A Structural Operational Semantics for an Edison-like Language (1)

Li Wei (李未)
(Beijing Institute of Aeronautics)

Received June 7, 1983.

Abstract

A structural operational semantics for Edison 1—an Edison-like language—is given. The static and dynamic (operational) semantics for various declarations and statements contained in this type of languages have been carefully studied by using a structural operational approach. The method used here can be generalised to cover more complicated concurrent programming languages.

The paper is divided into two parts. In the first part, an abstract syntax of Edison 1 is introduced and the static semantics of it is studied. In the second part, an operational (dynamic) semantics of Edison 1 is given.

The programming language, Edison, was invented by Brinch-Hansen[1] teaching the principles of concurrent programming and constructing reliable programs for multiprocessor systems. Edison was born after careful consideration of both the successful and ill-fated experiences in application and design of concurrent programming languages, especially Concurrent Pascal and Modula. The concepts of modularity, concurrency and synchronisation are separated in Edison by introducing (respectively) modules and procedures, concurrent statements, and when statements. This measure makes Edison simpler, more general and more flexible than Concurrent Pascal and Modula.

Communication in Edison is based on management of mutually exclusive access to shared variables (common data). This idea was developed independently by Brinch-Hansen[10] and Hoare[13] as the monitor mechanism, however Edison is of the highest level design in this direction. To show the elegance of Edison let us consider a typical example—a one-character buffer. In Concurrent Pascal (or Modula) this can be programmed by using a monitor mechanism.

As we know, a monitor in Concurrent Pascal or Modula is an intricate combination of shared variables, queues, procedures, process scheduling and modularity. Edison can replace this complicated scheduling by a simple statement for synchronization, that is the when statement. Thus, in Edison, a one-character buffer can be simply programmed as a module:

Example 1.

module buffer
  var x:char; b: bool;
  *proc send (c: char)
  begin
    when not b do
      x := c; b := true
end
*proc receive (var c: char)
begin
  when b do
    c: = x; b: = false
  end
end
begin b: = false end

The shared variable x is a slot to store a character for exchanging, where the boolean b indicates whether the buffer is full or not. The initial value of b is false, it means that the initial state of the buffer is empty.

The procedure send delays the calling process (if necessary) until the buffer is empty, then it will put the character (actual parameter) into x and activates the first receiving process into waiting in the received queue. The received procedure is similar to the send procedure.

The* preceding each proc declaration above indicates that the associated procedures are to be exported from the module. Note that Edison does not include the monitor concept and that there are no queues or queue operations here. The execution of when statements takes place strictly one at a time. If several processes need to evaluate (re-evaluate) the guards simultaneously they will do so only one at a time. The when statement means:

1. To wait until no other process is executing the "body" of any other when statement. This is called the synchronisation phase.
2. To evaluate the boolean expression b. If its value is true then execute the body of the when statement; otherwise, execute this when statement again. This is called the critical phase.

The purpose of this paper is to study the semantics of Edison. In Section 1 the abstract syntax of an Edison-like language is given. We call it Edison.1.

In Section 2 we discuss the static semantics of declarations variables, procedures and modules. And finally, in Section 3 a structural operational semantics is given to Edison.1.

1. The Syntax of Edison.1

The abstract syntax (see [8] and [9]) of Edison.1 is parameterised in the following disjoint sets:

Var — a countably infinite set of variables, ranged over by x.
Exp — a countably infinite set of expressions, ranged over by e.
Bexp — a countably infinite set of boolean expressions, ranged over by b.
Pnm — a countably infinite set of procedure names, ranged over by P.
Mnm — a countably infinite set of module names, ranged over by M.

The syntactic categories of guarded statements, statements and declarations are defined by using a BNF-like notation as follows:

Gts — a set of Guarded statements, ranged over by GS and defined by:

GS :: = b⇒GS∪GS

Stm — a set of statements, ranged over by S and defined by:

S :: = skip | abort | x: = e | S:S | if GS fi | do GS od | P(AP) | M.P(AP) | when GS end | S || S

Dec — a set of declarations, ranged over by D and defined by:

D :: = empty | var x | proc P (FP) BS | module M (EP) BS | D:D