Toward an Operational Semantics of PROMELA in ACL2

William R. Bevier

Computational Logic, Inc.
1717 West 6th Street, Suite 290
Austin, Texas 78703-4776
bevier@cli.com

1 Introduction

PROMELA is a language for modeling concurrent systems, developed specifically for describing communication protocols. Analysis of some properties of PROMELA models is automated by the SPIN model checker [Hol91].

Currently, the semantics of PROMELA is most completely and accurately defined by the C implementation of SPIN. It would be preferable to have a formal definition of PROMELA semantics independent of this implementation. This report describes an operational semantic definition for most of PROMELA in the logic of ACL2 [KM94]. It should be considered a preliminary version, which can be refined in response to public scrutiny.

Natarajan and Holzmann have described an operational semantics for a smaller subset of PROMELA. Their semantics is based on labeled transition systems [NH96]. The semantic definition presented here was created independently and does not reflect the structure of their definition, although there are many similarities.

This definition complements their work and offers several advantages. First, ACL2 performs some useful error checking on the definition. The number of arguments and results of each function is checked for consistent usage. The termination of each function is proved at the time it is accepted. The primitive ACL2 type of the result of each function is automatically deduced.

It is possible to use ACL2 to check properties of the definition. The semantic definition is complex, and it is helpful to verify some of its expected properties to avoid errors. Each function can be adorned with *guards*, which describe pre-conditions on function arguments. Guard proofs can be checked in ACL2. A useful thing to do, which we have only begun, is to define the predicate that characterizes a legal PROMELA state and show that it is an invariant of the semantic function.

ACL2 includes an automatic re-play facility. Once a semantic definition is completed, modifications to it can be re-played. Only events which fail to get accepted by the theorem prover need attention from the user. The re-play facility can help to make sure that important invariants of the model are preserved after modifications have been made. In general, proposed changes to the language can be analyzed with proof-checking support.

A second benefit of this style of model is that it is executable. We can simulate PROMELA models directly in this definition. The ability to "debug" the model via execution is very helpful. We have *chosen* to make the model executable; ACL2 does not limit one to building executable models. Because of this choice, a number of concepts are not modeled as generally as one would like.

Finally, a model such as this opens up possibilities in the combined use of theorem proving and model checking for analysis. Approaches to proof-based verification for notations like TLA [Lam91] and Unity [CM88] exist. The techniques developed for these notations can be adapted for *proving* properties of PROMELA programs. This can be another way around the state explosion problem.

Can this definition (or some modification of it) serve as a reference model for PROMELA semantics? Can the the definition fulfill some purpose in addition to documentation? In the near future we hope to make the definition publicly available for experimentation to help answer these questions.

This paper should be considered a report on work in progress, not a polished result. The purpose of this paper is to announce the presence of this model, and to see if there is interest in its use. Any errors are due to the author's misunderstanding of PROMELA semantics.

Section 2 presents the language that we handle, and gives an example program. Section 3 gives a sketch of the semantic model for this language. In the future, a full technical report will contain the details. Section 4 offers a comparison of the structures of the PROMELA ACL2 model and the semantic definition in [NH96]. Section5 discusses the state of the formal semantics as a software system. Section 6 provides a brief introduction to ACL2, and can be used as a reference for various constructs presented in the paper.

2 The Language

For the sake of simplicity, Natarajan and Holzmann have excluded the following issues from their semantic definition.

1. array variables
2. assert statements
3. never claims
4. correctness properties expressed by progress and accept labels
5. multiple arguments on run statements
6. multiple fields in a message
7. compound statements, including control flow statements
8. other "uninteresting" statements like `printf` and `skip`

Our definition handles all of the above except 3 and 4. We also model the dynamic creation of variables and channels in a style different from [NH96]. We do not model deterministic steps or process priorities.

This definition includes one major simplification, which is merely an expediency. All primitive numeric data types are unbounded integers. We have not yet extended ACL2 with a finite arithmetic library, so we rely on the unbounded arithmetic built in to ACL2. Fixing this is just a matter of time and energy. Section 2.1 gives the grammar of the language we handle. Section 2.2 shows an example program.

2.1 Grammar

We present a grammar for the forms directly read in by the PROMELA semantic function, which adopts the syntactic conventions of Common Lisp. Using lex and yacc we can easily build a front end that will let us parse the C-syntax of PROMELA and generate programs in this form. Lower case symbols in this grammar are non-terminals. Upper case symbols are terminals. Parentheses are not meta-symbols; they appear where they are syntactically required.

Toplevel Forms

```
start      ::=   unit+
unit       ::=   define | proctype | init | one-decl | mtype
define     ::=   (DEFINE name expr)
proctype   ::=   (PROCTYPE name decl-list body))
init       ::=   (INIT body))
mtype      ::=   (MTYPE name+)
```

Declarations

```
decl-list   ::=   one-decl*
one-decl    ::=   (typesymbol ivar+)
ivar        ::=   var-decl | (ASSIGN var-decl initializer)
initializer ::=   expr | ch-init
ch-init     ::=   (OF cap-spec (typesymbol+))
cap-spec    ::=   natural | symbol
var-decl    ::=   name | (name natural)
```

Statements

```
body        ::=   stmt
stmt        ::=   structured-stmt | elementary-stmt | (label @ stmt)
label       ::=   (symbol+)

structured-stmt  ::= atomic | selection | repetition | sequence

atomic      ::=  (ATOMIC stmt*)
selection   ::=  (IF option+)
repetition  ::=  (DO option+)
sequence    ::=  (SEQ stmt*)
option      ::=  (OPTION stmt*)
```

```
elementary-stmt ::= assertion | assignment | break | end | expr |
                    goto | one-decl | printf | receive | send | skip

assertion    ::=  (ASSERT expr)
assignment   ::=  (ASSIGN var-ref expr)
break        ::=  (BREAK)
end          ::=  (END)
goto         ::=  (GOTO label)
printf       ::=  (PRINTF string expr*)
receive      ::=  (RECEIVE var-ref convar*)
send         ::=  (SEND var-ref expr*)
skip         ::=  (SKIP)
```

Types and Expressions

```
typesymbol   ::=  BIT | BOOL | BYTE | SHORT | INT | CHAN
convar       ::=  symbol
name         ::=  symbol

expr            ::=  constant | var-ref | opr-application
constant        ::=  integer
var-ref         ::=  name | (NTH expr name)
opr-application ::=  (== expr expr)    |  (+ expr expr)   |
                     (- expr expr)     |  (* expr expr)   |
                     (/ expr expr)     |  (% expr expr)   |
                     (NEG expr)        |  (AND expr expr) |
                     (OR expr expr)    |  (NOT expr)      |
                     (IMP expr expr)   |  (IFF expr expr) |
                     (XOR expr expr)   |  (< expr expr)   |
                     (<= expr expr)    |  (> expr expr)   |
                     (>= expr expr)    |  (LEN expr)      |
                     (? expr (* expr)) |  (TIMEOUT)       |
                     (RUN (name (* expr))) |
                     (NTH expr expr)
```

2.2 An Example Program

Here is the PROMELA factorial program as presented in [Hol91].

```
proctype fact(int n; chan p)
{     int result;
      if
      :: (n <= 1) -> p!1;
      :: (n >= 2) ->
              chan child = [1] of { int};
              run fact(n-1, child);
              child?result;
              p!n*result
      fi
}
```

```
init
{     int result;
      chan child = [1] of {int };

      run fact(5, child);
      child?result;
      printf("result: %d\n", result)
}
```

The internal representation of this program accepted by our grammar, and processed by our interpreter appears as follows.

```
(proctype fact ((int n) (chan p))
          (seq (int result)
               (if (option (<= n 1 ) (send p 1))
                   (option (>= n 2 )
                           (chan (assign child (of 1 ( int ))))
                           (run (fact (- n 1) child))
                           (receive child result)
                           (send p (* n result))))
               (end)))

(init (seq (int result)
           (chan (assign child (of 1 ( int ))))
           (run (fact 5 child))
           (receive child result)
           (printf "~%result: ~d" result)
           (end)))
```

Here is a formatted display of the initial state of factorial program as created by the ACL2 PROMELA interpreter. The details of system state are introduced in Section 3.2. However, we can see that initially no system error is recorded, no process is executing atomically, and there are no global variables or channels. There is a single process, the initial process. It is active, and its program counter points to the toplevel of the `init` program body. At the start of the computation, there are no local variables in the initial process.

```
Error:     NIL
Atomic:    NIL
Globals:
Channels:

Process 0 (INIT): Active: T; PC: NIL;
   Locals:
```

After three steps, the first three statements in the toplevel sequential composition form in `init` have been executed. First, the local variable `result` is allocated in `init` with default initial value of `0`. Second, the channel `child` is allocated. An empty channel with capacity 1 is created in channel space, and a local variable `child` is allocated in `init` whose value is the address of the new channel. Third, a `fact` process is created. Parameters are allocated as local variables. Note that in this display, we do not show the type of variables, or the type and capacity of channels.

```
Error:     NIL
Atomic:    NIL
Globals:
Channels:  0 = NIL;

Process 0 (INIT): Active: T; PC: (4)
  Locals: CHILD = 0; RESULT = 0

Process 1 (FACT): Active: T; PC: NIL
  Locals: N = 5; P = 0
```

Here is the final state of a computation of 5!. It shows that five channels and five `fact` processes have been created. At the end of the process, all channels are empty. The local variables assigned to each process are displayed. The program counter of each process points to an `end` statement.

```
Error:     NIL
Atomic:    NIL
Globals:
Channels:  4 = NIL; 3 = NIL; 2 = NIL; 1 = NIL; 0 = NIL;

Process 0 (INIT): Active: NIL; PC: (6);
  Locals: CHILD = 0; RESULT = 120

Process 1 (FACT): Active: NIL; PC: (3);
  Locals: CHILD = 1; RESULT = 24; N = 5; P = 0

Process 2 (FACT): Active: NIL; PC: (3);
  Locals: CHILD = 2; RESULT = 6; N = 4; P = 1

Process 3 (FACT): Active: NIL; PC: (3);
  Locals: CHILD = 3; RESULT = 2; N = 3; P = 2

Process 4 (FACT): Active: NIL; PC: (3);
  Locals: CHILD = 4; RESULT = 1; N = 2; P = 3

Process 5 (FACT): Active: NIL; PC: (3);
  Locals: RESULT = 0; N = 1; P = 4
```

3 Sketch of the Semantic Definition

The state of a PROMELA computation is described by a complex structure discussed in the following sections. Among the components of a PROMELA state are a list of global constants, a list of global variables, a list of channels, and a list of processes. There are other control fields in the state structure, which are described later.

The semantics of PROMELA is captured in a function called `step-fn`, which defines the effect on a state by a primitive operation. The arguments to `step-fn` are a system state and a current process identifier. The value returned is an updated system state.

We present the definition of `step-fn` in the order that ACL2 processes it. This means we begin with primitive concepts and move toward higher levels. `step-fn` is the last definition. For the sake of brevity, many definitions are omitted.

Section 3.1 discusses preliminary definitions. Section 3.2 presents the structure of state. Section 3.3 describes expression evaluation. Statement executability is discussed in Section 3.4. A sketch of the semantics of some statements is given in Section 3.5. Section 3.6 gives the toplevel definition of `step-fn`.

3.1 Foundation

System Constants

The following system constants are defined. We give them specific values to make the step function executable. These can be changed to any natural number.

```
(defun maxprocesses () 100)

(defun maxchannels () 100)

(defun maxchannelcapacity () 10)
```

Types

A primitive type is one of `bit`, `bool`, `byte`, `short`, `int`, or `chan`. A value of type `chan` is a pointer to a message queue. The boolean type is synonymous with the bit type. We use symbolic representations for these types. For example, a bit type is defined as follows.

```
(defun bit-t () 'bit)
```

The predicate `type-p` recognizes a legal type. `type-list-p` recognizes a list of types, and is introduced by the `deflist` mechanism described in Section 6.

```
(defun type-p (x) (member x (list (bit-t) (byte-t) (short-t) (int-t) (chan-t))))

(deflist type-list-p (l) type-p)
```

The function `in-type` recognizes a value in the domain of a given type. `default-val` creates a default value for a given type. `cast-val` casts a value to a given type. We omit these definitions.

Channels

A message is a tuple, represented as a list of values. A channel is represented by a `chan` structure containing the following fields.

`id`	the id of a channel; a natural number
`cap`	the channel capacity
`fldtypes`	the type of a message, represented as a list of type expressions
`contents`	a FILO list of messages

```
(defstructure chan
    (id       (:assert (naturalp id)))
    (cap      (:assert (and (naturalp cap)
                            (< cap (maxchannelcapacity)))))
    (fldtypes (:assert (type-list-p fldtypes)) )
    (contents (:assert (true-listp contents))))
```

PROMELA channel space is represented as a list of `chan` structures. Since the first element of a `chan` is its id, a list of `chan`'s can be interpreted as an alist whose domain is the set of ids.[1] We use `assoc` to look up a channel with a given id, and `put-tuple` to add or replace a channel in a list. In the following, `chan-list-p` recognizes a list of `chan` structures.

```
(deflist chan-list-p (l) chan-p)
```

`make-channel` takes as arguments a channel capacity, a message type, and a list of channels. It returns a multiple value consisting of a new channel id, and an updated channel list containing a new channel. `make-channels` creates `n` channels. It returns a multiple value consisting of `n` new channel ids, and a list of channels with the new channels added.

```
(defun make-channel (cap fldtypes channels)
  (declare (xargs :guard
                  (and (naturalp cap) (type-list-p fldtypes)
                       (< 0 (len fldtypes)) (chan-list-p channels))))
  (let ((id (excess-natural (domain channels))))
    (mv id
        (cons (make-chan :id id :cap cap :fldtypes fldtypes :contents
                  nil)
              channels))))
```

Variables

A variable, its type, and current value is represented by a `dcell`.

[1] This is an ugly hack that's too convenient to resist.

`name`	variable name (a symbol)
`type`	type of the variable; interpreted as primitive or an array, depending on whether `nel` is 0 or positive
`nel`	number of elements; 0 for a primitive type, positive for an array type
`val`	current value of the variable; one of: a number, channel id, or array (represented as a list)

```
(defstructure dcell
    (name (:assert (symbolp name)))
    (nel  (:assert (naturalp nel)))
    (type (:assert (type-p type)))
    (val  (:assert (in-type val type nel))))
```

The following functions are defined in the model. `dcell-list-p` recognizes a list of dcells. `dcell-names` collects the variable names of a list of dcells. `dcell-types` collects the types of a list of dcells.

We use `initial-dcell` to construct a dcell. `default-dcell` constructs a dcell using the default value of the type as initial value. `initial-dcells` creates a list of integer-valued dcells from a lists of names, types and initial values.

```
(defun initial-dcell (name type nel val)
  (declare (xargs :guard
                (and (symbolp name) (type-p type) (naturalp nel) (naturalp val))))
  (if (zerop nel)
      (make-dcell :name name :type type :nel (nfix nel) :val
          (nfix val))
      (make-dcell :name name :type type :nel nel :val
          (make-list nel :initial-element (nfix val)))))
```

`initial-chan` initializes a single channel or an array of channels, determined by `nel`. Two values are returned. The first is a dcell containing a variable whose type is a channel id or list of channel ids. The second is an updated channel list, with the appropriate number of new channels allocated.

Program Control

A program is represented by an s-expression. A location in a program, called a *pc*, is a path into an s-expression. A pc is a list of natural numbers. A pc is *ok* with respect to a program body if each of its elements is, successively, a valid index at the next level of the body. `pc-get` fetches the statement at position `pc` in `body`. `push-pc` is used for adding one level of depth to a given pc. `incr-pc` increments a pc at its deepest point. `pop-pc` drops the lowest `n` levels from a pc.

```
(defun pc-ok (pc body)
  (declare (xargs :guard (natural-listp pc)))
  (cond ((endp pc) t)
        (t (and (< (car pc) (len body))
                (pc-ok (cdr pc) (nth (car pc) body))))))

(defun pc-get (pc body)
  (declare (xargs :guard (and (natural-listp pc) (pc-ok pc body))))
  (cond ((atom pc) body)
        (t (pc-get (cdr pc) (nth (nfix (car pc)) body)))))

(defun incr-pc (n pc)
  (declare (xargs :guard
                  (and (naturalp n) (consp pc) (natural-listp pc))))
  (append (butlast pc 1) (list (+ n (car (last pc))))))
```

3.2 System State

Programs

A program label is a list of symbols. End labels are those whose first element is the symbol `'end`. A program body is an s-expression. We assume that a pre-processor removes labels from programs. A mapping from labels to locations in a program represents this information explicitly. A program is represented by a `pgm` structure.

`name`	program name
`args`	list of dcells containing the names and types of program parameters
`labels`	mapping from labels to positions in the program body
`body`	an s-expression representing a statement without internal labels

```
(defstructure pgm
    (name   (:assert (symbolp name)))
    (args   (:assert (dcell-list-p args)))
    (labels (:assert (label-to-location labels)))
    body)
```

Processes

A `process` structure represents the execution state of an individual process.

`id`	a unique natural number identifier
`name`	the name of the program (proctype name) the process is running
`active`	the program is eligible for execution
`pc`	program counter
`locals`	local variables
`children`	ids of child processes

```
(defstructure process
    (id       (:assert (naturalp id)))
    (name     (:assert (symbolp name)))
    (active   (:assert (booleanp active)))
    (pc       (:assert (natural-listp pc)))
    (locals   (:assert (dcell-list-p locals)))
    (children (:assert (natural-listp children))))
```

Global State

A `st` structure represents a PROMELA system state. It contains all processes, globals and and channels, as well as necessary control information. In the remainder of this report, the variable name `st` represents a global state.

`error`	a system error
`atomic`	a process id or nil
`timeout`	a flag used to handle timeout semantics
`globals`	global variables
`channels`	channels
`output`	a format string which can be displayed by the environment
`constants`	constant symbols
`processes`	list of processes
`pgms`	program definitions

```
(defstructure st
    (error     (:assert (true-listp error)))
    (atomic    (:assert (or (naturalp atomic) (null atomic))))
    (timeout   (:assert (booleanp timeout)))
    (globals   (:assert (dcell-list-p globals)))
    (channels  (:assert (chan-list-p channels)))
    (output    (:assert (true-listp output)))
    (constants (:assert (dcell-list-p constants)))
    (processes (:assert (process-list-p processes)))
    (pgms      (:assert (pgm-list-p pgms))))
```

The following supporting functions are defined on state. `fetch-stmt` returns the current program statement for a given process. `active-processes` collects the identifiers of the active processes in a list of processes. `all-inactive` is true if all of list of process ids identify inactive processes in a given state.

3.3 Expression Evaluation

The semantics of each operation is defined by a function which returns three results: an error value (`nil` means no error), an operation value, and an updated state. The state is required since one expression, `run`, has a side effect on state. We omit the definitions of most of the semantic functions, since they are obvious. See Section 2 for the set of operations that are handled.

`check-fn` defines the semantics of the channel test operator. This operator asks if there exists a message on a channel, and if so, whether it matches the pattern of constants and variables supplied by the list `convars`. The function `msg-pattern-match` defines the pattern matching algorithm: any constant in `convars` must equal the corresponding field in `msg`.

```
(defun msg-pattern-match (convars msg constants)
  (declare (xargs :guard (and (true-listp convars)
                              (true-listp msg)
                              (dcell-list-p constants))))
  (cond ((endp convars) (endp msg))
        ((endp msg) nil)
        ((integerp (car convars))
         (and (equal (car convars) (car msg))
              (msg-pattern-match (cdr convars) (cdr msg) constants)))
        (t (let ((dcell (assoc-equal (car convars) constants)))
             (and (implies dcell (equal (dcell-val dcell) (car msg)))
                  (msg-pattern-match (cdr convars) (cdr msg) constants))))))

(defun check-fn (chid convars st)
  (let ((chan (assoc chid (st-channels st))))
    (cond ((null chan) (mv (list 'bad 'channel 'id chid) 0 st))
          ((null (chan-contents chan)) (mv nil 0 st))
          (t (let ((msg (car (chan-contents chan))))
               (mv nil
                   (if (msg-pattern-match convars msg (st-constants st)) 1 0)
                   st)))))
```

The `run` operation returns an error flag, the identifier of a new process (meaningful only if no error occurs), and an updated state. The initial set of local variables for the new process consists of its formal parameters bound to the actual arguments. The new process identifier is recorded in the parent process.

```
(defun run-fn (pgmname args pid st)
  (if (>= (len (active-processes (st-processes st))) (maxprocesses))
      (mv '(process limit exceeded) 0 st)
    (let ((pgm (assoc-equal pgmname (st-pgms st))))
      (cond ((null pgm) (mv (list 'unknown 'program pgmname) 0 st))
            (t (let* ((newpid (len (st-processes st)))
                      (newprocess (make-process :id newpid
                                                :name pgmname
                                                :locals
                                                (initial-dcells
                                                 (dcell-names (pgm-args pgm))
                                                 (dcell-types (pgm-args pgm))
                                                 args))))
                 (mv nil
                     newpid
                     (update-st st :processes
                                (append (add-child newpid pid st)
                                        (list newprocess)))))))))
```

The function `evl` takes an expression, a process identifier and a state as arguments. It returns an `(error, value, state)` triple as discussed above. `evl-list` takes as arguments a list of expressions, a process id and state. The value component of the triple returned by `evl-list` is a list of expression values. These functions do a pattern match on the form of an expression, and dispatch the appropriate semantic function. We omit most of the details.

```
(defun evl (expr pid st)
  (declare (xargs :guard (and (naturalp pid) (st-p st)
                              (< pid (len (st-processes st))))))
  (cond ((integerp expr) (mv nil expr st))
        ((symbolp expr)  (let ((dcell (assoc-equal expr (variable-env pid st))))
                           (cond (dcell (mv nil (dcell-val dcell) st))
                                 (t (mv `(unbound symbol ,expr) 0 st)))))
        ((consp expr)
         (case-match expr
           (('== a b)              (mv-let (err args st1) (evl-list (list a b) pid st)
                                           (if err (mv err 0 st)
                                             (eq-fn (car args) (cadr args) st1))))

           (('+  a b)              (mv-let (err args st1) (evl-list (list a b) pid st)
                                           (if err (mv err 0 st)
                                             (add-fn (car args) (cadr args) st1))))

           ...

           (('RUN (pgmname . args)) (mv-let (err vals st1) (evl-list args pid st)
                                           (if err (mv err 0 st)
                                             (run-fn pgmname vals pid st1))))

           (('TIMEOUT)             (if (st-timeout st)
                                       (mv nil 1 (update-st st :timeout nil))
                                     (mv nil 0 st)))

           (&                      (mv `(unrecognized opr ,(car expr)) 0 st))))
        (t (mv `(unrecognized expr ,expr) 0 st))))

(defun evl-list (exprs pid st)
  (declare (xargs :guard (and (naturalp pid) (st-p st)
                              (< pid (len (st-processes st))))))
  (cond ((endp exprs) (mv nil nil st))
        (t (mv-let (err1 val1 st1) (evl (car exprs) pid st)
                   (mv-let (err2 vals st2) (evl-list (cdr exprs) pid st1)
                           (mv (or err1 err2)
                               (cons val1 vals)
                               st2))))))
```

3.4 Executability

A statement s in behalf of active process p is executable if either no atomic transition is in progress, or p is executing atomically and some statement-specific condition is satisfied.

Many of the statements are always executable. Here we present in some detail the executability of expressions and the send statement.

An expression is executable (`evl-executable`) if it evaluates to a non-zero value without error. A list of expressions is executable (`evl-list-executable`) if all execute without error.

```
(defun evl-executable (expr pid st)
  (mv-let (err val st1) (evl expr pid st) (declare (ignore st1))
          (and (not err) (not (zerop val)))))

(defun evl-list-executable (exprs pid st)
  (mv-let (err vals st1) (evl-list exprs pid st)
          (declare (ignore vals st1)) (not err)))
```

A send operation is executable if it refers to an existing channel, and it is an asynchronous send to a channel that is not full, or it is a synchronous send to a process that is ready to receive the given message. The definition of an executable receive is analogous. (We omit the definition of the algorithm that detects a matching receive for a synchronous send.)

```
(defun send-executable (var-ref exprs pid st)
  (mv-let (err chid st1) (evl var-ref pid st)
          (if err nil
            (let ((chan (assoc chid (st-channels st))))
              (and chan
                   (or (and (< 0 (chan-cap chan))
                            (< (len (chan-contents chan))
                               (chan-cap chan)))
                       (and (= 0 (chan-cap chan))
                            (mv-let (err msg st2) (evl-list exprs pid st1)
                                    (if err
                                        nil
                                      (some-process-has-matching-receive
                                       chid msg st))))))))))
```

The following forms complete the definition of executability. The toplevel predicate is `executable`, which determines if the given process can execute in given current state.

```
(defun stmt-executable (stmt pid st)
  (case-match stmt
    (('assert expr)              t)
    (('assign var-ref expr)      t)
    (('atomic . stmts)           (or (null stmts)
                                     (stmt-executable (car stmts) pid st)))
    (('break)                    t)
    (('do . options)             (some-stmt-executable options pid st))
    (('end)                      (let ((p (nth pid (st-processes st))))
                                   (all-inactive (process-children p) st)))
    (('goto label)               t)
    (('if . options)             (some-stmt-executable options pid st))
    (('printf string . exprs)    t)
```

```
    (('option . stmts)            (or (null stmts)
                                      (stmt-executable (car stmts) pid st)))
    (('receive var-ref . convars) (receive-executable var-ref convars pid st))
    (('send var-ref . exprs)      (send-executable var-ref exprs pid st))
    (('seq . stmts)               (or (null stmts)
                                      (stmt-executable (car stmts) pid st)))
    (('skip)                      t)
    (('bit . ivars)               t)
    (('bool . ivars)              t)
    (('byte . ivars)              t)
    (('chan . ivars)              t)
    (('short . ivars)             t)
    (('int . ivars)               t)
    (& (evl-executable stmt pid st))))

(defun executable (pid st)
  (declare (xargs :guard (and (naturalp pid) (st-p st))))
  (and (or (null (st-atomic st))
           (equal (st-atomic st) pid))
       (stmt-executable (fetch-stmt pid st) pid st)))
```

3.5 Statements

The effect each statement has on state is defined by a semantic function. We display only a few of them. In each, the function `finish-step` is responsible for incrementing the program counter and marking exit from an atomic region.

Assertion

An assertion expression is evaluated. If an error occurs or the expression evaluates to zero, the an state error is signaled. Otherwise, control passes to the next statement.

```
(defun assert-fn (expr pid st)
  (mv-let (err val st1) (evl expr pid st)
          (if err (update-st st :error err)
              (if (equal val 0)
                  (update-st st :error (list 'assertion 'failure expr))
                  (finish-step pid st1)))))
```

End

The end operation de-activates a process if all its children are inactive.

```
(defun end-fn (pid st)
  (let ((p (nth pid (st-processes st))))
    (update-st st :processes
               (put-nth pid
                        (update-process p :active
                                        (not (all-inactive (process-children p) st)))
                        (st-processes st)))))
```

Send

An asynchronous send adds a message to a non-full channel. A synchronous send performs a rendezvous with a matching receive operation. `send-fn` performs a synchronous or asynchronous send depending on the capacity of the specified channel.

```
(defun send-fn (var-ref exprs pid st)
  (mv-let (errs msg st1) (evl-list exprs pid st)
          (if (some-error errs)
              (update-st st :error (report-errors errs))
            (mv-let (err id st2) (evl var-ref pid st1)
                    (if (or err (not (bound? id (st-channels st2))))
                        (update-st st :error (list 'bad 'channel var-ref))
                      (let ((chan (assoc-equal id (st-channels st2))))
                        (cond ((zerop (chan-cap chan))
                               (synchronous-send chan msg pid st2))
                              (t (asynchronous-send chan msg pid st2)))))))))
```

3.6 Step Function

The function `step-stmt` (definition omitted) interprets a statement in behalf of a given process. Primitive statements are handled by the individual semantic functions. Compound statements including `atomic`, `do`, `if`, `seq` and `option` are defined as well.

`step-fn` is the top-level function in the semantic definition. It takes a step in behalf of the indicated process. A new state is returned. A stuttering step is taken on a state in which an error is recorded, or if the selected process is inactive. Otherwise, `step-fn1` is applied with the `timeout` flag set if no statement is executable. A `timeout` expression is not executable in a state in which this flag is false, otherwise it is executable. Finally, if the current executable statement is interpreted and a resulting state is returned.

```
(defun step-fn1 (pid st)
  (cond ((not (executable pid st)) st)
        (t (step-stmt (fetch-stmt pid st) pid st))))

(defun step-fn (pid st)
  (declare (xargs :guard (and (naturalp pid) (st-p st) (not (st-timeout st))
                              (< pid (len (st-processes st))))))
  (cond ((st-error st) st)
        ((not (process-active (nth pid (st-processes st)))) st)
        (t (step-fn1 pid (update-st st :output nil
                            :timeout (all-blocked (active-procs st) st))))))
```

4 Comparison with Natarajan and Holzmann

The gross structure of our model resembles the semantic definition presented by Natarajan and Holzmann. Both describe a state transition system. The structure of state in [NH96] consists of a collection of labeled transition systems (LTS) of the form

$$< Name, Structure, Start, locals, ChansOwned, Active, Param >,$$

a set of local states of the form

$$< Name, pid, pc, locals, lChansOwned >,$$

a global state of the form $< X, g >$, where X is a set of local states and g is a structure of the form

$$< globals, gChansOwned, NrProcs, Handshake, Exclusive,$$

$$ChIdMap, ChContMap, timeout, dstep > .$$

Our `pgm` structure roughly corresponds to their LTS. However, we do not support the notion of a static collection of locals and channels assigned to a program. Rather, we have chosen to model the dynamic allocation of variables and channels. It would seem that the *Active* field is more appropriately associated with a local state than with an LTS. Also, their *Structure* includes dynamic aspects of computation, such as execution mode (*e.g.*, atomic, deterministic) and priority. We don't handle priorities, but again, it seems that this information would be more naturally modeled as part of local state.

Our `process` structure corresponds to their local state. Again, we omit the notion of owned channels. We include an `active` property in a `process` structure.

Our `st` structure corresponds to their $< X, g >$ pair. The `processes` field of an `st` is analogous to their set of local states X. The rest of the `st` structure maps more or less to their global structure g. We collapse their *ChIdMap* and *ChContMap* into a single list of channel structures. Their *Exclusive* is our `atomic`; their *globals* is our `globals`. We compute *NrProcs* from the list of processes. We omit a notion corresponding to *Handshake*, and model synchronous message communication without a hidden intermediate state. We have adopted their strategy for modeling timeouts. We don't handle deterministic steps (yet).

There are various differences in the details of how we define steps. They are suggested by the differences in state structure, so we do not enumerate them. Additionally, the details will likely be modified in response to further scrutiny of the model.

5 Running the Model

In this section we summarize the status of this model as a software system. The model relies on the following software.

ACL2. The logic of ACL2 is very close to a subset of Common Lisp. ACL2 itself is implemented in Common Lisp. See `http://www.cli.com` for instructions on downloading the latest version of ACL2.

ACL2 libraries. Some public libraries are included in the ACL2 distribution. We have modified these slightly for the purposes of building this model. When this model becomes available, all pertinent libraries will be distributed with it.

The PROMELA model. This is an ACL2 script containing all the definitions and lemmas supporting the introduction of `step-fn`.

The simulation environment. To be able to run programs, we have defined a very simple interpreter in Common Lisp, which repeatedly applies `step-fn`. This interpreter randomly selects an active process for execution until all processes have terminated, or a system error occurs. Rudimentary support for running PROMELA programs is present, including step tracing.

In general, the PROMELA-ACL2 system is neither robust nor complete. Should there be any interest in carrying this work forward, a review of the model will lead to its revision. The attempt to prove properties of the model will require some library development. A reasonable interface to a simulation environment would be desirable.

6 A Brief Introduction to ACL2

In this section we summarize some of the features of ACL2. We rely to a large extent on the reader's assumed familiarity with Common Lisp. An ACL2 *script* is a file of forms which is processed in the order presented. This summary describes some of the forms which can appear in a script.

Defun. The `defun` form creates a function. In the example below, the function `foo` is defined. `foo` has two arguments, `x` and `y`. Assumptions about the arguments are declared (optionally) in the guard. In this example, `x` and `y` are declared to be integers. The guard is evaluated at run time, and causes an error if it is not satisfied. Following the declaration is the function body. A termination proof must be completed before a function is accepted as a logical extension of ACL2.

```
(defun foo (x y)
  (declare (xargs :guard (and (integerp x) (integerp y))))
  (* (+ x y) 2))
```

Guard Verification. One can avoid the cost of the guard check by *verifying guards* with the theorem prover. Verifying the guards of a function `f` demonstrates that the guards to all functions that `f` calls are satisfied at their point of call. One can tell the ACL2 system to automatically attempt or avoid a guard verification proof at the time that a function is introduced. This setting can be overridden by an explicit hint.

Multiple Values. A function may return more than one value. One way of returning multiple values is to return a list of values. However, using the ACL2 multiple value primitives, `mv` and `mv-let`, allows the system to check for the right number of values at the time a definition is processed. In this example, `dog` returns a multiple value, and `cat` uses an `mv-let` to unbind the values. `cat` returns the sum of a number and its double. `mv` and `mv-let` correspond to Common Lisp's `values` and `multiple-value-bind`.

```
(defun dog (x)
  (declare (xargs :guard (integerp x)))
  (mv x (* 2 x)))

(defun cat (y)
  (declare (xargs :guard (integerp y)))
  (mv-let (i j) (dog y)
     (+ i j)))
```

Defthm. A `defthm` form proposes a theorem about previously introduced functions. The mechanical proof checker within ACL2 attempts a proof of the proposed theorem. In this example, we suggest the theorem that the function `foo` returns an even number.

```
(defthm evenp-foo
        (implies (and (integerp x)
                      (integerp y))
                 (evenp (foo x y))))
```

Case-Match. `case-match` is a control-flow and pattern matching construct. The first argument of `case-match` is a variable. Remaining arguments are pairs of patterns and expressions. The patterns are attempted in order, and when a match is found, the corresponding expression is evaluated. A pattern match dynamically binds variables from the pattern to values from the form. The pattern `&` matches anything, and is used as the default.

```
(case-match form
   (('and x y) (foo x y))
   (('or x y)  (bar x y))
   (&          nil))
```

Deflist. `deflist` is a macro defined by the author that generates a recursive function which recognizes a list, all of whose elements satisfy a given unary predicate. Additionally, it automatically generates a large number of `defthm` forms that inform the theorem prover of important properties of the new function. The following example introduces a function `integer-listp` that recognizes a list if integers.

```
(deflist integer-listp (l) integerp)
```

Defstructure. `defstructure` is a macro that provides a capability similar to Common Lisp's `defstruct`. It allows one to define a record structure, including its accessor, constructor and update functions. The following example defines a `person` record structure consisting of `height` and `weight` fields. The fields are constrained to be numbers.

```
(defstructure person
    (height (:assert (numberp height)))
    (weight (:assert (numberp weight))))
```

The automatically generated functions include the following.

`(make-person :height h :weight w)`	construct a `person` structure
`(person-height p)`	access the `height` field of a person `p`
`(person-weight p)`	access the `weight` field of `p`
`(update-person p :weight w)`	update the `weight` field of `p`
`(person-p p)`	a predicate that recognizes a `person` structure, including type constraints on the fields

References

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design, a Foundation.* Addison Wesley, 1988.

[Hol91] Gerard Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[KM94] M.J. Kaufmann and J S. Moore. Design goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, December 1991.

[NH96] V. Natarajan and Gerard J. Holzmann. Outline for an operational-semantics definition of PROMELA. Technical report, Bell Laboratories, July 1996. Can be found at http://netlib.bell-labs.com/netlib/spin/ws96/papers.html.