School of Information

Technology and Engineering (SITE)

Department of Electrical and Computer Engineering

Faculty of Engineering

SEG-2106 - Software Construction - Winter 2008

LAB 8

Performance

Paul M. Baillot - 2273596

W.C Campbell – 50111

Lab 8: Performance evaluation

Introduction

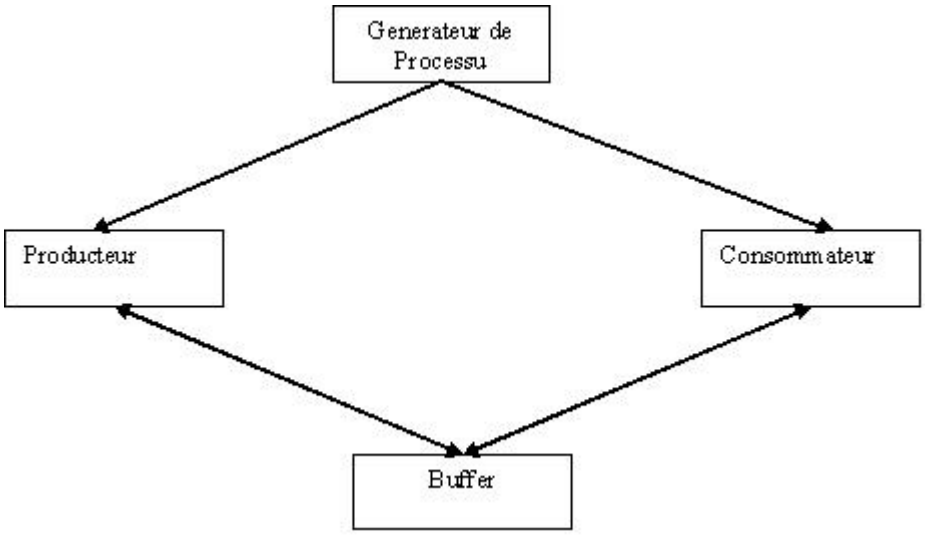
This lab deals with performance and consists of evaluating, measuring and comparing the performance of two implementations of the same general design, one implemented in Java and the other in SDL.

Objectives and Environment

Description of the application

A simple application is considered, simulating the behavior of a set of concurrent producers and consumers of messages. The producers create messages that are stored in a common buffer; the consumers take the messages from that same buffer. For simplicity, we assume that the capacity of the buffer is a single message. The following constraints must be satisfied by the application:

1. Clearly, a producer can deposit a message in the buffer only when the latter is not full; in our case, only when it is empty.
2. A consumer can read a message from the buffer when it is not empty.
3. The producers wait in line when the buffer is full (or when another producers enters a message into the buffer).
4. The consumers wait in line when the buffer is empty (or when another consumer takes a message from the buffer).
5. After a producer / consumer has finished his operation, it will wake up a consumer / producer in line (if there is one).
6. The following graph (diagram 1 – below) shows the coordination between the different actors in the system: the process generator (Générateur de **Processus**) generates producer and consumer processes. The producer and consumer processes get in line to write or read a message.



Test Environments

Two test environments are developed, implemented executed and analyzed to observe performance measurement in a buffered producer consumer environment.

First given a baseline set of Java code, a complete Java producer/consumer simulation is done, using 10 producers and 10 consumers.

Then an SDL producers/consumer environment is designed, developed, built and executed using 10 separate producer and 10 consumer processes within a single system block. Within the system block is one controlling process representing buffering, one producer block containing the producer processes and one consumer block containing the consumer processes.

Multiple runs in both environments were conducted, to observe the degree of variance from run to run, and obtain general performance assessments of the simulated runs, both in terms of number of operations over a timed period, and the variance in this performance over a repeated set of runs.

Modification made to Java Program

The modification made to the given Java program included changes to the provided consumer, consumer-generator, provider, and provider-generator function classes. A shared buffer class was provided to share the information, but required no modifications. An additional Timer class was added to control the execution of the simulation within a specified time period.

In order to make the program run, specially, the **currProducer.run ();** and a **currConsumer.run ();** start methods respectively were added to the **genProducer** and genConsumer classes. Without these methods the threads would not start up and run.

Counter variables for read and write operations were added to the Shared Buffer class to count the number of operations in the consumer and provider functions. This included method to update and read these values. Calls to update methods were included for both the consume and produce threads, just prior to the notify and release of the resource.

After time expiry and termination of the threads, in the main class, the get counter value methods obtained the number of reads and writes, and output the values, along with the time execution. These values were then assessed, with multiple runs of the simulation, to determine the correctness of a random simulation of a timed set of reads and writes.

As a well the timer function which was added to assure the consumers and provides would only run for an allotted period of time, used a Gregorian Calendar time stamp function, with an initial start time and sale and compare time functions allowing a specified run time period.

Design and implementation of the SDL version

The system design of the SDL simulation of a multiple producer consumer environment with single buffer is based on a single system block containing a concurrency control block, and a consumer block (see appendix diagrams). The Producer block contains a set of 10 identical producer processes. The Consumer block contains a set of 10 identical consumer processes. On a technical note, a synonym type of "Integer" that has a valid range of 0 to 100 was created and the ANY built-in function was used to pick random numbers.

As for the duration for the timer delays, different approaches were taken to try and randomize the duration itself however nothing ended working correctly. An attempt to provide some degree of randomness is provided by starting each of the producer (and consumers) processes with a minor incremental offsets in their start timer parameter instead.

The IO process, or the process acting as the "SimpleBuffer" in this case makes use of the "Save" SDL box. The Save box stores and queues write and read requests when the "IO" process is overwhelmed by messages. The messages that do get through have a "PId" attached to them letting the "IO" know who sent the message request and therefore who to send back the reply once the request has been fulfilled.

Simulation was conducted using the TAU/SDL simulator, which generated pseudo-random signals from the SDL model. Message sequence diagrams were observed. It was difficult to capture these in a reasonable printable form, due to form factor fitting of output to printer page (problems summary below).

Performance Issues

Results of the performance measurements and discussion

The java program on several runs had a minor variance on read/write operations over a present 1000 milliseconds timed run. Read/wires a varied from the same number to one more or less than the other. This indicates that this simulation appeared to express somewhat realistic simulation of the operations. This is different as compared to the SDL simulation, which appeared to have a fairly deterministic repeatable outcome for each run. This tends to favors the java program as a possible better simulation.

Problems encountered and lessons learned

The Java implementation was relatively straightforward. The only issue was a determination of timer methods. This was assisted thought advice from the Teaching-Assistant for a baseline code design. This general design class was adapted to the final design provided here. There was the issue of a straightforward termination of the threads. This implementation utilized deprecated stop methods that seemed to do the job.

For the SDL simulation, there existed the usual problems with TAU SDL product, and configuration setup issues, and path variables. There were multiple design options including the use of one common system block and an array of processes, or the set of a repetitive set of processes. The later option was chosen, that involved more blocks and processes, but simplified the over design structure.

Attempts to provide some degree of randomness, caused errors in development. Therefore a simpler random, but deterministic approach was therefore used. This provided difficulty in a true assessment of a random concurrent set of activities.

For true simulations, normally a large number of runs are made, with a random generation of process, and a statistical average is taken. This is possible with the Java simulation, but not the SDL one.

As noted, trying to capture a MSC output, with a large number of concurrent processes, and fit it onto a printable page was very awkward. A better scaling of output needs to be provided in the TAU/SDL product.

Conclusions

There is a tradeoff in selection either one of these software tools for simulation.
For design mock-up and functional testing, and error testing, SDL is a good choice over Java. It is much easer to conceptualize a model with its blocks, processes and signals, than coding purely in a java or other programming mode. This is a limitation in other simulation environments that use procedural coding as well, such as Opnet that requires code fudging (C code), and can cause a loss of the overall problem visibility.
However, for true statistical and real time assessments of the system under simulation, SDL appears to be inferior to the Java simulation.
Our recommendations therefore would be to do a system design mock-up, and functional verification in SDL, and then assess real-time performance with a follow up Java model. Thus both approaches used in combination provide the best develop, test and assess analysis.

Appendix - SDL Design Blocks and Processes

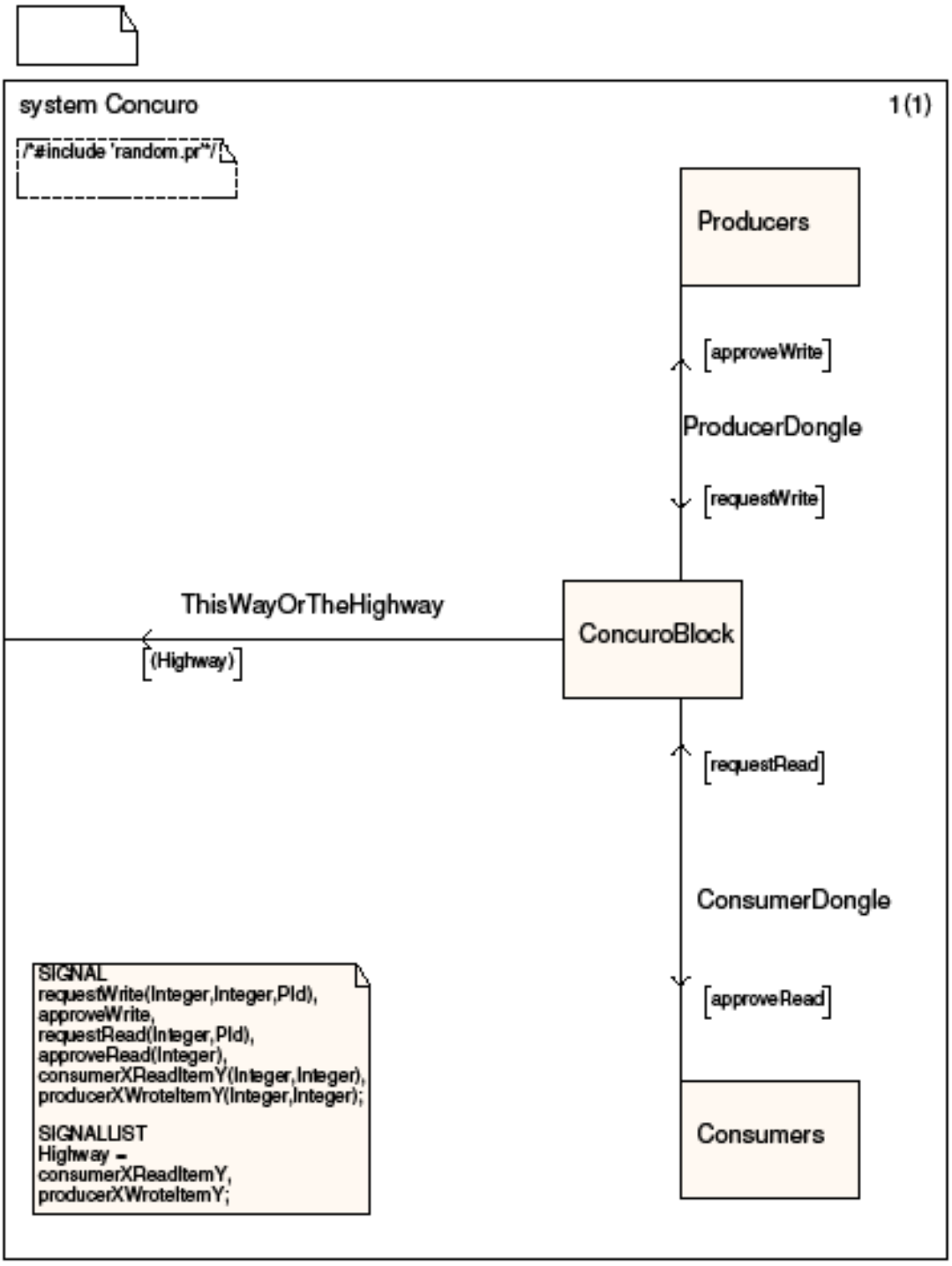


Diagram 2 - SDL System Block

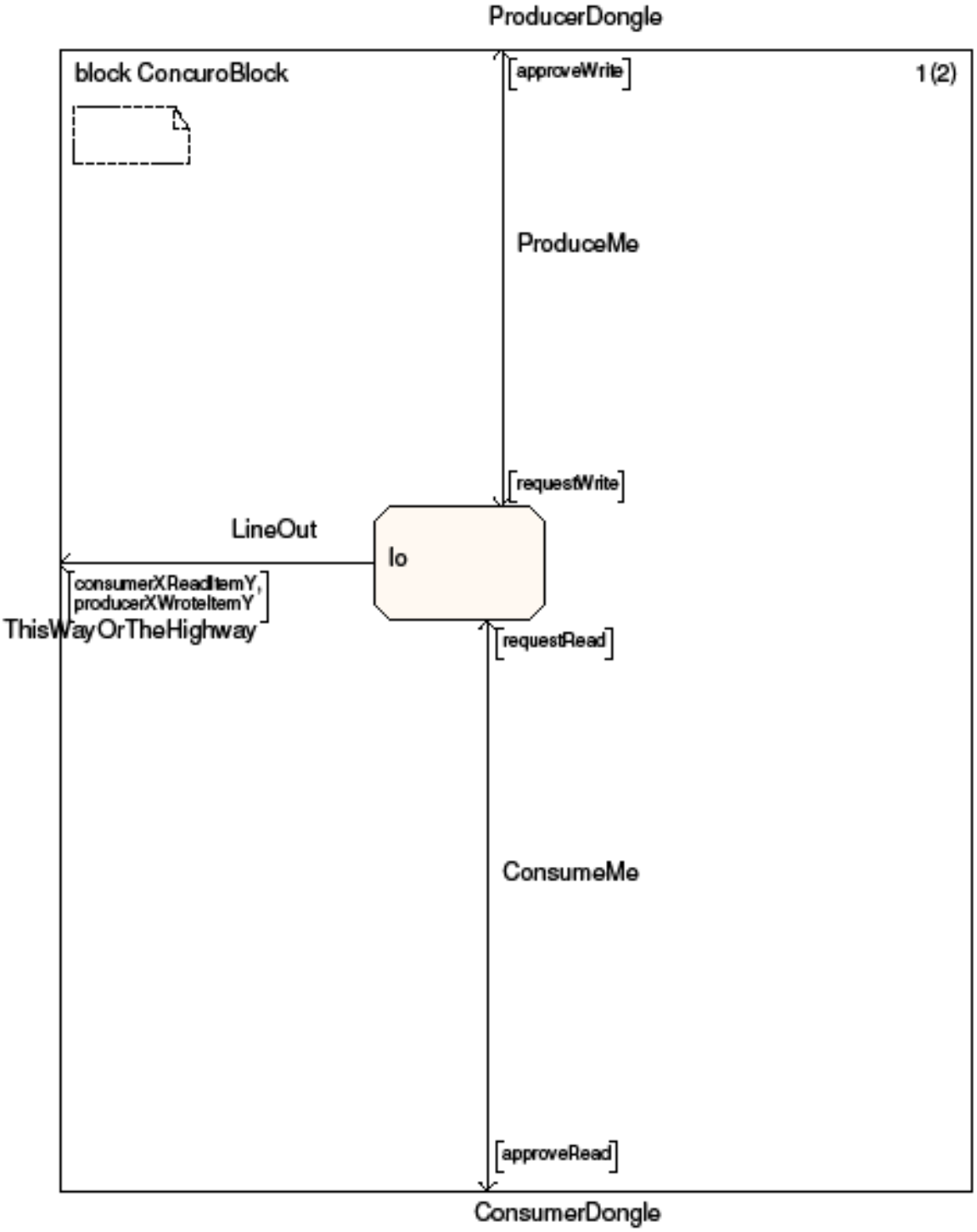


Diagram 3 - SDL Concuro

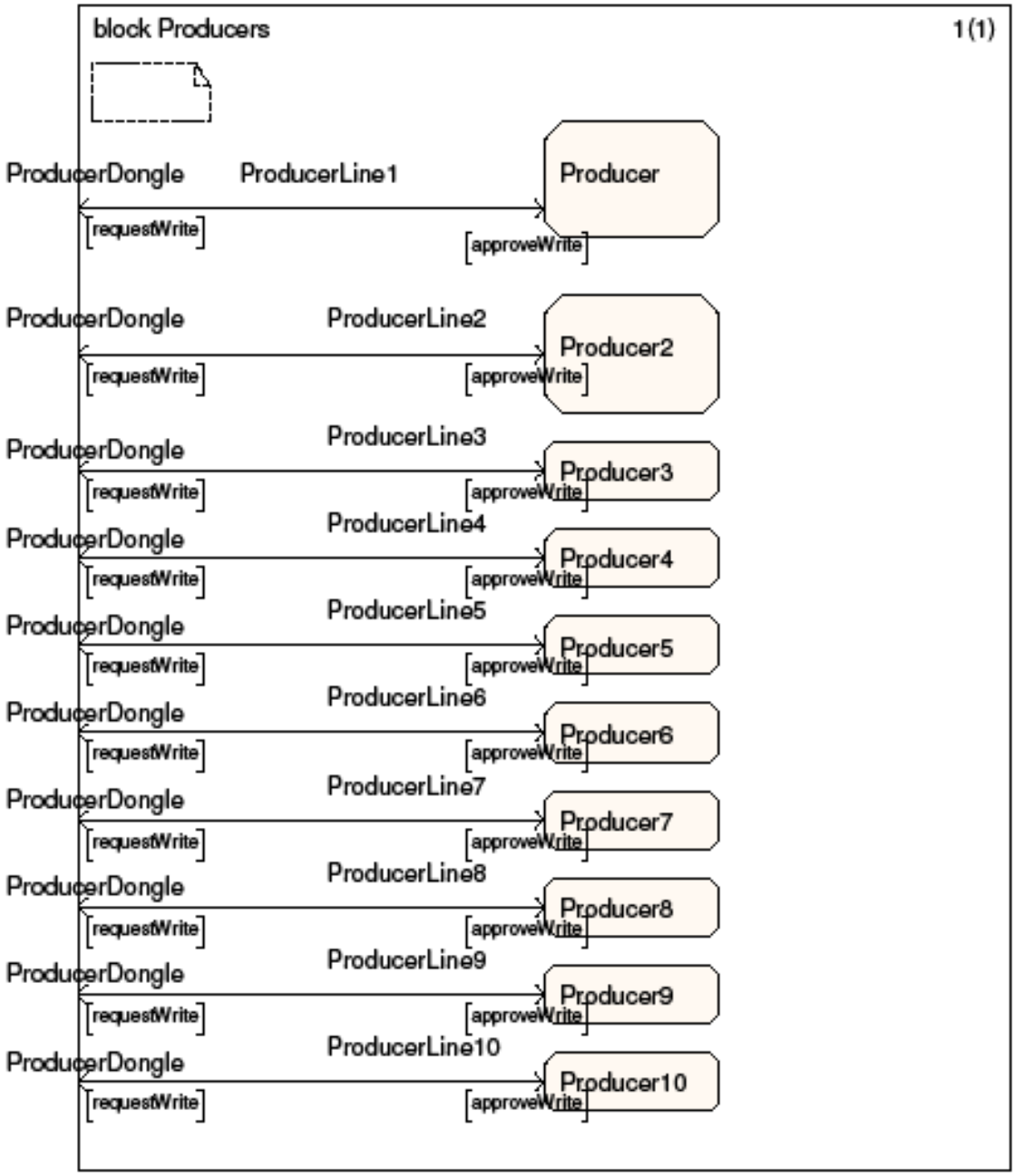


Diagram 4 - SDL Producer Block
With 10 Identical Producer Processes

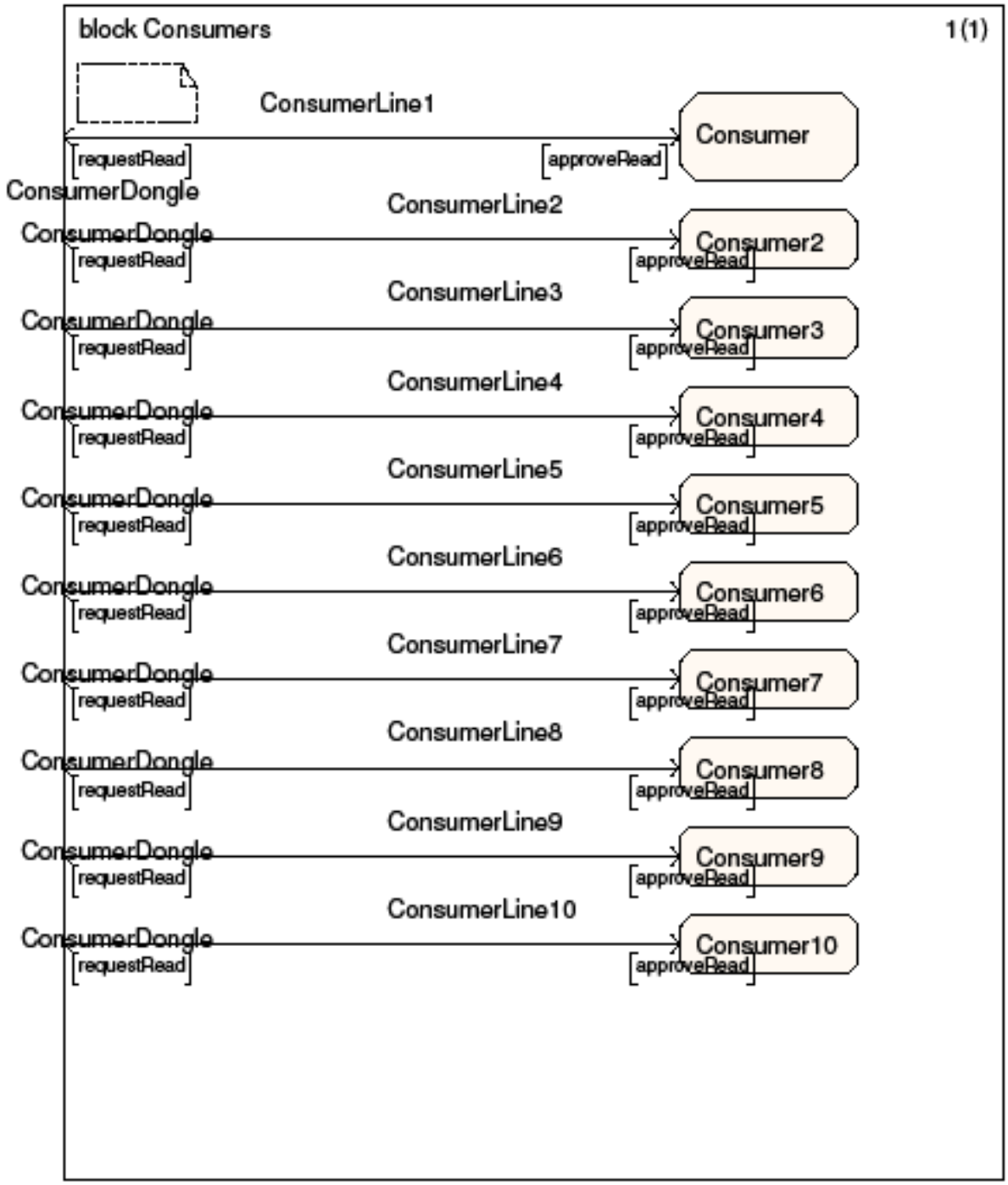


Diagram 5 - SDL Consumer Block
With 10 Identical Consumer Processes

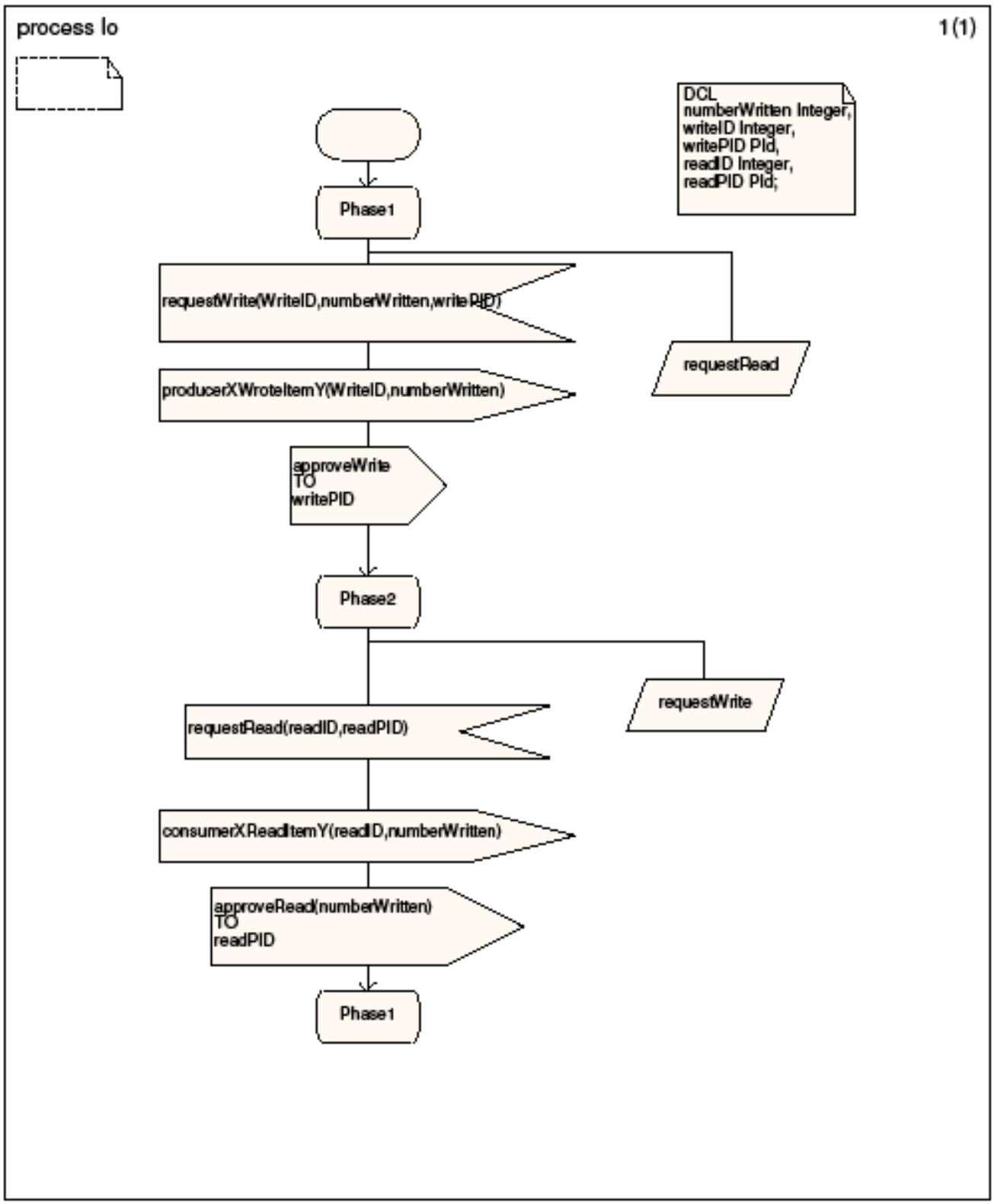
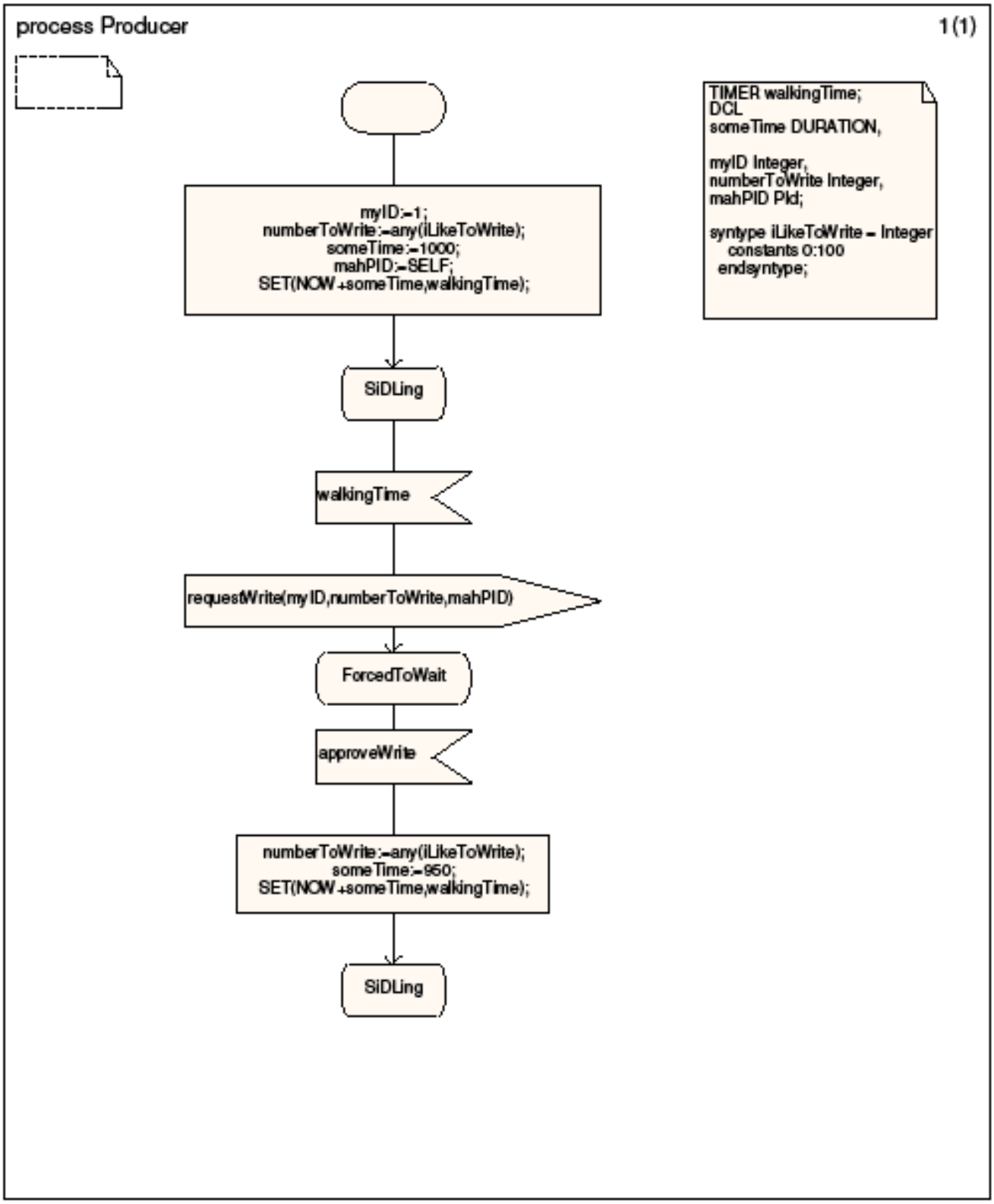
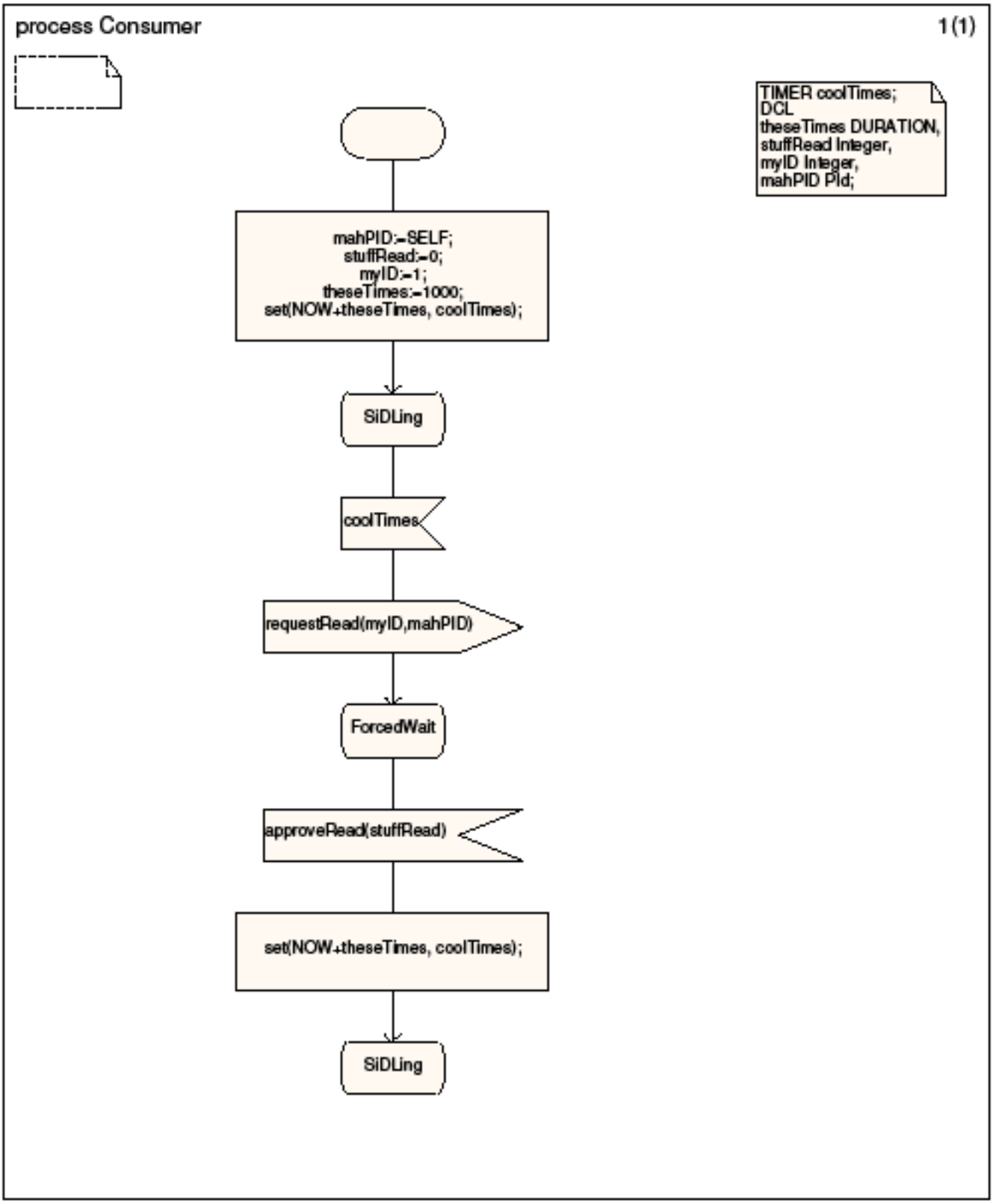


Diagram 6- SDL lo Process


process Producer
1 (1)
TIMER walkingTime;
DCL
someTime DURATION,
myID Integer,
numberToWrite Integer,
mahPID Pid;
syntype iLikeToWrite = Integer
constants 0:100
endsyntype;
myID:=1;
numberToWrite:=any(iLikeToWrite);
someTime:=1000;
mahPID:=SELF;
SET(NOW+someTime,walkingTime);
SiDLing
walkingTime
requestWrite(myID,numberToWrite,mahPID)
ForcedToWait
approveWrite
numberToWrite:=any(iLikeToWrite);
someTime:=950;
SET(NOW+someTime,walkingTime);
SiDLing


process Consumer
1 (1)
TIMER coolTimes;
DCL
theseTimes DURATION,
stuffRead Integer,
myID Integer,
mahPID Pid;
mahPID:=SELF;
stuffRead:=0;
myID:=1;
theseTimes:=1000;
set(NOW+theseTimes, coolTimes);
SiDLing
coolTimes
requestRead(myID,mahPID)
ForcedWait
approveRead(stuffRead)
set(NOW+theseTimes, coolTimes);
SiDLing