New York. CCS, Milner's Calculus of Communicating Systems [3] is a calculus for the description and algebraic manipulation of concurrent communicating systems. Because of the existence of a formal (operational) semantics for CCS, the translation of NIL into CCS effectively gives NIL a formal semantics.

The primary motivation for this work is to provide a formal definition of the semantics of the NIL language, the first such definition. We also believe that the translation will enable people who know CCS to learn NIL and, conversely, enable people who know NIL to learn CCS.

Milner, in Chapter 9 of [3], illustrated the feasibility of using CCS as a semantics for programming languages by presenting a CCS translation of a simple parallel programming language. Other programming languages that have been given CCS semantics include Hoare's CSP [4, 5], a subset of Ada [6], and Sticks and Stones [7]. Furthermore, CCS has been used as a semantics for several programming systems including the ISO OSI [8] and the system-calls level of UNIX [9]. The sum of these efforts demonstrates the utility of CCS as a formal model for concurrent programming languages and systems.

Our translation of NIL into CCS differs from the above work in several respects. Regarding the Ada translation [6], we were required to model NIL's dynamic binding of communication ports to processes; such binding is static in Ada. Also, the particular code body being executed by a NIL process is determined dynamically (see [10] for a much more elaborate comparison of Ada and NIL). Our translation of NIL is inherently different from the UNIX translation [9]. In UNIX, process creation is at the "fork" level; its translation into CCS is thus primarily concerned with the copying and creation of memory segments. Interprocess communication is at the file level (i.e., pipes). The corresponding NIL constructs are at a much higher level of abstraction.

The structure of this paper is as follows: Section 2 provides an overview and comparison of NIL and CCS. In Section 3, we present our translation of NIL into CCS. Some highlights of our translation are also given in the beginning of Section 3. The translation procedure is illustrated in Section 4 using a NIL implementation of a simple command shell. Finally, Section 5 concludes with a discussion of some of the issues in using CCS as a semantics for NIL.

2. An overview of NIL and CCS

In this section we first present an overview of NIL; we follow with an overview of CCS; and conclude with a capsule summary of the main differences between the two.

• *NIL*

NIL is a high-level systems programming language developed at the IBM T. J. Watson Research Center. The single unit of modularity in NIL is the *process*, which subsumes the notions of procedures, tasks, data abstraction, and others. Unlike abstract data types, where the user of the type is actively making calls and the abstract data type module is passively accepting calls, the relationships between processes in NIL are symmetric.

Each NIL process is a strongly typed, sequential program built from the standard control constructs of Algol-like imperative languages: sequential composition, **if** statements for alternation, and **while** statements for iteration. In addition, NIL supports exception handling. Each statement may result in a *normal* termination, which is followed by the execution of the next sequential statement, or an *exception* termination, in which case execution continues in an *exception handler* associated with the exception condition. The possible exception conditions which can be raised by a given statement are known statically: from the statement name for primitive statements, and from the interface type definition for **call** statements.

Processes in NIL communicate only over *communication channels*; there is no sharing of data across process boundaries. Communication is supported in the language by the type families *input port*—whose values are message queues, and *output port*—whose values are connections to input ports. Such a connection constitutes the right or *capability* to access the input port's message queue for the purpose of enqueueing messages. **Figure 1** depicts a communication channel between two processes. The circles totally within a process represent the local variables. The circles on the boundaries of the processes represent port variables.

Ports, like all variables in NIL, are statically typed. A channel can connect only output ports and input ports of the same user-defined type. A port's type determines the type of the messages that may be transmitted along the channel. Several output ports may be connected to a single input port, but not vice versa. Messages arriving at an input port are enqueued. Processes use guarded commands similar to Ada select statements to selectively respond to communication on their input ports.

NIL supports both *one-way* (asynchronous) and *two-way* (synchronous) communication. Two-way communication, which uses the operations **call**, **accept**, and **return**, involves the transmission of a *callrecord* (collection of actual parameters) and the suspension of the calling process until the accepting process has processed and returned the callrecord. The **accept** operation removes a single callrecord from the input port, or waits until one is available. Sometimes a process which has accepted a callrecord later decides that the call should be serviced by some other process. In this case, the accepting process can also **forward** the callrecord, and the responsibility of returning the callrecord to the caller, to another process. One-way communication, designated by **send** and **receive**, does not cause the sending process to wait. The **receive** operation dequeues a message from the input port, if one is present, and otherwise causes the receiving process to wait. Successive messages sent over a single output port will be received in FIFO order, but no specific order other than a *fair merge* is guaranteed for messages sent over different output ports which arrive at the same input port.*

Figure 1

A communication channel: an input port connected to an output port.

Since NIL is a systems programming language, as opposed to an applications programming language, it is essential that connections between modules change dynamically. The designers of NIL thus chose to allow communication channels to be created dynamically under program control, rather than statically as in languages like CSP [11] and Ada [12]. A process, say P, can create a communication channel through the statement "Y = **outport of** X," where Y is an output port and X is an initialized input port. P can initialize input port X using the statement "**initialize** X." By passing Y to another process Q, P gives Q the capability to send messages to input port X.

Processes are created and destroyed dynamically. An object of type *process* is initialized by issuing a **create** operation, supplying as parameters the name of the file containing the compiled NIL program to be executed by the process, and a list of *creation-time parameters*. These parameters are used to pass initial data and capabilities to an initialization routine within the created process. Like any other NIL object, a process can be passed in a message from one process to another, with the result that the receiving process now owns the passed process.

A process is destroyed when its owner issues a **cancel** operation. Canceling a process which has not already

* *Fairness*, informally stated, means that if a choice between two types of events A and B is offered sufficiently often, eventually each type of event will be chosen. In particular, *fair merge* means that if output port A and output port B are connected to the same input port, and messages are available on both ports, the receiver must not infinitely often choose the B message in preference to the A message or vice versa.

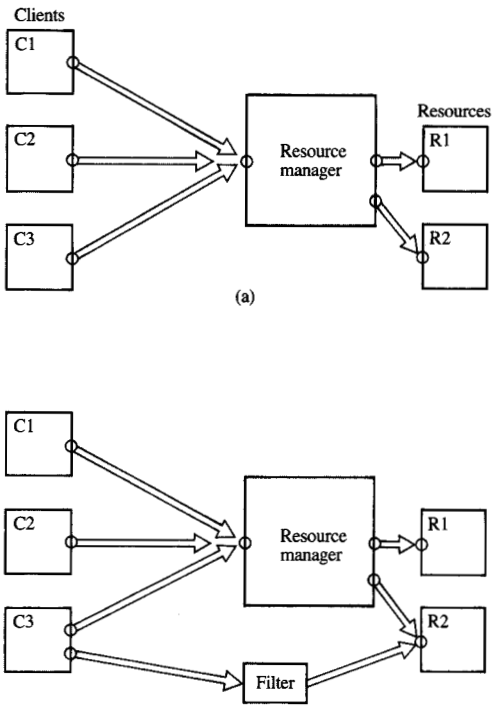

Figure 2

(a) Configuration before the call; (b) configuration after client C3 has called the resource manager and has received a port granting indirect access to resource R2.

terminated causes that process to eventually enter the *canceled state.* A process in the canceled state will have a *CANCEL* exception raised as a result of issuing a subsequent waiting operation (**select**, **receive**, or **call**), or on a subsequent loop iteration. The process is then permitted to perform "last wishes" by providing a handler for the *CANCEL* exception. However, the language rules stipulate that once a CANCEL exception has been raised, the process will terminate in a finite amount of time.

Unlike CCS, NIL has no concept of global time or simultaneity of events in distinct processes. There is only a partial order between the local times of each process as determined by causality of events: The event of a sender sending message *M* precedes the event at which the receiver receives *M*. Events which are not related by communication (either directly or indirectly) are incomparable.

A more complete summary of NIL can be found in [1], and a status report in [13]. The following is a simplified but illustrative example of how NIL is used in systems programming.

Suppose there exist a number of system resources—e.g., printers, files, databases, etc. There are *clients* who from time to time require the use of these resources, and a *resource manager* whose job is to grant clients temporary access to these resources.

Assume that each client, each resource, and the resource manager are separate processes. To simplify the example, assume that the resources each have a single input port for servicing requests to operate on the resource, and that the resource manager has a single input port for servicing requests from clients to obtain access to a resource. Then the configuration of port bindings before any client has requested a resource resembles that in **Figure 2(a)**—each client owns an output port connected to the resource manager's input port, and the resource manager owns output ports connected to the input ports of each resource.

A client requests access to a resource by issuing a **call** statement, passing a callrecord with two components: an **input** parameter *Rq.ResourceClass* specifying the type of resource desired, and an **output** parameter *Rq.Capability* which upon return from the call will contain an output port connected to the resource.

Since the resource manager may wish to grant limited access to the resource, and may wish to retain the right to cancel the access, the resource manager will grant access to the resource not by directly connecting *Rq.Capability* to the resource, but instead by creating an intermediate process called *Filter.* The filter process can be programmed to arbitrarily limit the client's access to the filter. Furthermore, the resource manager can withdraw a client's access to the resource by canceling the filter. The configuration after a successful call from a client to the resource manager resembles **Figure 2(b).**

The NIL code for the resource manager process is shown in **Figure 3.** The resource manager process begins by creating resources R1 and R2. Each **create** operation causes an initialization call to the newly created resource process, which returns an output port connected to the resource's input port—that is, a *capability* to access the resource. After creating the resources, the resource manager iteratively processes service requests.

First the process accepts a request *Rq* from its input port. The definition for the type *GetResourceInterface* specifies what a request must be—namely, an **input** parameter *Rq.ResourceClass* and an **output** parameter *Rq.Capability.* The function *ChooseResource* is then invoked. The code of this function is not shown, but its result will be to select an appropriate resource based upon the requested resource class and to return a capability to that resource. That capability is stored in the variable *ResourceCapability.* An instance of a filter process is created, whose code body is "*Filter.*" The filter is initialized by passing it the capability to the resource as an input parameter, and receiving the capability to the filter as an output parameter. The capability to the filter is

stored in *Rq.Capability*. The filter process is saved in the table *ActiveResources*, and then the request is returned to the calling client.

A realistic version of a resource manager would contain additional code to handle deletion of resources. This code is not shown.

```
/* ..... */
ResourceManager: process
  Resources: TableOfResourcePorts;
  ActiveResources: TableOfFilters;
  ResourceProcess: Process;
  ResourcePort: ResourceInterface;
  ResourceCapability: ResourceInterface;
  FilterProcess: Process;

  InputPort: GetResourceInterface;
  ChooseResource: ChooseResourceInterface;

/* initialize Resources */
  create (ResourceProcess, "R1", ResourcePort);
  insert ResourceProcess into ResourceProcesses;
  insert ResourcePort into Resources;
  create (ResourceProcess, "R2", ResourcePort);
  insert ResourceProcess into ResourceProcesses;
  insert ResourcePort into Resources;

/* service requests */
  while true do
    accept Rq from InputPort;
      ResourceCapability := ChooseResource (Resources,
                                            Rq.ResourceClass);
      create (FilterProcess, "Filter", ResourceCapability,
                                            Rq.Capability);
    insert FilterProcess into ActiveResources;
    return Rq;
  end while;
end ResourceManager;
```

Figure 3

NIL code for resource manager process.

- *CCS*

Milner's CCS (Calculus of Communicating Systems) is a calculus for the description and algebraic manipulation of systems of communicating processes [3]. Various equivalence relations, most notably observational equivalence and congruence, along with their corresponding equational systems, have been proposed for CCS [3, 14]. These systems, which safely allow one CCS term to be rewritten as another, give CCS its manipulative power.

Like NIL, communication in CCS is port-based. The *sort* of a process is the set of ports through which it can communicate with other processes. Unlike NIL, communication in CCS is unbuffered—the sender and receiver of a message must agree to synchronize at some point in time—and the sort of a process is fixed.

The semantics of concurrent composition in CCS is one of interleaved execution of the component processes, with simultaneous moves by two processes whenever they communicate (see Milner's "expansion theorem" [3]).

In order to describe CCS, we first present an example from [3], a variant of which will be used in our translation of NIL. Consider a process that behaves like an infinite queue of elements from some value domain V. We call this behavior *queue*, and it is parameterized by a string s from V^*:

$$queue(s) \Leftarrow in\ (x)\ .\ queue\ (append\ x\ s)\ +$$
$$if\ s = \epsilon\ then\ \overline{out}\ (\$)\ .\ queue(s)$$
$$else\ \overline{out}\ (first\ s)\ .\ queue\ (rest\ s)$$

queue (s) becomes a queue whose contents are x appended to s, whenever a value $x \in V$ is received over input port *in*; it becomes a queue whose contents are *rest s* whenever it outputs its first value over output port $\overline{out}$; and remains the empty queue whenever it outputs \$ (the empty queue symbol) over $\overline{out}$. This example illustrates how the synchronization and communication behavior of a data type can be captured in CCS while leaving its type-theoretic behavior to outside reasoning.

Fairness is not directly expressible in CCS. Thus, one possible behavior for *queue* (ϵ) is to remain the empty queue forever despite repeated attempts by its environment to enqueue some element. Recently, fairness has been incorporated into SCCS in terms of a "finite-delay" operator [15, 16]. SCCS [14] is a variant of CCS in which processes move as if under the control of a global clock. The issue of fairness in CCS is discussed further in Section 5.

We can augment the queue example by defining a transmitter and receiver process which enable end-users to communicate asynchronously with each other using *queue*:

$$transmitter \Leftarrow user\_trans(x)\ .\ \overline{in}(x)\ .\ transmitter$$

$$receiver \Leftarrow out(x)\ .\ \overline{user\_rec}(x)\ .\ receiver$$

The concurrent system consisting of the transmitter, receiver, and queue can be expressed as

$$(transmitter \mid queue\ (\epsilon) \mid receiver) \backslash \{in,\ out\}$$

The \ (backslash) denotes the *restriction* operation and, in this case, limits direct use of the queue to the transmitter and receiver processes. User processes access the set of ports $\{\overline{user\_trans},\ user\_rec\}$ in order to communicate.

We now present the syntax and informal semantics of CCS with *value expressions* over a presupposed value domain D. We use $e_1, e_2, \cdots$ to denote expressions (e.g., function applications and constants) over D, and $x, y, \cdots$ to

Table 1 NIL versus CCS.

NIL	*CCS*
A high-level distributed systems programming language, IBM Yorktown Heights [1, 2]	"A Calculus of Communicating Systems" [3]
Port variables	Port constants only
Dynamic interconnection of communication ports	Static (syntactic) interconnection of input and output ports
Message queues—no "action at a distance"	Synchronous communication
Dynamic creation and cancellation of processes	Composition operator plus recursion
Dynamic binding of code bodies to processes	Static binding of behavior expressions to behavior identifiers
Fairness requirements	Fairness not part of the original calculus

denote variables over D. A complete exposition on CCS including a formal (operational) semantics is given in [3, 14].

Input ports in CCS are denoted by *names* $\alpha, \beta, \cdots$, and output ports are denoted by *conames* $\bar{\alpha}, \bar{\beta}, \cdots$. (In practice, mnemonic English names are often used in place of small Greek letters.) A communication can take place only over *complementary* ports, e.g., α and $\bar{\alpha}$.

Names and conames will also be used, respectively, to denote input *actions* and output *coactions* taken by a CCS process. For example, $\alpha(x)$ is an action that inputs a value for x from port α, and $\bar{\alpha}(e)$ is a coaction that outputs the value e over port $\bar{\alpha}$. In λ-calculus-like terms, the variable x is *bound* by α and the value expression e is *qualified* by $\bar{\alpha}$. Actions $\alpha(x)$ and $\bar{\alpha}(e)$ must occur simultaneously to effect a communication, the result of which is intuitively "$x := e$."

CCS programs, called *behavior expressions* by Milner, can be defined inductively as follows:

- *NIL* (not to be confused with the programming language) is a behavior expression which does absolutely nothing.

Let $\alpha(x)$ be an input action, $\bar{\alpha}(e)$ an output action, and B and C behavior expressions. Then

- $\alpha(x)\,.\,B$ is a behavior expression which first inputs some value v over port α and then behaves like B with all free occurrences of x bound to v.
- $\bar{\alpha}(e).\,B$ is a behavior expression which first outputs the value of e over the port $\bar{\alpha}$ and then behaves exactly like B. Note that every variable in e must be bound for this coaction to make sense.
- $B + C$ is a behavior expression which nondeterministically behaves like B or C. The operator + is the binary version of $\sum$.
- $B \mid C$ is a behavior expression which behaves as the concurrent composition of B and C. The operator $\mid$ is the binary version of $\prod$.
- $B \setminus \{\alpha_1, \cdots, \alpha_n\}$ is a behavior expression which behaves like B with the set of ports $\{\alpha_i, \bar{\alpha}_i \mid 1 \le i \le n\}$ deleted from its sort. $\setminus$ is called the *restriction operator* and effectively hides all α_i-actions and $\bar{\alpha}_i$-coactions from B's outside world.
- $B[\nu_1/\mu_1, \cdots, \nu_n/\mu_n]$ is a behavior expression which behaves like B with all actions/coactions μ_i *relabeled* as ν_i, $1 \le i \le n$.
- *if cond then B else C* is a behavior expression which behaves like B if *cond* is true and like C otherwise.
- $P(e_1, \cdots, e_k)$ is a *behavior identifier* with actual parameters $e_1, \cdots, e_k$. We write $P \Leftarrow B$ to associate behavior identifier P with behavior expression B.

CCS also allows for parametric port names (e.g., α_x) and behavior identifiers (e.g., P_i). In fact, Milner shows in [17] that the entire calculus with value expressions can be encoded into a simpler calculus devoid of value expressions using sets of parametric port names of the form $\{\alpha_d, \bar{\alpha}_d \mid d \in D\}$, where D is the value domain in question.

- *NIL versus CCS*

NIL and CCS were designed with very different purposes in mind—NIL as a high-level systems programming language, and CCS as an algebraic model of concurrency. As such, a direct comparison of the two languages may not be practical. However, both languages define concurrent systems of processes that communicate only through the exchange of messages at ports. **Table 1** summarizes the essential differences between NIL and CCS with respect to interprocess communication and process creation. These differences in turn constitute the main issues addressed by the translation.

3. The translation

In this section we present a syntax-directed translation of NIL into CCS, which will be given in terms of a set of translation rules (one for each NIL construct). The translation rule for a given construct (syntactic unit) S of NIL yields a CCS behavior expression $[[S]]$. The translation is syntax-directed since $[[S]]$ is produced independently of the context of S. For example, [[**if** E **then** S **else** S']] will be derived uniquely from $[[E]]$, $[[S]]$, and $[[S']]$.

Variables in NIL will be modeled as registers in CCS. The restriction operator is applied to prevent other processes from having access to the ports of the local variables of a particular process. However, NIL input and output ports will

be modeled as globally accessible ports in the CCS translation.

• *The sequential component of NIL*

In Chapter 9 of [3], Milner presents a translation of a simple parallel programming language into CCS. This work is fundamental to our own and we thus describe it here.

NIL programs, like programs from Milner's language, will be built from *declarations D* and *statements S*. Declarations in NIL associate variable names with types. Here we use type-free declarations whose syntax is $D \rightarrow \text{var } X; \cdots; \text{var } X$, where X is a *program variable.*

Statements will be built from *expressions E* having syntax $E \rightarrow X \mid F(E, \cdots, E)$, where F is a *function symbol* standing for the built-in function f.

A variable X will be represented by a CCS behavior expression corresponding to a register with sort $\{write_X, \overline{read}_X\}$:

$$LOC_X \Leftarrow write_X(x) \,.\, REG_X(x)$$

$$REG_X(y) \Leftarrow write_X(x) \,.\, REG_X(x) + \overline{read}_X(y) \,.\, REG_X(y)$$

Note that X will be "born" as LOC_X and then become $REG_X(v)$ as soon as it inputs a value v. The set of ports needed to access LOC_X (the *access sort* of LOC_X) is $L_X = \{\overline{write}_X, read_X\}$.

Each $n$ary function symbol F (denoting function f) will be represented by the behavior b_f, which first inputs its n arguments, outputs the value of the corresponding application of f, and then dies:

$$b_f \Leftarrow \rho_1(x_1) \,.\, \cdots \,.\, \rho_n(x_n) \,.\, \overline{\rho}(f(x_1, \cdots, x_n)) \,.\, NIL$$

The translation rules for an expression E containing variables $X_1, \cdots, X_k$ will yield a behavior expression of sort $\{read_{X_1}, \cdots, read_{X_k}, \rho\}$. It uses port $read_{X_i}$ to read the value of X_i, and like function symbols, delivers its result at ρ and then dies.

Several translation rules (e.g., the one for assignment statements) will yield a behavior expression that is dependent on the result delivered by an expression. Thus, for some behavior expression B, Milner abbreviates the CCS expression $([[E]] \mid \rho(x) \,.\, B)\backslash\{\rho\}$ as $[[E]]$ *result* $(\rho(x) \,.\, B)$.

A statement S containing variables $X_1, \cdots, X_k$ will be represented by the behavior expression $[[S]]$ whose sort includes the set $L_{X_1} \cup \cdots \cup L_{X_k} \cup \{\overline{\delta}\}$. The port $\overline{\delta}$ is used by $[[S]]$ to signal its completion and thus effect flow of control. Milner defines the following auxiliary behavior expressions in this light:

$$done = \overline{\delta} \,.\, NIL$$

$$B_1 \; before \; B_2 = (B_1[\beta/\delta] \mid \beta \,.\, B_2)\backslash\{\beta\},$$

where β is not in the sort of B_1 or B_2.

The following translation rules are for expressions [3]:

$$[[X]] = read_X(x) \,.\, \overline{\rho}(x) \,.\, NIL$$

$$[[F(E_1, \cdots, E_n)]] = ([[E_1]][\rho_1/\rho] \mid \cdots \mid [[E_n]][\rho_n/\rho] \mid b_f)\backslash\{\rho_1, \cdots, \rho_n\}$$

In the first rule, the value of X is read from its register and delivered as the result of the expression. In the second rule, the value of each expression is read by the behavior b_f, which then delivers the appropriate function application as its result.

What follows are the translation rules [3] for assignment, sequential composition, conditional, iteration, and begin-block statements. The NIL syntax of these statements can be seen on the left-hand side of the rules:

$$[[X := E]] = [[E]] \; result \; (\rho(x) \,.\, \overline{write}_X(x) \,.\, done)$$

$$[[S; S']] = [[S]] \; before \; [[S']]$$

$$[[\textbf{if } E \textbf{ then } S \textbf{ else } S' \textbf{ end if}]] = [[E]] \; result \; (\rho(x) \,.\, (if \; x \; then \; [[S]] \; else \; [[S']]))$$

$$[[\textbf{while } E \textbf{ do } S \textbf{ end while}]] = w,$$

where w is a new behavior identifier such that

$$w \Leftarrow [[E]] \; result \; (\rho(x) \,.\, if \; x \; then \; ([[S]] \; before \; w) \; else \; done)$$

$$[[\textbf{block declare var } X_1; \cdots; \textbf{var } X_n; \textbf{ begin } S \textbf{ end block}]] = (LOC_{X_1} \mid \cdots \mid LOC_{X_n} \mid [[S]])\backslash L_{X_1} \cup \cdots \cup L_{X_n}$$

In the assignment statement rule, the result of evaluating E is stored in X's register. For sequential composition, the *before* operator ensures that $[[S]]$ is executed before $[[S']]$. In the conditional statement rule, the result of evaluating E is used to determine whether to execute $[[S]]$ or $[[S']]$. For the while statement, E is evaluated to determine whether to re-execute $[[S]]$ or to deliver the *done* signal of $[[S]]$. Finally, for the begin statement, register behaviors for each declared variable are started up in parallel with $[[S]]$. These variables are made local to the scope of the **begin** through restriction.

• *Translating the rest of NIL*

In the previous subsection we considered the translation of expressions, assignment statements, sequential composition, **if-then-else** statements, **while** statements, and **begin-block** statements. Now we consider NIL constructs for dynamic creation and deletion of processes and ports, message passing, nondeterministic message selection, and exception handling. Some additional flow-of-control constructs are also considered.

NIL supports variables of type *inport* (*receiveport*, *acceptport*) and *outport* (*sendport*, *callport*). For example, statements such as $X := Y$ are permitted, where X and Y are outport variables. (After this statement is executed, X will be

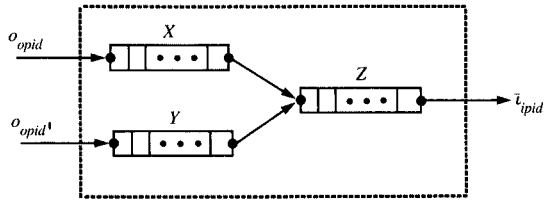

Figure 4

NIL channel.

connected to the same inport as Y.) Also, inports and outports can be passed as messages.

In CCS there is no notion of port variable but only of port constant. The effect of port variables can be obtained in CCS through the use of an indexed set of ports [17]. That is, we associate the CCS port $\bar{o}_{opid}$ with each instance of a NIL outport variable X, where *opid* (short for *outport id*) is a unique index. The outport X may now be referred to in the CCS translation by its index *opid*, which we store in a register associated with X. For example, X may be passed in a message by reading the value of *opid* from its register and then passing this value.

Similarly, we associate a CCS port ι_{ipid}, and a register in which to store the value of *ipid*, with each instance of a NIL inport variable. Note that under this convention, the sort of the CCS behavior expression resulting from the translation of a NIL process will include the set $\{\iota_j, \bar{o}_j \mid j \in \mathbb{N}\}$.

To supply a source of ids in our translation, we define the following behavior expression:

$$IdGen(n) \Leftarrow \overline{gen}(n) \,.\, IdGen(n+1)$$

Through relabeling we obtain a source for inport and outport ids: $IdGen(0)[igen/gen]$ and $IdGen(0)[ogen/gen]$, respectively. Note that both sources start with id 0. We will also need a source of process ids (see the translation of **create**) for which we use the behavior $IdGen(0)[pgen/gen]$.

As described in Section 2, message passing in NIL is completely asynchronous in that messages are queued at the receiving end, and the transit time of a message along a communication channel is indeterminate. To model this NIL asynchrony in CCS, where message passing is unbuffered and synchronous, we "attach" an infinite queue behavior to each of the ι_j, $\bar{o}_j$ CCS ports used in the translation. The effect of a communication channel between outport $\bar{o}_{opid}$ and inport ι_{ipid} can then be obtained by diverting the output of the $\bar{o}_{opid}$ queue to the input of the ι_{ipid} queue. Note that it is necessary to attach an infinite queue to $\bar{o}_{opid}$ (as well as to ι_{ipid}) in order to "desynchronize" transmitting processes.

To illustrate, consider a NIL channel configuration consisting of outports X and Y connected to inport Z. Let $\bar{o}_{opid}$, $\bar{o}_{opid'}$, and ι_{ipid} be the respective CCS ports. Our translation would yield the picture in **Figure 4.**

The access sort of this channel configuration is $\{\bar{o}_{opid}, \bar{o}_{opid'}, \iota_{ipid}\}$ as desired. If an infinite queue were not attached to each of ports o_{opid} and $o_{opid'}$, then the following scenario could occur: Let e be the event of transmitting message M over outport X, and let e' be the event of transmitting message M' over outport Y. If e precedes e' (in the partial order of events in a NIL system—see Section 2), then M would necessarily be enqueued at inport Z before M', violating the aspect of NIL semantics that says the transit time of messages is indeterminate.

What follows are the CCS behavior expressions for outport and inport queues. In each case, we first define a behavior that models an *empty queue*, which evolves into the behavior for a *nonempty queue* upon inputting a value. Of course, a nonempty queue becomes an empty queue after outputting its last value.

Outport queues are provided with a port $\overline{who}_{opid}$ which can be interrogated to determine the id of the inport to which it is connected. This feature is needed in translating outport assignment. Inport queues are provided with two auxiliary ports. Port $\overline{poll}_{ipid}$ can be queried to detect the current state of the queue, a feature needed in the translation of select. Port $make_{ipid}$ can be used to trigger the creation of an outport *opid* connected to inport *ipid.*

Empty outport queue

$$outq\ (opid, ipid) \Leftarrow o_{opid}(x) \,.\, outq'\ (append\ x\ \epsilon, opid, ipid) + \overline{who}_{opid}(ipid) \,.\, outq\ (opid, ipid)$$

Nonempty outport queue

$$outq'\ (s, opid, ipid) \Leftarrow o_{opid}(x) \,.\, outq'\ (append\ x\ s, opid, ipid) + \overline{out}_{ipid}(first\ s) \,.\, if\ (rest\ s = \epsilon)\ then\ outq\ (opid, ipid)\ else\ outq'\ (rest\ s, opid, ipid) + \overline{who}_{opid}(ipid) \,.\, outq'\ (s, opid, ipid)$$

Empty inport queue

$$inq\ (ipid) \Leftarrow out_{ipid}(x) \,.\, inq'\ (append\ x\ \epsilon, ipid) + \overline{poll}_{ipid}(\epsilon) \,.\, inq\ (ipid) + make_{ipid}(opid) \,.\, (inq\ (ipid) \mid outq\ (opid, ipid)) \backslash \{out_{ipid}\}$$

Nonempty inport queue

$$inq'\ (s,\ ipid) \Leftarrow out_{ipid}(x)\ .\ inq'\ (append\ x\ s,\ ipid)$$
$$+\ \bar{\iota}_{ipid}(first\ s)\ .\ if\ (rest\ s = \epsilon)\ then\ inq\ (ipid)$$
$$else\ inq'\ (rest\ s,\ ipid)$$
$$+\ \overline{poll}_{ipid}(s)\ .\ inq'\ (s,\ ipid)$$
$$+\ make_{ipid}(opid)$$
$$.\ (inq'\ (s,\ ipid)\,|\,outq\ (opid,\ ipid))\backslash\{out_{ipid}\}$$

Notice the asymmetry in the definitions: *outq* (and *outq'*) names the inport queue to which it transmits messages, while *inq* (and *inq'*) receives messages anonymously. This parallels the situation in NIL where more than one outport may be connected to a *single* inport, but an outport may be connected to only one inport. Also note that port $\overline{out}_{ipid}$ in $outq'(s,\ opid,\ ipid)$ complements out_{ipid} in $inq'(s,\ ipid)$, thus effecting the connection. These ports are hidden for every inport-outport channel.

What follows are the translation rules for the rest of NIL. The formal syntax of each statement type is evident in the left-hand side of its corresponding translation rule. Comments about the semantics of each statement and its translation are included.

$$[[\textbf{initialize}\ X]] = igen\ (ipid)\ .\ \overline{write}_X(ipid)\ .\ (done\,|\,inq\ (ipid))$$

creates an initialized inport (i.e., capability) X. We note that another possibility for $(done\,|\,inq\ (ipid))$ is $\delta\ .\ inq\ (ipid)$. We view the former as an "optimization" of the latter.

$$[[Y := \textbf{outport of}\ X]] = ogen\ (opid)\ .\ \overline{write}_Y(opid)$$
$$.\ read_X(ipid)\ .\ \overline{make}_{ipid}(opid)\ .\ done$$

creates a unidirectional channel from outport Y to inport X.

$$[[Y := X]] = \qquad (X,\ Y\ \text{both outports})$$
$$read_X(opid\ 1)\ .\ who_{opid\,1}(ipid)\ .\ ogen\ (opid\ 2)$$
$$.\ \overline{write}_Y(opid\ 2)\ .\ \overline{make}_{ipid}(opid\ 2)\ .\ done$$

Y becomes an outport connected to the same inport to which X is connected.

$$[[\textbf{send}(E_1,\ \cdots,\ E_n)\ \textbf{to}\ Y]] = ([[E_1]][\rho_1/\rho]\,|\,\cdots\,|\,[[E_n]][\rho_n/\rho]\,|$$
$$(\rho_1(x_1)\ .\ \cdots\ .\ \rho_n(x_n)\ .\ read_Y(opid)$$
$$.\ \bar{o}_{opid}(x_1,\ \cdots,\ x_n)\ .\ done))\backslash\{\rho_1,\ \cdots,\ \rho_n\}$$

The values of expressions $E_1, \cdots, E_n$ are output to sendport Y. The sender does not wait for a reply.

$$[[\textbf{receive}(X_1,\ \cdots,\ X_n)\ \textbf{from}\ Y]] = read_Y(ipid)$$
$$.\ \iota_{ipid}(x_1,\ \cdots,\ x_n)\ .\ \overline{write}_{X_1}(x_1)\ .\ \cdots\ .\ \overline{write}_{X_n}(x_n)\ .\ done$$

Values are input into variables $X_1, \cdots, X_n$ from receiveport Y. The receiver must wait if Y's message queue is empty.

$$[[\textbf{call}\ Y(X_1, \cdots, X_n)]] = igen\ (ret\_ipid)\ .\ (done\,|\,inq\ (ret\_ipid))$$
$$before$$
$$ogen\ (ret\_opid)\ .\ \overline{make}_{ret\_ipid}(ret\_opid)\ .\ done$$
$$before$$
$$([[X_1]][\rho_1/\rho]\,|\,\cdots\,|\,[[X_n]][\rho_n/\rho]\,|$$
$$\rho_1(x_1)\ .\ \cdots\ .\ \rho_n(x_n)\ .\ read_Y(opid)$$
$$.\ \bar{o}_{opid}(x_1,\ \cdots,\ x_n\ ret\_opid)\ .\ \iota_{ret\_ipid}(x_1',\ \cdots,\ x_n')$$
$$.\ \overline{write}_{X_1}(x_1')\ .\ \cdots\ .\ \overline{write}_{X_n}(x_n')\ .\ done)\backslash\{\rho_1,\ \cdots,\ \rho_n\}$$

Like Ada's **call**, a *callrecord*, i.e., a list of actual parameters $X_1, \cdots, X_n$, is output to callport Y. The caller must wait for the return of the callrecord. NIL also has a facility for returning an exception on a call.

Regarding the translation, a communication channel for the return message is first created. The capability for this channel is passed along with the callrecord parameters to Y. In this translation, all parameters are considered to be **in/out**. The translation of **in** and **out** parameters is handled similarly. For example, since **out** parameters may be uninitialized at the time of call, no attempt is made to read their values.

$$[[\textbf{accept}\ CALLREC\ \textbf{from}\ Y]] = read_Y(ipid)$$
$$.\ \iota_{ipid}(callrec)\ .\ \overline{write}_{CALLREC}(callrec)\ .\ done$$

Like Ada's **accept,** a callrecord is input from acceptport Y into variable *CALLREC*, a record having one field for each formal parameter. The acceptor must wait if Y's queue is empty. After dequeueing a callrecord, the acceptor is responsible for either forwarding or returning the callrecord.

Regarding the translation, the $(n + 1)$th field of *CALLREC* is the capability *ret_opid* to be used by a **return** statement to return *CALLREC* to the caller.

$$[[\textbf{return}\ CALLREC]] = [[CALLREC]]\ result\ (\rho(callrec)$$
$$.\ \bar{o}_{callrec_{n+1}}(all\_but\_last\ callrec)\ .\ done)$$

CALLREC is returned to the caller.

$$[[\textbf{forward}\ CALLREC\ \textbf{to}\ Y]] = read_Y(opid)$$
$$.\ read_{CALLREC}(callrec)\ .\ \bar{o}_{opid}(callrec)\ .\ done$$

CALLREC is forwarded along sendport Y. The process that eventually receives *CALLREC* assumes the responsibility of returning *CALLREC* to the original caller or of reforwarding *CALLREC*.

[[**select**

event (X_1) **guard** (G_1) S_1

.

.

.

event (X_n) **guard** (G_n) S_n

end select]] = $(\prod_{1 \le i \le n} Watchdog_i \mid Controller) \backslash \{gotcha, stop_i, 1 \le i \le n\}$

where

$Watchdog_i \Leftarrow read_{X_i}(ipid)\ .\ ([[G_i]]\ result\ (\rho(g_i))\ .$

$if\ g_i\ then\ W_i\ else\ stop_i\ .\ NIL))$

and

$W_i \Leftarrow stop_i\ .\ NIL + poll_{ipid}(state)\ .$

if $state \neq \epsilon$ *then* $(\overline{gotcha}\ .\ [[S_i]] \mid stop_i\ .\ NIL)$

$+\ stop_i\ .\ NIL$

else Wi

and

$Controller \Leftarrow (gotcha\ .\ \prod_{1 \le i \le n} \overline{stop_i}\ .\ NIL)\ before\ done$

Like Ada's **select,** one of the "open" statements S_i is nondeterministically chosen for execution. No fairness assumptions are made about the selection process. An S_i is open if the queue of its inport X_i is nonempty and its guard (Boolean expression) G_i is true. If none of the S_i are open, the process waits.

In the translation, $Watchdog_i$ (through W_i) continually polls the inport queue of X_i waiting for it to be nonempty. When this is the case and G_i is true (i.e., the ith alternative is open), it tries to get selected by signaling *Controller* with $\overline{gotcha}$. *Controller* will nondeterministically issue a complementary *gotcha* with one of the open alternatives and then kill all of the $Watchdog_i$. The signal $stop_i$ is needed in three different places in W_i to make sure it gets killed. (The killing of the $Watchdog_i$ is for the sake of cleanliness—we view processes as resources—it does not affect the semantics of the translation of **select**.)

[[**create** $(Q, NAME, X_1, \cdots, X_n)$]] =

$igen\ (ipid)\ .\ (done \mid inq\ (ipid))$

before

$ogen\ (opid)\ .\ \overline{make}_{ipid}(opid)\ .\ done$

before

$pgen\ (pid)\ .\ \overline{write}_Q(pid)\ .\ read_{NAME}(name)$

$.\ ([[\textbf{call}\ \text{"}opid\text{"}\ (X_1, \cdots, X_n)]] \mid P_{name}(ipid))$

creates a process Q that executes the compiled NIL program contained in file *NAME*. Creation-time parameters $X_1, \cdots, X_n$ are passed to Q over its *initialization port*. A *process id* (*pid*) is returned as the value of Q, which the owner of Q can use to signal Q's termination (see the translation of the **cancel** operation and the Appendix).

This translation has the effect of starting up an instance of the parameterized behavior P_{name}, which will correspond to the created NIL process Q. P_{name} must have previously been defined through the translation of a process statement labeled *name*. Before that a channel is created for passing the creation-time parameters to Q. The parameter of P_{name} is the inport id for this channel, to which P_{name} will refer when doing an **accept** over its initialization port. Quoted arguments to translations (e.g., *opid* in [[**call** "*opid*" $(X_1, \cdots, X_n)$]]) are to be treated as constants, and thus no access to a register is required. (This is strictly a notational convenience, since we could factor out the accessing of the callport variable register from the translation of **call.** Similar comments apply to other uses of quoted arguments to translations.)

[[**accept** *CALLREC* **from** *Y*]] = (where *Y* is of type **initport**)

[[**accept** *CALLREC* **from** "*init_ipid*"]]

In the translation, *init_ipid* is the first parameter of the behavior currently being defined (see ahead for the translation of the **process** construct).

[[**cancel** *Q*]] = $read_Q(pid)\ .\ \overline{can}_{pid}\ .\ done$

The signal $\overline{can}_{pid}$ will invoke the cancel handler of Q (see the Appendix) if Q has not already terminated.

[[*L*: **block declare var** X_1; $\cdots$; **var** X_m;

begin

S_0;

on $(EX_1)S_1$;

.

.

.

on $(EX_n)S_n$;

end block]] = $((LOC_{X_1} \mid \cdots \mid LOC_{X_m} \mid [[S_0]] \mid$

$EX_1\ .\ \overline{ex}_1\ .\ NIL \mid \cdots \mid EX_n\ .\ \overline{ex}_n\ .\ NIL) \backslash L_{X_1} \cup \cdots \cup L_{X_m}$

$\cup \{EX_1, \cdots, EX_n\}$

$\mid ex_1\ .\ [[S_1]] \mid \cdots \mid ex_n\ .\ [[S_n]] \mid L\ .\ done) \backslash \{ex_1, \cdots, ex_n\}$

$\cup \{L\}$

The begin-block construct is enlarged to include the definition of a block label (L) and exception handlers: Statement S_i is executed whenever exception EX_i is raised

during the execution of S_0 and is not handled by an exception handler of an inner begin block.

In the translation, identifiers EX_i are used as names of "exception ports." By restricting the names of these exception ports, an exception will activate the handler defined in the most closely surrounding block. The ex_i intermediate communications are used to ensure that exceptions raised within one of the handlers S_i do not activate a handler S_j, $1 \le i, j \le n$, but rather a handler from a block surrounding L. When an exception is raised (see below for the translation of **raise**), $[[S_0]]$ terminates without issuing a *done* signal; the *done* signal of the handler becomes the *done* signal of the block.

[[**raise** EX]] = $\overline{EX}$. NIL

raises exception EX.

[[**leave** L]] = $\overline{L}$. NIL

Compound statements (**begin**, **while**, and **select**) may have an optional label. The **leave** statement causes flow of control to be diverted to the point immediately following the compound statement labeled L, and must be nested within this compound statement.

A **leave** statement is translated as an exception with a null handler handled only by block L. (Refer to the behavior L . *done* defined in connection with the translation of a **block** statement labeled L.) **Leave** statements that refer to other types of compound statements are translated similarly.

[[*Ident*: **process** D;

begin

S;

end process]] = $P_{Ident}(init\_ipid)$

Ident is the name of the entry in the executable library which is created upon successful compilation of the NIL program (process) being defined. This name is used by a **create** statement to load the module, at which time statement S is executed. As for a normal begin-block, exception handlers can be provided with the **process** construct. A default handler for the *CANCEL* exception is executed if no such handler is provided explicitly.

The CCS behavior P_{Ident} will be equal to the translation of a begin-block having declarations D and body S. The parameter *init_ipid* will correspond to the id of the initialization port for P_{Ident} (see also the translation of **create** and of **accept** over an initialization port).

To complete the translation, we present the CCS behavior corresponding to a NIL system generation. This behavior will initiate the generators for inport, outport, and process ids, and the translation of a special NIL process "*Init*," with predefined inports and outports. *Init* is supplied with (constant-valued) creation-time parameters.

$IdGen(0)[igen/gen] \mid IdGen(0)[ogen/gen] \mid IdGen(0)[pgen/gen]$

$\mid [[$**create** $(Q,$ "*Init*", $E_1, \cdots, E_n)]]$

In a typical NIL implementation, *Init* would start up the device drivers and then behave as a command shell.

4. An example of the translation

We present a NIL implementation of a simple command shell (interpreter) in order to illustrate our NIL-to-CCS translation. We believe that this example is small enough to be easily presented, yet large enough to highlight several interesting features of NIL and its translation into CCS.

Our shell repeatedly prompts the user for a command, waits until a command is input, and then creates a child process to perform the desired task. Our shell is simplistic in that background processing and I/O redirection are not provided, users cannot kill any processes created by the shell on their behalf, and command names are not checked for their legality.

Our NIL implementation of the shell uses the following variables:

- *StdIO* is a record variable with component fields *StdIO.in*, a receiveport, and *StdIO.out*, a sendport.
- *initp* is the shell's initport.
- *Child* is a process variable and will correspond to the command that the shell is executing.
- *CmdName* is a string variable and is the name of the program that the user would like executed.
- *Parms* is a record variable and corresponds to the user-supplied parameters.

In our NIL implementation of the shell, user commands are processed within a while loop immediately following the shell's initial **accept**. This implementation is somewhat artificial in that there is no return from this **accept**.

What follows is our NIL program for the shell:

```
Shell: process
  declare
    var StdIO; var initp; var Child; var CmdName; var Parms;
  begin
    accept (StdIO) from initp;
    while true do
      send ('?') to StdIO.out;
      receive (CmdName, Parms) from StdIO.in;
      create (Child, CmdName, Parms);
    end while;
end process
```

The CCS behavior expression resulting from our translation of *Shell* is presented in **Figure 5.**

$P_{Shell}(init\_ipid) \Leftarrow$
$(LOC_{StdIO} | LOC_{initp} | LOC_{Child} | LOC_{CmdName} | LOC_{Parms} |$
$((init\_ipid\,(callrec)\,.\,\overline{write}_{StdIO}(callrec)\,.\,done)$
before
w
$)$
$)L_{StdIO} \cup L_{initp} \cup L_{Child} \cup L_{CmdName} \cup L_{Parms}$

where

$w \Leftarrow$
$b_{TRUE}\,result$
$(\rho(x)\,.$
if x then
$(b_{\tau}[\rho_1/\rho] | (\rho_1(x)\,.\,read_{StdIO.out}(opid)\,.\,\overline{o}_{opid}(x)\,.\,done))\backslash\{\rho_1\}$
before
$(read_{StdIO.in}(ipid)\,.\,\iota_{ipid}(x_1, x_2)\,.\,\overline{write}_{CmdName}(x_1)\,.\,\overline{write}_{Parms}(x_2)\,.\,done)$
before
$($ $igen\,(ipid)\,.\,(done | inq\,(ipid))$
before
$ogen\,(opid)\,.\,\overline{make}_{ipid}(opid)\,.\,done$
before
$pgen\,(pid)\,.\,\overline{write}_{Child}(pid)\,.\,read_{CmdName}(fn)\,.$
$(\ (\ igen\,(ret\_ipid)\,.\,(done | inq\,(ret\_ipid))$
before
$ogen\,(ret\_opid)\,.\,\overline{make}_{ret\_ipid}(ret\_opid)\,.\,done$
before
$((read_{Parms}(x)\,.\,\overline{p}(x)\,.\,NIL)[\rho_1/\rho] |$
$(\rho_1(x)\,.\,\overline{o}_{opid}(x, ret\_opid)\,.$
$\iota_{ret\_ipid}(x')\,.\,\overline{write}_{Parms}(x')\,.\,done)$
$)\backslash\{\rho_1\}$
$)$
$| P_{fn}(ipid)$
$)$
$)$
before w
else done
$)$

Figure 5

CCS behavior expression resulting from translation of shell.

5. Discussion

The NIL and CCS models of concurrency share several fundamental principles: (1) Programs define dynamic systems of communicating, nondeterministic, sequential processes. (2) Message passing is the only means for processes to communicate. (3) The interface of a process to the rest of the system is the set of ports that it owns.

This commonality between NIL and CCS is certainly one of the reasons we chose CCS as a semantic model for NIL. Furthermore, the operation of restriction in CCS allowed us to model NIL's static scoping of variables and exception handlers.

NIL and CCS do not agree on the issues of synchronous vs. (buffered) asynchronous communication and dynamic port creation. However, these differences were reconcilable. To model the asynchronous communication of NIL, infinite-queue processes at both the transmitting and receiving ends were used in the CCS translation. NIL dynamic port creation was modeled in CCS using indexed sets of ports, a technique suggested by Milner in [17] and applied in [9].

One of the primary concerns in giving a programming language a formal semantics is the level of *abstractness* of the semantics. In particular, the semantics should be abstract enough to allow all possible implementations. Otherwise, compilers for the language might be constrained to generate code that is less than ideal with respect to a particular target environment.

Our translations of NIL's **select** and **cancel** may not be abstract enough since they begin to suggest particular implementations: the use of watchdog processes for the former and the "canceled state" approach for the latter (see the Appendix). For **cancel**, a truly abstract semantics would be a temporal statement of the form, "a canceled process eventually terminates." As such, *how* termination is obtained is left to the implementors and is not constrained by the semantics.

Our translation of NIL into CCS is from one (programming) language into another. One potential application behind this approach is to be able to use CCS to reason algebraically about NIL programs—e.g., to prove that NIL program transformations, such as those given in [18], are semantics-preserving. Alternatively, we could directly give NIL a more abstract semantics such as archives [19], sets of infinite strings of input/output events. Archives are attractive as a semantics for NIL because they are very abstract and naturally allow one to express fairness.

Proof systems for CCS are based upon various equivalence notions, e.g., observational equivalence and congruence [3, 14]. When reasoning about NIL programs, however, one typically wants to show that a program Q is a correct implementation of a program P, rather than equivalence of P and Q. Specifically, we would like Q to "satisfy" (or refine) P; i.e., every property true of P is true of Q but not necessarily vice versa (see also p. 22 of [15]). Thus the addition to CCS of a proof system for satisfiability, along the lines of the [15] proposal, would address this issue.

In general, the concept of *fairness* is not directly expressible in CCS. For example, consider our CCS description of NIL inport queues, and let P and Q be two NIL processes having outports connected to a single inport X. A possible computation of our translation would allow the messages of P to be enqueued at X infinitely more often than the messages of Q, thereby violating the "fair-merge property" [20]. In certain cases, however, fairness can be enforced in CCS by introducing additional "machinery."

This is evidenced by our translation of **cancel** as described in the Appendix which, with one noted exception, guarantees the eventual termination of the canceled process.

Fairness in CCS has been a subject of intense study in recent years ([15, 16, 21] to name only a few). For our CCS translation of NIL, the work of [21] seems to be the most relevant. They propose to modify the basic operational semantics of CCS by introducing two rules (for both weak and strong fairness) that rule out unfair computations. It would be interesting future work to see if the [21] proof rules are sufficient to satisfy our fairness requirements.

Finally, formal definitions of programming languages can be used to substantiate "folklorish" assertions about the behavior of programs written in the language. In NIL, one such assertion is *security* [1]: the guarantee that processes in a system cannot affect each other except by explicit communication. Security, in turn, can be viewed in terms of three *modularity properties*:

1. Local variables are accessed only by the process in which they are declared.
2. Parameters passed by calls are accessed in a manner consistent with their declarations, i.e., **in**, **out**, or **in/out**.
3. Connections to inports are accessed by a process only after having been received through explicit communication. (This last property is one of "access control.")

By examining the CCS translation of NIL, the modularity properties of NIL can be substantiated:

1. Ports $read_X$ and $write_X$, used to implement the register behavior for local variable X, are restricted in the translation of the begin-block construct, effectively limiting the scope of X to the NIL process in which it is declared.
2. Consider first **out** parameters. These variables may be uninitialized at the point of call, and therefore must not be read at that time. This is consistent with the translation of **call**, where, if X is an **out** parameter, no attempt is made to access port $read_X$ before the call is made. If X is an **in** parameter, the checking that X is never modified by the called process is done statically by the NIL compiler. Parameters of type **in/out** may be accessed by the caller and callee as if they were ordinary variables.
3. Processes in the CCS translation of a NIL program access connections to inports through the set of ports $\{o_{opid} \mid opid \in \mathbb{N}\}$. Thus, the value of a NIL outport variable in the CCS translation is a natural number. The type and typestate checking of NIL programs [13] ensures that this natural number was obtained through the explicit communication of an outport or through local outport assignment.

Acknowledgments

The authors are indebted to Rocky Bernstein, Alessandro Giacalone, Peter Wegner, and especially Shaula Yemini for many helpful discussions. They would also like to thank the referees of the IFIP WG 2.2 Conference on Formal Description of Programming Concepts—III for a careful reading of an earlier version of this paper that produced numerous comments/corrections. This work was initiated in the summer of 1984 when the first author was visiting IBM, Yorktown Heights, New York, as a faculty researcher. He would like to thank IBM for its support and also acknowledge the support he received under NSF Grant DCR-8505873.

Appendix: Semantics of cancellation

• *Informal semantics*

Every NIL process is the value of a variable of type *process* within its owner. The **create** *P* statement creates a new process and assigns it as the value of a process variable *P*; the **cancel** *P* statement causes the process designated by *P* to enter the *canceled state.* A process in the canceled state *may* execute some finite number of actions ("last wishes") before terminating, but is guaranteed to eventually terminate.

Processes may block indefinitely as a result of **while**, **select**, **receive**, **accept**, or **call** statements. It would be inconsistent for the semantics both to require that canceled processes always terminate and to have some statements not be live. Therefore, the above five statements are defined so that they may terminate by raising the *CANCEL* exception, as well as by normal termination. Normal termination will occur as defined in the main body of the paper. The exception termination must occur whenever normal termination is impossible and failure to terminate would result in the failure of a canceled process to terminate.

• *Derivation of the CCS specification*

The CCS specification for termination of **while**, **select**, **receive**, or **accept** for canceled processes is relatively straightforward. If the process attempting to execute **while**, **select**, or **receive** is in the canceled state, then the statement may raise the *CANCEL* exception, or it may continue normal execution. For **receive/accept** and **select**, a blocked process will be waiting at ι_{ipid} or *gotcha*, respectively; by waiting alternatively for a cancel "beacon" signal, these statements will be guaranteed to eventually become unblocked if the cancel signal is issued. For **while**, it is necessary to execute the loop in parallel with a behavior which waits for either the termination of the **while** or the cancel beacon. If the cancel beacon is sensed, then if the loop still does not terminate, a *stop* message is sent, forcing the loop to terminate.

The CCS specification for termination of **call** is more intricate as a result of the requirement that **call** must not

terminate, even with an exception, without the callrecord being returned. Now it is necessary to require that the called process raise an exception if it does not make progress while holding a callrecord belonging to a canceled calling process.

A process under obligation to return its callrecord in order to allow some calling process to proceed toward cancellation is said to be *forced.* If a forced process is itself blocked because of a **call** statement, it may in turn cause the process it is calling to become forced. A forced process ceases to be forced after it has returned any callrecords it has accepted from canceled or forced processes. So long as a process is in the canceled state or is forced, attempts to execute otherwise nonterminating statements will terminate with an exception. Since in NIL, raising an exception always terminates the current block, eventually all blocks will terminate. The NIL *typestate rules* [13] guarantee that processes finalize all their variables before termination, and in particular that any accepted callrecords will be returned.

The CCS solution for **call** is as follows:

- Every NIL process runs in parallel with a monitor process which tracks the canceled and forced states. When the monitor process receives a *can* signal, it enters the *Canceled* state. If the NIL process becomes forced because it owns a callmessage originating from another forced or canceled process, then the monitor will receive a *force* signal and enter the *Forced* state. A count is kept which is incremented each time the *force* signal is received, and decremented each time a callmessage from a canceled or forced process is returned. When canceled or forced, the *beacon* signal is repeatedly offered. The *beacon* signal causes the blocked **while**, **select**, **receive**, or **accept** statement to terminate.
- A monitor is provided which records whether the associated NIL process is engaged in a call: Either (a) no call is in progress; (b) a call is in progress to process *pid* but the caller has not yet tried to force the call to return; or (c) a call is in progress and another process has been put in the forced state because of the call.
- The call behavior is modified so that if the call does not return but the cancelbeacon is sensed, then a new state is entered which attempts to either receive the returned callrecord or force the current owner of the callmessage. The callrecord is augmented so that it includes the *pid* (process id) of the calling process.
- The return behavior is modified so that if the returning process has had its forced count incremented as a result of this callrecord, the forced count will be decremented.
- The forward behavior is modified so that if the forwarding process has had its forced count incremented as a result of this callrecord, the forward will be treated as a return, and if not, the calling process will be aware of the identity of the new owner of the callrecord.
- Output ports are modified so that the identity of the owner of the associated input port can be determined. (This modification is not shown here.)

Formal definitions

- $Who\,(pid) \Leftarrow \overline{whoami}\,(pid)\,.\,Who\,(pid)$
- $Normal\,(pid) \Leftarrow can_{pid}\,.\,Canceled\,(pid)$
  $+\,force_{pid}\,.\,Forced\,(1, pid)$
- $Canceled\,(pid) \Leftarrow force_{pid}\,.\,Canceled\,(pid)$
  $+\,unforce_{pid}\,.\,Canceled\,(pid)$
  $+\,\overline{beacon}\,.\,Canceled\,(pid)$
- $Forced\,(i, pid) \Leftarrow force_{pid}\,.\,Forced\,(i+1, pid)$
  $+\,unforce_{pid}$
  $.\,(\textit{if } i-1=0 \textit{ then } Normal \textit{ else } Forced\,(i-1, pid))$
  $+\,\overline{beacon}\,.\,Forced\,(i, pid) + can_{pid}\,.\,Canceled\,(pid)$
- $NotCalling\,(pid) \Leftarrow call\,(pid\ 2)\,.\,Calling\,(pid, pid\ 2)$
- $Calling\,(pid, pid\ 2) \Leftarrow returned\,.\,NotCalling\,(pid)$
  $+\,return_{pid}\,.\,Calling\,(pid, pid\ 2)$
  $+\,forward_{pid}(pid\ 3, opid, callrec)$
  $.\,\overline{o}_{opid}(callrec)\,.\,Calling\,(pid, pid\ 3)$
  $+\,forcecallee\,.\,\overline{force}_{pid2}\,.\,Forcing\,(pid, pid\ 2)$
- $Forcing\,(pid, pid\ 2) \Leftarrow returned\,.\,NotCalling\,(pid)$
  $+\,return_{pid}\,.\,\overline{unforce}_{pid2}\,.\,Forcing\,(pid, pid\ 2)$
  $+\,forward_{pid}(pid\ 3, opid, callrec)$
  $.\,\overline{o}_{callrec_{n+1}}(all\_but\_last\_two\ callrec)$
  $.\,Forcing\,(pid, pid\ 2)$

Translation of **create** The last line of the translation of **create** given in Section 4.2 now reads:

$([[\textbf{call}\ \text{“}opid\text{”}\ (X_1, \cdots, X_n)]]\,|$

$(P_{name}(ipid)\,|\,Who\,(pid)\,|\,Normal\,(pid)\,|\,NotCalling\,(pid))$

$\backslash\{whoami, call, returned\})$

Translation of ***while*** We define two new behaviors that will run in parallel with the translation of **while**. *Cancelmonitor* waits for either the beacon or the loop termination; if the beacon is sensed, a *CANCEL* exception will be raised on a subsequent loop boundary:

$Whilemonitor \Leftarrow check\,.\,(Whilemonitor$

$+\,stop\,.\,[[\textbf{raise}\ CANCEL]])$

$Cancelmonitor \Leftarrow \delta\,.\,done + beacon\,.\,(\delta\,.\,done + \overline{stop}\,.\,NIL)$

[[**while** E **do** S **end while**]] = $((w \mid Whilemonitor)\backslash\{check\}$

$\mid Cancelmonitor)\backslash\{\delta, stop\}$

where w is as in Section 3 but begins with a $\overline{check}$ action each time around. Note: If no fairness assumptions about CCS are made, there is no guarantee that the *stop* will eventually be received even though continuously enabled, because *check* may be infinitely often chosen.

Translation of ***receive*** Nondeterministically or the behavior *beacon* . [[**raise** *CANCEL*]] to the translation of **receive** given in Section 3.

Translation of ***accept*** Nondeterministically or the behavior *beacon* . [[**raise** *CANCEL*]] to the translation of **accept** given in Section 3.

Translation of ***call*** The behavior immediately following the second *before* of the translation of **call** given in Section 3 now reads

$$
\begin{aligned}
&([[X_1]][\rho_1/\rho] \mid \cdots \mid [[X_n]][\rho_n/\rho] \mid \rho_1(x_1) \,.\, \cdots \,.\, \rho_n(x_n) \\
&\quad .\ read_Y(opid) \,.\, owner_{opid}(pid\ 2) \,.\, \overline{call}\ (pid\ 2) \\
&\quad .\ whoami\ (pid\ 3) \,.\, \overline{o}_{opid}(x_1, \cdots, x_n, ret\_opid, pid\ 3) \\
&\quad .\ (\iota_{ret\_ipid}(x_1', \cdots, x_n') \,.\, \overline{write}_{X_1}(x_1') \,.\, \cdots \\
&\quad .\ \overline{write}_{X_n}(x_n') \,.\, returned \,.\, done \\
&\quad + beacon \,.\, (\iota_{ret\_ipid}(x_1', \cdots, x_n') \,.\, \overline{write}_{X_1}(x_1') \,.\, \cdots \\
&\quad .\ \overline{write}_{X_n}(x_n') \,.\, returned \,.\, done) \\
&\quad + \overline{forcecallee} \,.\, (\iota_{ret\_ipid}(x_1', \cdots, x_n') \,.\, \overline{write}_{X_1}(x_1') \,.\, \cdots \\
&\quad .\ \overline{write}_{X_n}(x_n') \,.\, returned \\
&\quad .\ (done + [[\textbf{raise}\ CANCEL]])))\backslash\{\rho_1, \cdots, \rho_n\}
\end{aligned}
$$

Translation of ***return*** Same as the translation given in Section 3 for **return** except insert the action $\overline{return}_{callrec_{n+2}}$ immediately before the $\overline{o}_{callrec_{n+1}}$ action.

Translation of ***forward*** The translation of **forward** given in Section 3 now reads

$$
\begin{aligned}
&read_Y(opid) \,.\, read_{CALLREC}(callrec) \,.\, owner_{opid}(pid) \\
&\quad .\ \overline{forward}_{callrec_{n+2}}(pid, opid, callrec) \,.\, done
\end{aligned}
$$

References

1. R. E. Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, San Francisco, June 1983, pp. 73–82.
2. Robert Strom and Nagui Halim, "A New Programming Methodology for Long-Lived Software Systems," *IBM J. Res. Develop.* **28,** No. 1, 52–59 (January 1984).
3. R. Milner, *A Calculus of Communicating Systems, Lecture Notes in Computer Science* **92,** Springer-Verlag New York, 1980.
4. M. Hennessy, W. Li, and G. Plotkin, "A First Attempt at Translating CSP into CCS," *Proceedings of the 2nd IEEE International Conference on Distributed Computing*, 1981.
5. E. Astesiano and E. Zucca, "Semantics of CSP Via Translation into CCS," *Proceedings of the 10th Symposium on Mathematical Foundations of Computing, Lecture Notes in Computer Science* **118,** 172–182 (1981).
6. M. Hennessy and W. Li, "Translating a Subset of Ada into CCS," *Proceedings of the IFIP Conference on Formal Description of Programming Concepts—II*, North-Holland Publishing Co., Amsterdam, 1983, pp. 227–247.
7. L. Cardelli, "Sticks and Stones: An Applicative VLSI Design Language," *Report No. CSR-85-81*, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1981.
8. M. W. Sheilds and M. J. Wray, "A CCS Specification of the OSI Network Service," *Report No. CSR-136-83*, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, August 1983.
9. T. W. Doeppner, Jr., and A. Giacalone, "A Formal Description of the UNIX Operating System," *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, Montreal, August 1983, pp. 241–253.
10. R. E. Strom, S. Yemini, and P. Wegner, "Viewing Ada from a Process Model Perspective," *Proceedings of the AdaTec Symposium on the Ada Programming Language*, Paris, 1985; available as *ACM SIGADA Proceedings*.
11. C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM* **21,** 666–677 (August 1978).
12. "Reference Manual for the Ada Programming Language" *MIL-STD 1815A*, U.S. Department of Defense, Washington, DC, February 1983.
13. R. E. Strom and S. Yemini, "The NIL Distributed Systems Programming Language: A Status Report," *Proceedings of the Seminar on Concurrency, Lecture Notes in Computer Science* **197,** 512–523 (1985).
14. R. Milner, "Calculi for Synchrony and Asynchrony," *J. Theor. Comput. Sci.* **25,** 267–310 (1983).
15. R. Milner, "A Finite-Delay Operator in Synchronous CCS," *Report No. CSR-116-82*, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, May 1982.
16. M. Hennessy, "Axiomatising Finite Delay Operators," *Acta Informat.*, No. 21, 61–88 (1984).
17. R. Milner, "Lectures on a Calculus for Communicating Systems" *Proceedings of the Seminar on Concurrency, Lecture Notes in Computer Science* **197,** 197–220 (1985).
18. R. E. Strom and S. Yemini, "Synthesizing Distributed and Parallel Programs Through Optimistic Transformations," *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp. 632–642.
19. R. M. Keller and P. Panangaden, "Semantics of Networks Containing Indeterminate Operators," *Proceedings of the Seminar on Concurrency, Lecture Notes in Computer Science* **197,** 479–496 (1985).
20. D. Park, "On the Semantics of Fair Parallelism," *Lecture Notes in Computer Science* **86,** 504–526 (1980).
21. G. Costa and C. Stirling, "Weak and Strong Fairness in CCS," *Report No. CSR-167-85*, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, January 1985.

Received March 11, 1987; accepted for publication May 22, 1987

Scott A. Smolka *State University of New York, Department of Computer Science, Stony Brook, New York 11794.* Professor Smolka received his B.A. and M.A. in mathematics from Boston University, Massachusetts, in 1975 and 1977. In 1984, he received his Ph.D. in computer science from Brown University, Providence, Rhode Island.

Since 1983, he has been an assistant professor at SUNY Stony Brook. His current research interests include the formal analysis of communicating processes and related computational complexity issues, design environments for concurrent systems, and distributed algorithms.

Robert E. Strom *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Strom completed his B.A. at Harvard University, Cambridge, Massachusetts, in 1966, and his doctoral studies at Washington University, St. Louis, Missouri, in 1972. He joined IBM in 1977 as a Research staff member at the Thomas J. Watson Research Center. His current interests are programming language design and implementation, program transformations, distributed systems, and semantics.