DONNER83
Donner, M.D.; The Design of OWL a Language for Walking; ACM SIGPLAN Notices 18, 16 (June 1983)

SYNOPSIS

This paper describes a special purpose language designed to facilitate the computerized control of a six legged walking machine.

The author wished to investigate the walking behaviour of insects which seems to be controlled by low level (local) mechanisms with only a limited amount of high level (global) input. i.e. each leg and its motions are largely independent. This approach requires loosely coupled concurrent processes and the author felt that existing concurrency control features which were designed for operating systems were not well suited to his problem since "they focus on synchronization of mutually oblivious tasks at the point of resource requests, whereas what I need is a way to permit concurrent mutually constrained processes to cooperate." Furthermore the author wanted good real time performance and verifiability and these imply a bound on the time taken by any process. He also wanted a capability like hardware interrupts, where one process can disable and terminate other processes without their cooperation. The approach taken was to modify Hoare's CSP which uses guarded commands and add features that modelled interrupts in software. OWL has the following syntax :

<process> := <sequence> | <concurrence> |
process call | primitive call |
done(Boolean) | alert(Boolean) | both(Boolean)
<sequence> := { ProcessList }
<concurrence> := [ProcessList]
<ProcessList> := process | process, processList

with informal semantics as follows

A sequence { ... } only terminates when one of its subprocesses executes done(true), otherwise each subprocess (starting with the first) is executed in turn, the successor of the last subprocess is the first subprocess.

A concurrence [...] is a list of processes which are all executed concurrently. The concurrence terminates when all its subprocesses terminate. However if one of its subprocesses executes alert(true) then all of its siblings subprocesses are terminated and only the subprocess that raised the alert continues.

A primitive call is a construct from the base language e.g. a procedure or a function. Currently the author has an implementtion using 'C' as a base language.

both(...) asserts both alert(true) and done(true) simultaneously.

The following is an example program from the walking code

```
Exist :
{
 [
  { { both(detectCollision) }, handleCollision, done(true) },
  { { both(detectTipping) }, handleTipping, done(true) },
  Walk
 ]
}
```

Activation of the Exist process activates a sequence which is executed forever (since there are no 'done's' within the sequence). Inside, the concurrence has three subprocesses which are all activated simultaneously , the first two wait for error conditions while the other (Walk) is the normal walking process. If an error occurs the Walk process is terminated by the 'alert', the error handler is called and then the subprocess terminates (with done(true)) which terminates the concurrence. The whole concurrence is then

reactivated by the outermost sequence.

The language was also designed with verification in mind. The actual walking machine weighs 1600 lbs and moves quite fast, so the author is interested in verification for pragmatic reasons. The basic assertion the author wished to show was "at no time in the execution of this program will there be two independent threads of control concurrently active that attempt to issue conflicting commands to the same actuator (leg)" although he doesn't say how he accomplished this.

COMMENTS

The paper takes a different approach to concurrency, the 'alert' feature which models interrupts cannot implemented with the usual concurrency features (monitors, semaphores, rendevous). The paper also illustrates that a viable approach to the solution of some problems is to design a language in which it is easy to describe and thus solve the problem.