System Construction with Object-Oriented Pictures

George W. Cherry
Thought**Tools, Inc.
5151 Emerson Road
Canandaigua, New York 14425

Abstract and Introduction

This paper describes the process guide, behavior model, icons, and diagrams of SCOOP-3, a pictorial method for developing reactive systems. SCOOP-3's semantics are Concurrent C++ or Ada: its icons and diagrams are mechanically translatable to these languages. SCOOP-3's process guide supports reuse, prototyping, and concurrent specification and design. Its behavior model (S-R Machines) integrates the notions of finite and infinite automata, data abstractions, and objects. My earlier notes in SEN demonstrated the Finite State Automata power of S-R Machines. This note demonstrates their far greater power and expressiveness. SCOOP-3's black box, machine, and clear box diagrams have the same objective (and names!) as Mills' Box-Structured approach: stepwise provable specifications and designs.

SCOOP-3 Technical Process Guide

Figure A is SCOOP-3's technical process guide. It supports prototyping (it assumes the purpose of the jth system is to satisfy the jth set of requirements and evoke the (j + 1)th set of requirements). It supports reuse (for example, it places its explicit reuse step, process 2, early in the development cycle). It supports concurrent specification and design (we'll discuss this below). Notice that the work products of Figure A are three views of a system: black box, library, and clear box (which includes state machines). The black box is the system's expected external behavior. (Some writers use the terms interface or specification for black box.) The library is the system's set of reusable components, recruited to support its clear box. The clear box is its internal mechanisms, which generate its black box behavior. (Some writers use the terms body or implementation for clear box.)

Stimulus-Response (S-R) Machines

Central to SCOOP-3 is its notion of an S-R Machine. S-R Machines are modules of behavior; that is, they are architectural entities as well as behavior entities. Complex systems are built from networks of S-R Machines, which communicate with one another by message or call stimuli/responses. Figure B is the schematic of an S-R Machine. Its triangles are computations which operate on their inputs to produce their outputs. Its rectangles (open at one end) are memory (possibly unbounded in the case of "data..."). We have provided Figure B to help you visualize the following definition. A general S-R machine is an 9-tuple:

1. a finite set of kinds of stimuli (the S's in the input stream are members of this set),
2. a finite set of kinds of responses, (the R's in the output stream are members of this set),
3. a finite set of state values (the values stored in the variable "state" are members of this set),
4. a distinguished state value (often called the start state),
5. a data type ("data..." object belongs to this type),
6. Guards: Boolean functions computed from (state, stimulus, data) triples (they determine which Stimuli are currently acceptable),
7. Next: computes the next state from the current (state, stimulus, data) triple,
8. Output: computes a response from the current (state, stimulus, data) triple,
9. Update: computes a data update from the current (state, stimulus, data) triple.

We usually encapsulate "data...", its update procedures, and its query functions in an Data Abstraction. (Note the dashed lines enclosing these entities in Figure B.)

To draw Figure B we had to stand on the shoulders of giants. Figure B encompasses all kinds of automata (for example, Finite State Machines and Pushdown Automata), Data Abstractions, the Objects of the double O school (OOPL, OORA, OOD, ...), procedural abstraction, variables, files, and the notions of data encapsulation, information hiding, strong module cohesion, nondeterministic waiting, and guarded accept statements (to mention some–not all).

For example, delete "Guards" and the entire "Data Part" and you have a classical (Mealy) Finite State Machine. (But keep the guards to reduce the blobs and arrows on State Transition Diagrams by orders of magnitude. See [2]) Delete "Guards" and you have a classical Data Abstraction. Use Concurrent C++ or Ada data structures and structured control statements in the blobs on Figure B, and your have a machine with the deep simplicity, elegance, and power of Turing machines–but without their maddening primitiveness. Delete "Guards", the persistent "state" and "data..." (and the functions that update them) and you have functional or procedural abstractions.

It is easy to show that Figure B is Turing Machine Equivalent, and therefore capable of representing any formulaic computation: simply take "data..." to be a Turing tape and the Update and read functions to be Turing's basic tape operations. This point may seem purely academic; but it's very practical. We can't solve all problems with networks of communicating Finite State Machines or hierarchies of Data Abstractions. We can solve all problems (solvable by computers) with S-R Machines. (Some of the OO literature implies that Data Abstractions are enough; they aren't: they must be orchestrated by some kind of step machine. The object viewpoint is very good at modelling the world as things; but in its eagerness to reduce the world to things, it has given less than enough attention to process and orchestrating behavior.)

The machine in Figure B repeatedly executes the following behavior cycle. The machine evaluates its Guards (if any), and then waits for an acceptable stimulus (a stimulus to an open accept statement), accepts one when it arrives, computes a data update, computes its response (if any), and computes its next state; then it revaluates its guards and starts another cycle.

Concurrent C++ and Ada offer elegant ways to build S-R Machines, while SCOOP-3 offers elegant graphical representations for Concurrent C++ and Ada. In the remaining section we'll give examples. We've chosen Ada for these examples. The diagrams are similar for Concurrent C++.

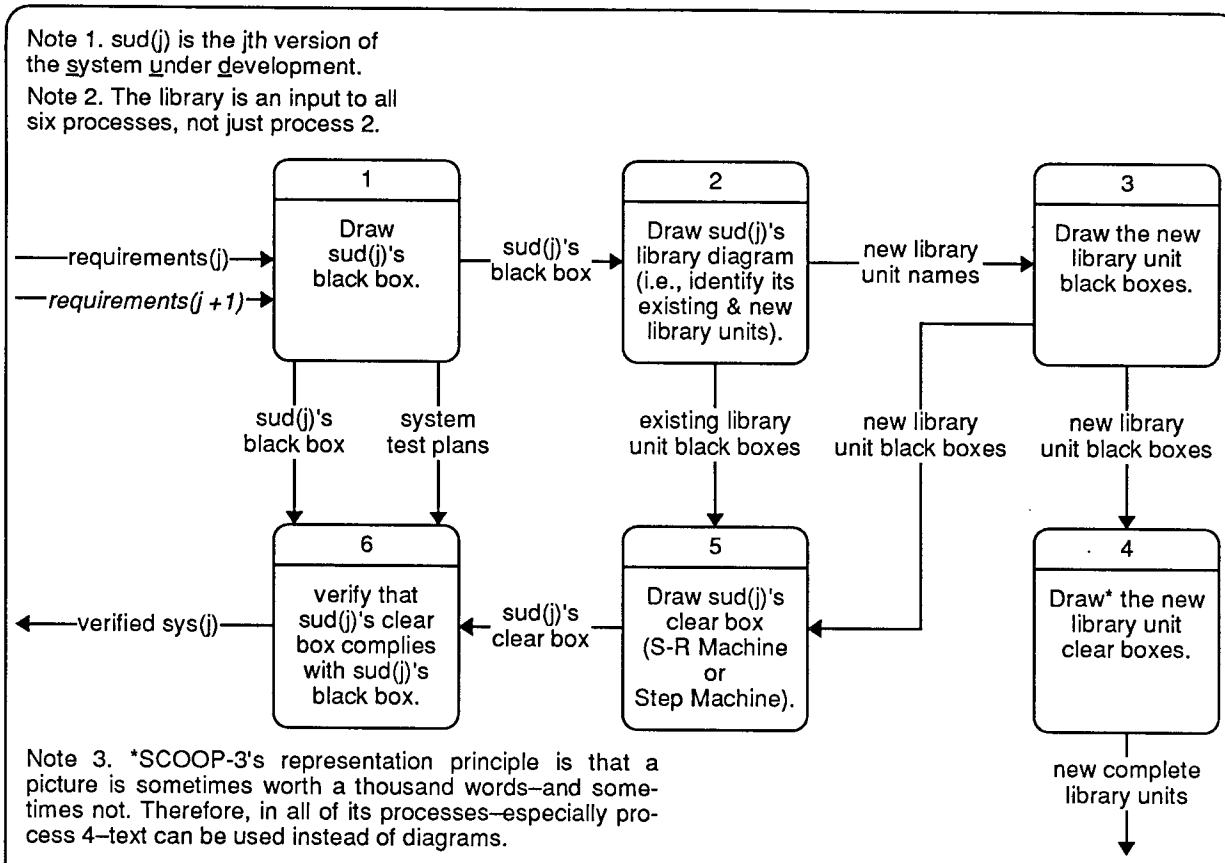


Figure A. Process for "Software Construction with Object-Oriented Pictures"

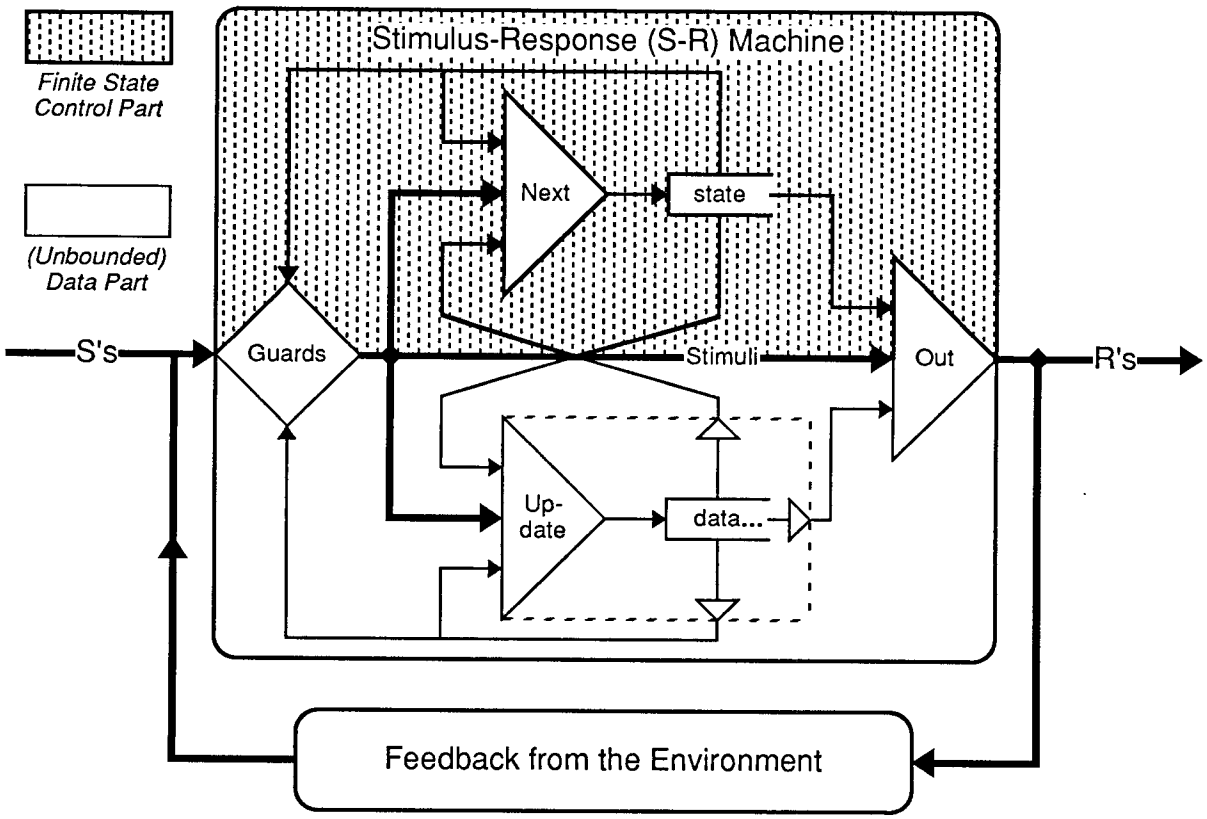


Figure B. Schematic S-R Machine (Turing Machine Equivalent)

Module Icons

Module icons must set the values of five attributes: **1.** library (heavy border) or non-library (thin border); **2.** package (doubled interior lines) or non-package (single interior line); **3.** concurrent (rounded corners) or sequential (angled corners); **4.** black box (black fill) or clear box (clear fill); **5.** generic (cross-hatched border) or ordinary (plain border). For example, the icon at the top of Figure C represents a generic, sequential, library, package black box. (Optionally, the top field of a unit icon displays its part number rather than the "task", "package", "procedure", "function" notation, which we used in this paper's figures. The explicit notation isn't necessary to identify the kind of program unit: the graphics do that; however, the explicit notation is helpful in introductory or teaching situations.)

An S-R Machine Stack Example

Figures C-D are a Stack, an example of a Data Abstraction S-R Machine. Figure C is its black box. Figure C's arrows are stimuli: calls, or, in OO's suggestive jargon, messages. The arrows point in the direction of the calls (message flow). Calls may have parameters, which may be of mode in (!), in out (!?), or out (?). Therefore, !? and ? parameters provide responses to call stimuli. Alternatively, the response to a call may be an exception (enclosed in square brackets in SCOOP-3). Function calls (a function is a value-returning subprogram) are distinguished from procedure calls by the ? prefix. If a call is parameterless, SCOOP-3 uses empty parentheses. Thus, an arrow whose label contains (...) is the basic call icon.

Figure D is the Stack's clear box. Its state variable is "top"; its data variable (bounded in this case) is "bin". Consider the stimulus "push(! item)"; it's accepted when issued (that is, it's not guarded). If top < size, then push's Next function is "top := top + 1"; its Update function is "bin(top) := item"; and it has no overt response. If top is not less than size, the push stimulus will not modify top or bin; however, it will elicit the overt response of the full_stack exception. Consider the stimulus (a function call) "? count()"; its response is to return the value stored in the state variable top. Note that many of the connections in Figure B do not appear in Figure D. Figure B is general (or generic); while Figure D is a specific instance of an S-R Machine.

A Reactive Example: A Drunk-Proofed, Theft-Alarmed Car Lock

We now give an example large enough to illustrate SCOOP-3's process guide and its three kinds of diagrams. Our example is a pushbutton car lock, intended to deter thieves as well as prevent drunk drivers from operating their cars. Our strategy to deter thieves is to turn on a burglar alarm when someone tries to unlock the car with an incorrect sequence of digits (that is, a sequence not equal to the combination). Our strategy to prevent drunk drivers from operating their cars is to require drivers to input the correct sequence of digits rapidly. (The data entry speed and skill of drunk subjects is badly degraded.) The pushbutton car lock could be used to operate the car door lock or the ignition.

Process 1 (Figure A): Draw the system's black box (Figure 1). The purpose of a system's black box is to describe the system's expected external behavior. The black box should depict the system's external objects, the stimuli they generate, and the responses they absorb. It should also describe the system's or component's program (syntactic) interface. Finally, the black box should capture behavior by means of one of the following: by a table of preconditions, stimuli, responses, and postconditions (for examples, see Figure C and Figure 3); by stimulus-response traces (for an example, see Figure 1); or by a reference to the user's guide for an interactive system.

Figure 1 is the black box for our Car Lock. It shows that pressing any digit button generates "a_digit" interrupt whilst pressing the 'L' button generates a "a_lock_request" interrupt. Note the $ sign next to the input port for digit: this sign indicates the Car Lock may time out of waiting for a digit. (The origin of this time-out icon is, "Time is money–$".) What looks like gibberish on the bottom of Figure 1 are S-R traces: we've used them to describe the Car Lock's expected external behavior.

For simple systems and components, we may use a set of stimulus-response traces; a table of preconditions, S's, R's, and postconditions; or a reference to a user's guide, to describe the unit's expected external behavior. For behaviorally complex systems these strategies are inadequate; we need a modelling language that can describe data, modes, or states, and structured mechanisms for manipulating them. Statecharts and Structured Analysis (supplemented by State Transition Diagrams) are frequently used to describe complex reactive and interactive behavior. These notations are often used to write the system's SRS (i.e., describe its expected external behavior). SCOOP-3 offers an alternative to these notations. For example, Figures 2-8 and especially Figures 5-8 are an elegant way to write an SRS–that is, describe expected external behavior.

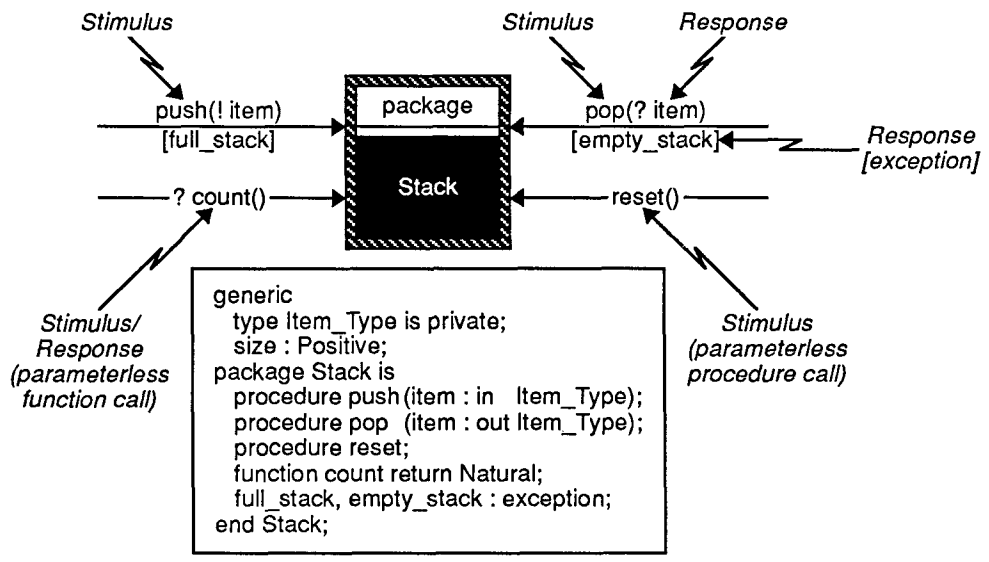
Process 2 (Figure A): Draw the system's library (Figure 2). Some of the behaviors described by the black box may be (and therefore should be) woven from reused or reusable components. Process 2 is an implementation step since it identifies library units we'll use to build our "new" unit. Note (Figure B) that an S-R Machine distinguishes state and data. The list package encapsulates the data part of the Car Lock S-R Machine.

Process 3 (Figure A): Draw the new library unit black box(es) (Figure 3). Process 3 specifies the black boxes of any new library units named in Process 2. Process 3 should be conducted jointly with the developer responsible for process 5, who will be these library units' first user. (A good way to promote a library unit's reusability is to ensure its usability during first use.) Figure 3 is the package list's black box .

Process 4 (Figure A): Draw the new library unit clear box(es) (Figure 4). Figure A's processes 4 and 5 may be conducted concurrently–in accordance, of course, with the library unit black box contracts. Figure 4 contains the clear box of the package list, a data abstraction.

Process 5 (Figure A): Draw the system's clear box (Figures 5, 6, 7, 8). The system's clear box should identify its state (or mode) machines. Figure 5 is a the Car Lock's clear box (in terms of its state machines). In Figure 5, the state machines are black boxes: we must make clear what they do later.

We're concerned here with reactive systems, systems we characterize as event-driven or stimulus-driven. It's not that the environment can shove these systems around; but that the environment determines a stimulus sequence and the system determines an associated response sequence, a kind of dance. The environment and the system interact as practically equal partners, each making its contribution to observable behav-



Precondition	Stimulus	Response	Postcondition
()	count call	return 0	()
()	push(a) call	----------	(a)
(a)	push(b) call	----------	(a, b)
(a, b, c)	pop(x) call	return x = c	(a, b)
(a, b, c)	reset call	----------	()
(a, b, c)	count call	return 3	(a, b, c)
(a, b, c, d)	push(e) call	full_stack	(a, b, c, d)
()	pop(x)	empty_stack	()

Figure C. A Stack S-R Machine Black Box (assuming size = 4)

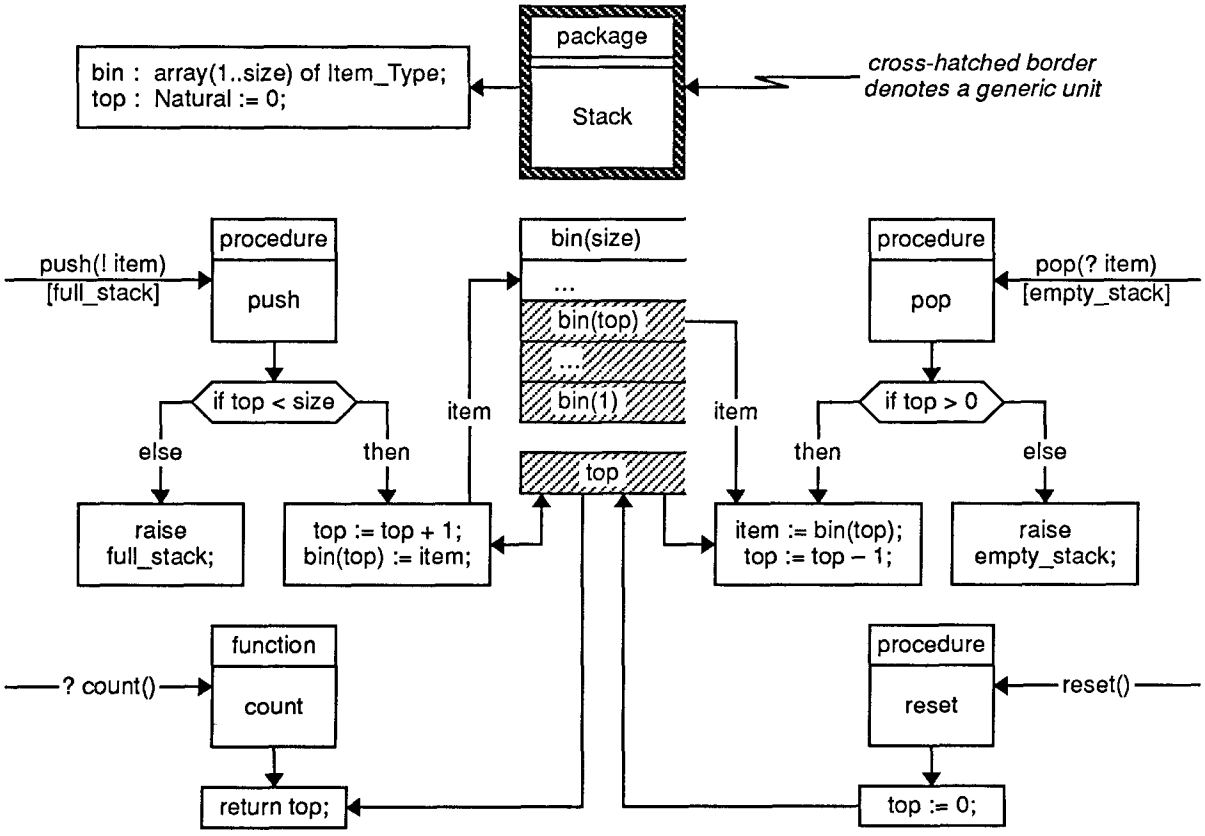


Figure D. A Stack S-R Machine Clear Box

ior. A reactive system usually has a reaction loop, which allows it to repeatedly select a stimulus and react to it appropriately. A reactive system is similar to a martial arts expert, who sidesteps some stimuli and accepts and reacts to others, making the appropriate change of state and response.

We believe that such systems are best designed by describing their stimuli and what to do about them–when to accept them and how to react to them. We have used an equivalence relation on stimuli, "is accepted in the same task states as", to partition the Car Lock task into three lower-level state machines: an "in every state" machine that accepts 'L' stimuli, an "awaiting d1" machine that accepts d1 stimuli, and an "awaiting d2 thru dn" machine that accepts d2, d3, ..., dn stimuli..

(We've found equivalence relations on stimuli the most useful of all partitioning principles. Reference 1 applies this principle to more complex systems and also describes other approaches to partitioning. The following hierarchy of partitions is very useful.

1. Packages of tasks for stimuli accepted in the same set of orthogonal states. (This use of "orthogonal" is due to Harel, [4].) A machine involves orthogonal states when its description uses concurrent threads of execution. Each thread of execution has associated with it a sequence of states, which is orthogonal to the states in other threads of execution;
2. Tasks for stimuli accepted in the same sequence of states;
3. Task states for stimuli accepted in some subset of a task's set of states;
4. Task accept statements for stimuli accepted by the same accept statement.)

Figure 6 is the clear box for "in every state". Its accept statement waits nonbusily in every state for an interrupt caused by someone pressing 'L'. Its Out function is "output(lock_car)"; its Update operation is "list.be_empty"; and it Next transition is "state := awaiting_d1". It's a communication nicety (not really necessary) to show graphically the state machine's sending a "be empty" message to list.

We recommend that engineers give the preconditions for each state machine's correct operation and the postconditions established by each state machine. For the most part, these conditions are self-documenting (as in the example of the text box "leaf" in the bottom center of Figure 6.)

Figure 7 is awaiting_d1's clear box. For this clear box to function correctly, its predecessors must lock the car and empty the list and the length of the combination must be at least 2. Please check now to see that Figures 6 and 8 establish (or maintain) the first two conditions, before they transfer control to awaiting_d1.

Figure 8 is the clear box for "awaiting d2 thru dn". This machine times out of its wait for a digit after "maximum interdigit delay". It's implemented by a selective wait statement with a delay alternative (a timeout), a statement with identical semantics in Concurrent C and Ada. Because Figure 8 contains the most complex logic we've seen so far, we'll explain in detail the execution of the S-R Machine in this state. Its first act (see Figure 5) when it executes another cycle of its reaction loop is to evaluate its guards to determine which state machines are open (that is, which state machines are currently receptive to a stimulus). In the "awaiting d2 thru dn" state, the open machines are "in every state" and, of course, "state = awaiting d2 thru dn". Thus the accept alternative on Figure 6 and the accept alternative and delay alternative on Figure 8 are all open. The delay immediately starts counting down, and the S-R Machine commences a wait for either an 'L' stimulus, a digit stimulus, or the expiration of the delay. If the delay expires before an 'L' or digit arrives, the clear box on Figure 8 executes the delay statement's associated textbox (statements). But if a digit arrives before an 'L' stimulus or the delay expiration, the accept statement for "a digit" executes its associated statements. Note that there are two execution arrows departing this accept icon. These control flows are ordered. We specify execution order by the following convention: locate the execution arrow with an "o" (for order or origin) on its tail, execute it, and then execute the other arrows, in a counter-clockwise order. You should have no difficulty reading the rest of Figure 8.

Process 6 (Figure A): Verify the system's clear box complies with its black box (Proof Table). We agree with Harlan Mills that managers shouldn't let developers execute their programs. (A different team should do the testing.) Managers should tax developers to prove their programs correct, by reasoning, proof tables, and inspections (inspections are especially effective, because they bring other human intelligence and knowledge of the world to bear on the problem and its solution). (We've found that inspections of specifications or designs in the form of Figures 2-8 are about twice as effective as inspections of source text or other diagramming schemes, and that our formal, intuitive guidance and notation create programs of high conceptual clarity and integrity.) If developers execute their programs, they will be tempted to test whether they work, rather than prove that they work. There's a world of quality difference between a system that works because of state-by-state verification and conceptual integrity, and a system that works because its developers have removed its readily detectable errors.

The traces in Figure 1 constitute system test scenarios. To verify the S-R Machine's body, we must prove it generates these traces. For example, the Proof Table (which follows the Car Lock Figures) shows how the S-R Machine and its clear boxes generate the last trace in Figure 1. (The symbol ¬ in the New State column means "go to the beginning of the next row.")

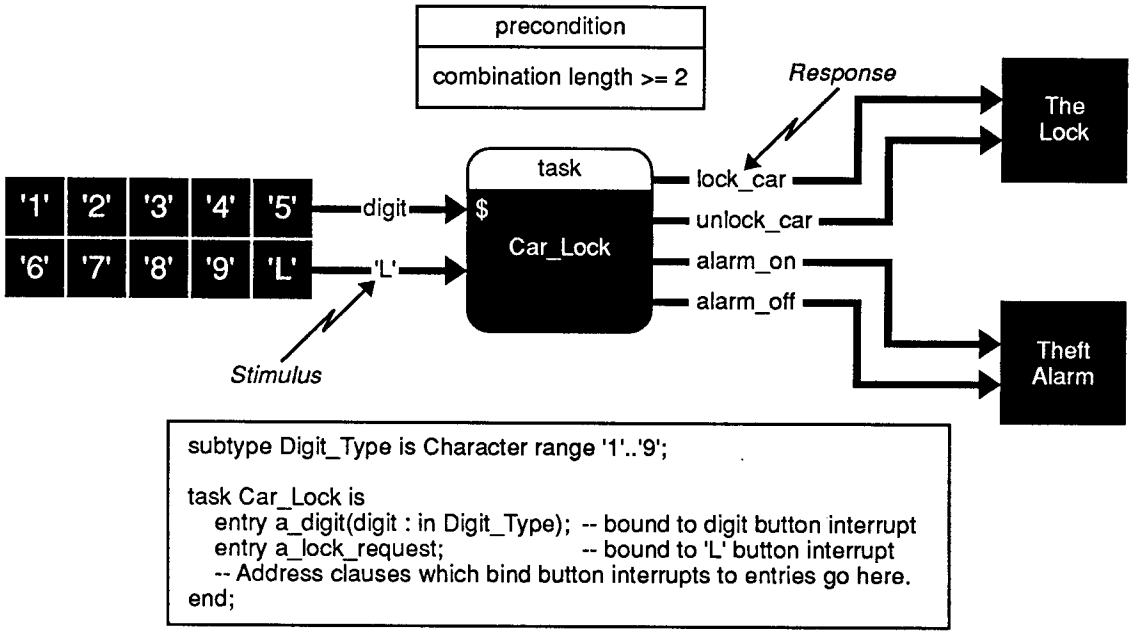
DeBalkanizing the Life-Cycle: Concurrent Specification and Design

In the traditional waterfall life-cycle model, system engineers write (in an informal language, usually) an SRS, deliver it to the customers as a description of what the developer will do, and toss it over the transom to software engineers, who actually design and implement the system.

Here's Dave Parnas' description [3] of this approach:

"Systems engineers do what comes before the software is written. They write large amounts of documents, they pile them on a government contract monitors desk, he accepts them, pays for them, they gather dust and all the work starts over again and that's why we never see that stuff. ...They do the system engineering and then somebody sits down and really designs the system.."

Given the poor (but expensive!) results of this approach, we should be willing to try concurrent specification and design. In this approach, instead of the systems engineers tossing an SRS over the transom, management tosses software engineers over the transom to system engineering, and systems engineers and software engineers work together to pro-




```
subtype Digit_Type is Character range '1'..'9';

task Car_Lock is
   entry a_digit(digit : in Digit_Type);  -- bound to digit button interrupt
   entry a_lock_request;                  -- bound to 'L' button interrupt
   -- Address clauses which bind button interrupts to entries go here.
end;
```

S-R traces for combination = ('1', '3', '6', '9').

notes: S* = S arrived late; NR = No Response

< unlock_car, 'L', lock_car, '1', '3', '6', '9', alarm_off, unlock_car, '1', '3', '6', '9', NR >
< unlock_car, 'L', lock_car, '1', '3', '9', '6', alarm_on, '1', '3', '6', '9', alarm_off, unlock_car>
< unlock_car, 'L', lock_car, '1', '3', '6', '9'*, NR, 'L', lock_car, '1', '3', '6', '9', alarm_off, unlock_car>
< unlock_car, 'L', lock_car, '1', '3', '6', '9'*, NR, '1', '3', '6', alarm_on, '1', '3', '6', '9', alarm_off, unlock_car>

Figure 1. Car Lock Black Box

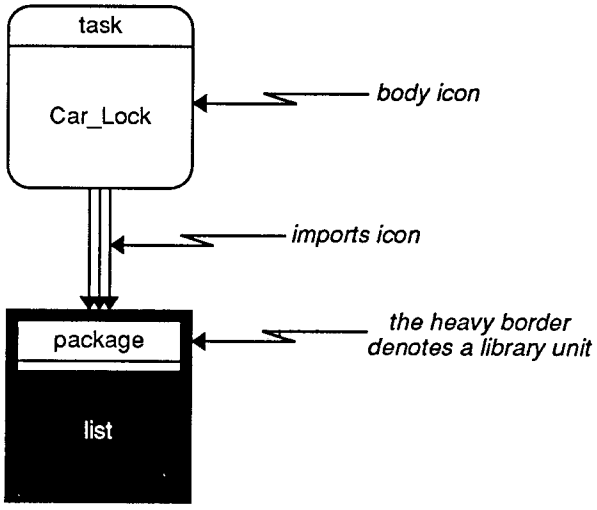
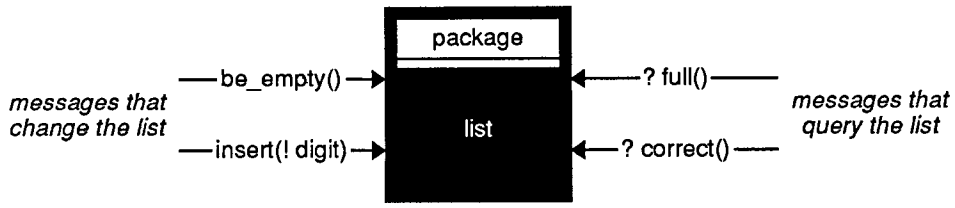


Figure 2. Car Lock Library



list behavior for combination = (a, b, c, d)			
Precondition	Stimulus	Response	Postcondition
(x, y, ...)	be_empty call	----------	()
(a)	full call	return false	(a)
(a, b)	insert(c) call	----------	(a, b, c)
(a, b, c)	full call	return false	(a, b, c)
(a, b, c)	insert(d)	----------	(a, b, c, d)
(a, b, c, d)	correct call	return true	(a, b, c, d)
(a, b, c, e)	correct call	return false	(a, b, c, e)
(a, b, c, e)	be_empty call	----------	()

```
package list is  -- an abstract object; it encapsulates the list and the combination
   subtype Digit_Type is Character range '1'..'9';
   subtype List_Length is Positive range 2..Positive'Last;
   procedure be_empty;                      -- makes the list empty
   procedure insert(digit : in Digit_Type); -- inserts a digit into the next position in the list
   function full return Boolean;            -- returns true if the list is full, else returns false
   function correct return Boolean;         -- returns true if the list = the combination
end;
```

Figure 3. List Black Box

```
package body list is
   n : constant List_Length := 4;
   type List_Type is array(1..n) of Digit;
   the_list : List_Type;
   the_combination : constant List_Type := ('1', '3', '6', '9');
   index : Natural range 0..n := 0;

   procedure be_empty is
   begin
      index := 0;
   end be_empty;

   procedure insert(digit : in Digit_Type) is
   begin
      index := index + 1;
      the_list(index) := digit;
   end insert;

   function full return Boolean is
   begin
      return index = n;
   end full;

   function correct return Boolean is
   begin
      return index = n and then the_list = the_combination;
   end correct;
end list;
```

Figure 4. List Clear Box

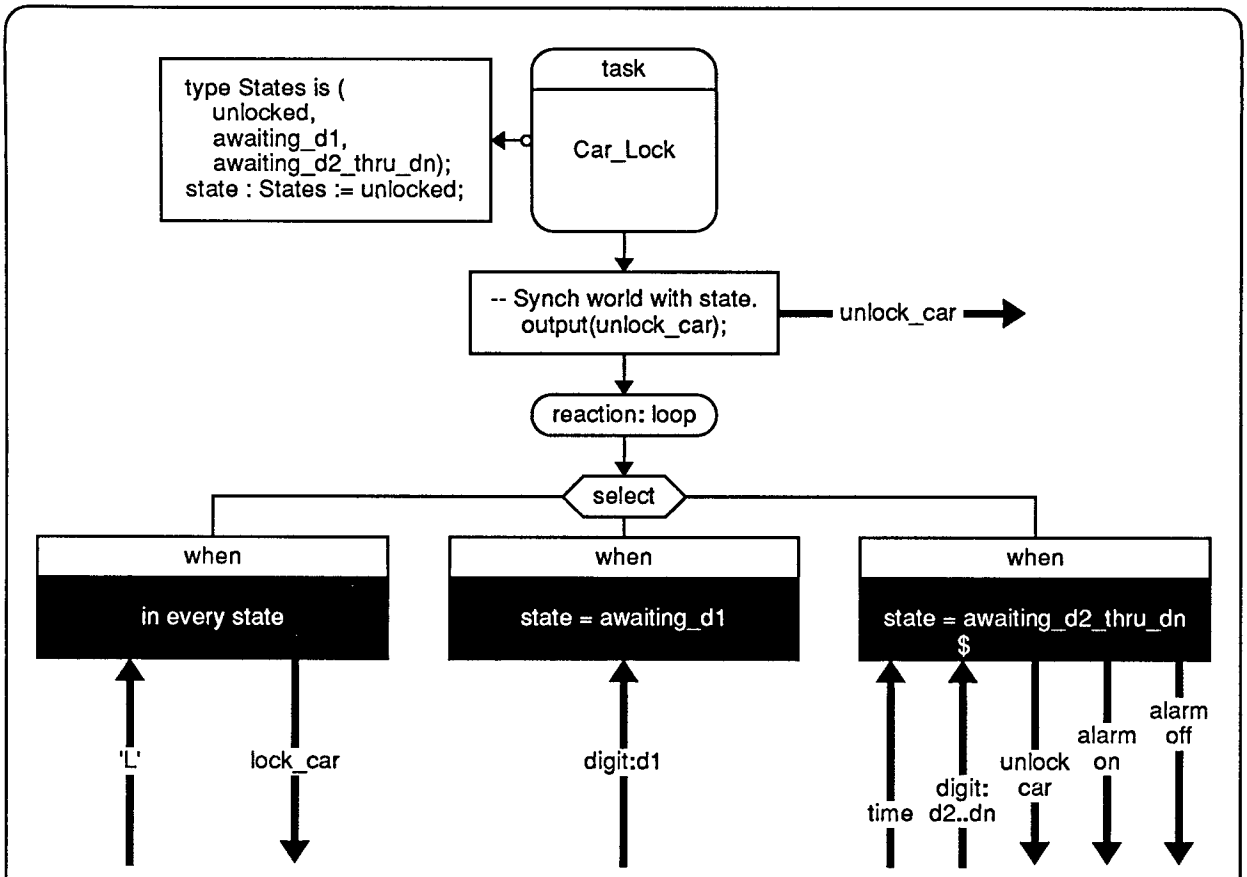


Figure 5. Car Lock Clear Box

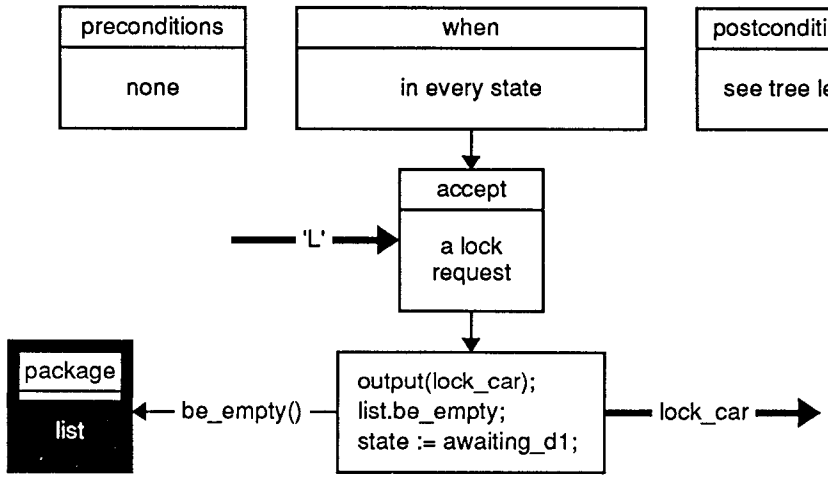


Figure 6. Clear Box for accepting 'L', a lock request

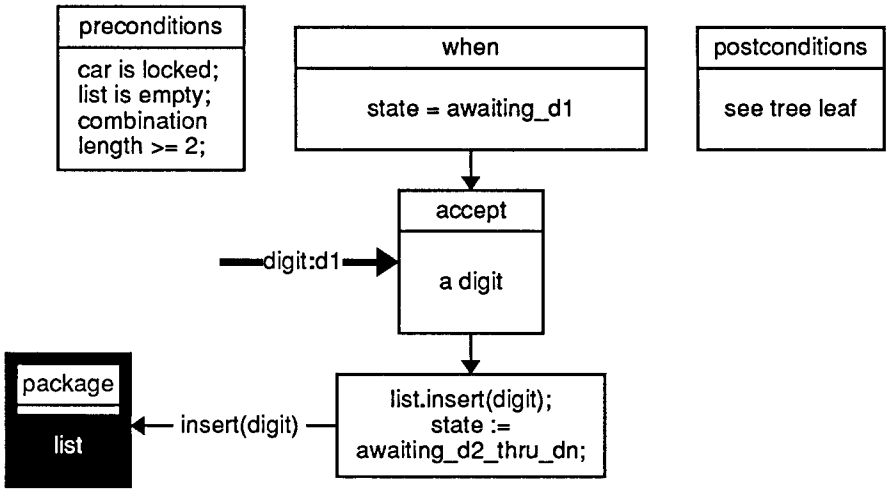


Figure 7. Awaiting_d1 Clear Box

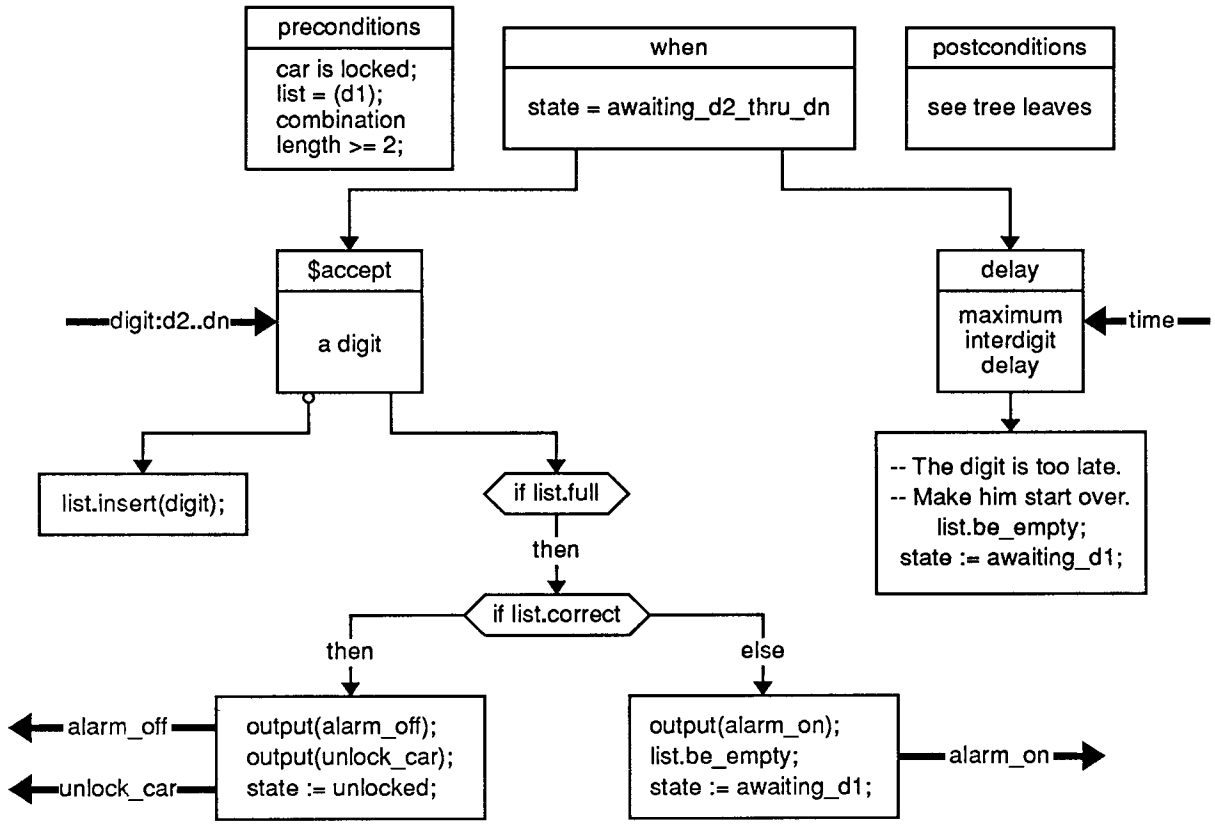


Figure 8. Awaiting_d2_thru_dn Clear Box

Proof Table for the Last Trace on Car Lock Black Box (Figure 1)
(Notes: ¬ means 1st column of next row; '9'* means '9' arrived too late.)

State	Figure	Stimulus	New List Value	Response	Next
initialization	5	none	don't care	unlock_car	¬
unlocked	6	'L'	()	lock_car	¬
awaiting_d1	7	'1'	('1')	no_response	¬
awaiting_d2_thru_dn	8	'3'	('1', '3')	no_response	¬
awaiting_d2_thru_dn	8	'6'	('1', '3', '6')	no_response	¬
awaiting_d2_thru_dn	8	time_out	()	no_response	¬
awaiting_d1	7	'9'*	('9')	no_response	¬
awaiting_d2_thru_dn	8	'1'	('9', '1')	no_response	¬
awaiting_d2_thru_dn	8	'3'	('9', '1', '3')	no_response	¬
awaiting_d2_thru_dn	8	'6'	('9', '1', '3', '6'), ()	alarm_on	¬
awaiting_d1	7	'1'	('1')	no_response	¬
awaiting_d2_thru_dn	8	'3'	('1', '3')	no_response	¬
awaiting_d2_thru_dn	8	'6'	('1', '3', '6')	no_response	¬
awaiting_d2_thru_dn	8	'9'	('1', '3', '6', '9')	alarm_off, unlock_car	

duce an executable specification of the system (j = 1), using the notation illustrated in this paper. Instead of giving the customer a document stamped SRS and saying, "This is what we're going to do." it gives him the 1st version of the system. and says, "This is what we've done so far; what else do you want us to do?"

Of course, software bureaucrats will howl at the idea that developers use the same notation (and even the same teams) across the life cycle. Specifying in one notation, designing in another notation, and implementing in another notation is deemed to be a GOOD THING. This Balkanization and Babelization of the life cycle has created many fiefdoms and factions. They serve many selfish and entrenched interests, and these interests can and will marshall intellectual arguments why things should continue to be as they have been. But we must keep pointing out the results they achieve, rubbing their noses in these results whenever we can.

Summary

Figures 1-8 are an SRS, a design, and an implementation, depending on what use you want to make of them. We believe it's sound and efficient to make multiple uses of them, a kind of reuse we don't hear enough about These Figures and especially Figures 5-8 can serve as an SRS because they clearly, completely, and unambiguously describe the system's expected external behavior, and end-users familiar with the problem domain can understand the pictures (easier, for example, than they can understand Statecharts). Or, at the other end of the spectrum, these Figures can serve as an implementation because–after their mechanical translation into code (see the Appendix)–they're compilable, linkable, and executable. (Reference [1] contains additional arguments for using the same semantics and representations across the life cycle.)

We have described a software development process that doesn't Balkanize the life cycle. We use the same notation across the life cycle; there is no semantic gulf, no changes in notation or culture. Our notation is formal and executable and lends itself readily to the precondition-postcondition kind of contract and to proof tables. Our S-R Machines unify classic and modern software engineering entities (Automata, Data Abstractions, Objects), and give a mathematical answer to the hot question of "What is an Object?". And, finally, prepared end-users and contract monitors readily understand our process and its products.

References

1. Cherry, George. W. *Software Construction by Object-Oriented Pictures: Specifying Reactive and Interactive Systems.* Thought**Tools, Canadaigua, N.Y. 1990. [This book describes the *SCOOP-3™* method, including S-R Machines (also called Abstract State Machines). It is available from Dorset House Books, 1-800-342-6657.]

   ™SCOOP-3 is a trademark of Thought**Tools, Inc.

2. Cherry, George. W. "S-R Machines: A Visual Formalism for Reactive and Interactive Systems", *Software Engineering Notes,* July 1991, 52-55. [This paper shows how to reduce the blobs and arrows on visual representations of Finite State Machines by orders or magnitude.]

3. Observation by Dr. Parnas at the 1989 Workshop on *Formal Methods for Trustworthy Computer Systems (FM89)* Springer-Verlag, New York.

4. Harel, D. "On Visual Formalisms", *Communications of the ACM,* May, 1988, 514-530. [Harel's criticisms of State Transition Diagrams led to his Statecharts and our S-R Machines.]

Appendix: Ada Source Code for Car Lock

The following Ada source code was mechanically generated from Figures 1-8. The semantically identical Concurrent C++ code is syntactically similar.

```
-- For the Car Lock task specification, see Figure 1.
-- For the package list's specification and body, see Figures 3 and 4

with list; -- From Figure 2.
task body Car_Lock is -- from Figure 5
   type States is (
      unlocked,
      awaiting_d1,
      awaiting_d2_thru_dn);
   state : States := unlocked;
begin
   output(unlock_car);
   reaction: loop
      select
         -- no guard = in every state
         accept a_lock_request do -- from Figure 6
            output(lock_car);
            list.be_empty;
            state := awaiting_d1;
         end;
      or
         when state = awaiting_d1 =>
         accept a_digit(digit : in Digit_Type) do -- from Figure 7
            list.insert(digit);
            state := awaiting_d2_thru_dn;
         end;
      or
         when awaiting_d2_thru_dn =>
         accept a_digit(digit : in Digit_Type) do -- from Figure 8
            list.insert(digit);
            if list.full then
               if list.correct then
                  output(alarm_off);
                  output(unlock_car);
                  state := unlocked;
               else
                  output(alarm_on);
                  list.be_empty;
                  state := awaiting_d1;
               end if;
            end if;
         end;
      or
         when awaiting_d2_thru_dn =>
         delay maximum_interdigit_delay;  -- from Figure 8
         list.be_empty;
         state := awaiting_d1;
      end select;
   end loop reaction;
end Car_Lock;
```