

2G1516 Formal Methods

CCS Assignment*

Autumn 2005

This document describes a series of tasks on the Concurrency Workbench, as an obligatory part of the course 2G1516. The purpose is to demonstrate how behaviours can be modelled in a process calculus, and to explore relatively simple ways of automatic verification.

In the following we assume the reader to be familiar with the basics of CCS from the lecture slides, the course book and, optionally, the papers *Process Algebra* by Cleaveland and Smolka, and *The Concurrency Workbench: A Semantics-Based Verification Tool for Finite-State Systems* by Cleaveland, Parrow, and Steffen. You also need to have access to the concurrency workbench as well as the paper “Verifying a CSMA/CD-protocol with CCS” by Joachim Parrow.

1 Requirements

The assignment should be done individually or in groups of two. Each group should present their solution at a work-station and at the same time submit one report documenting their model and results, as for previous labs. Don't overdo the reports. It is sufficient that you document your solutions. Time slots for lab reporting will be allocated at lecture 12, 6th Dec. 2005.

2 Getting acquainted with the Workbench

Read through the manual so that you know roughly what is in it. You can safely skip the material on TCCS and SCCS. The most useful commands here are probably:

- Online Help (page 16).
- Environment commands (pages 16–17): `agent`, `print`, `clear`.
- File handling commands (page 18): `input`, `output`
- Derivative commands (pages 19–22): `tr`, `vs`, `obs` etc.

*Author: Joachim Parrow, with contributions by B. Victor, D. Gurov, P. Giambiagi, M. Dam

- Relational commands (pages 22–23): `eq`, `diveq`, `mayeq`, `min`, `pb`.
- The `sim` command on page 27.
- Analysis commands (pages 20–22): `fd`, `fdobs`, `states`, `sort`, `statesexp`, `statesobs`.
- Model Checking commands (pages 26–27): `cp`, `dfweak`, `dfstrong`.

Running the Workbench

The Workbench can be started using the UNIX command `cwb`. First, in the session you may need to give the command `module add cwb` to set up the proper links.

Technical tips about the Workbench and Emacs can be found in the manual, sections 2 and 3.

3 A Simple Protocol

Read the introduction to “Verifying a CSMA/CD-protocol with CCS”. The first tasks are to model and verify a simple data link protocol, where some unrealistic assumptions will be made in order to keep the tasks simple.

The Protocol Specification is as follows: there is a *sender*, a *receiver*, and a *medium*. The sender transmits messages (without any sequence numbers) to the medium. The medium can, after reception of a message from the sender, do one of two things: either the message is delivered to the receiver, or it is lost. The loss of a message should be modelled with a τ -action, since the loss itself does not constitute an observable event. When a message is lost a timeout in the sender will occur — model this timeout as a communication signal sent from the medium (in case of message loss) to the sender. In case of a timeout, the sender will retransmit the message. When the receiver gets a message from the medium, the message is reported to the layer above, and then an acknowledgment is sent to the sender. Assume that the acknowledgment is sent *directly* to the sender and not through any medium.

Task 1 Model the above protocol P in the Concurrency Workbench, and draw its flowgraph. Draw the transition graphs of the sender, receiver, and medium. \square

Task 2 Define a simple service specification SS such that $SS \approx P$. Prove this using the Workbench. \square

Task 3 Redefine the protocol specification in the following way: when the receiver gets a message from the medium, it first sends the acknowledgment to the sender and then reports the message to the layer above. Is the new protocol still equivalent with SS ? If not, find a new simple service specification for it. Draw the relevant flow graphs and transition graphs. \square

Task 4 What happens if the protocol specification is redefined in the following way: acknowledgments are not sent at all from the receiver to the sender? What happens if additionally the sender never expects acknowledgments? Describe the resulting behaviour of the protocol (draw the transition graphs). Does a “sensible” service specification exist for any of these cases? A sensible specification for this protocol should at least have no deadlocks. If such service specifications exist, define them. Hint: there is a sensible service specification at least for the second case, i.e., when acknowledgements are neither sent nor expected. \square

Task 5 (To solve this task you don’t really need the Workbench, but you need to understand the theory behind it.) Consider the following execution of the protocol: the sender sends a message to the medium. The medium loses the message, and then the timeout signal is sent to the sender. The sender retransmits the message, and again the message is lost. This continues ad infinitum: the medium never delivers a message to the receiver. This is what sometimes is called a livelock or divergence: no observable action ever happens, but the execution never terminates. How can it be that in spite of this livelock, the protocol is *formally* proven “correct” using the \approx relation? \square

4 The CSMA/CD-protocol

Read the rest of “Verifying a CSMA/CD-protocol with CCS”. As a preliminary step, type in the definitions of that paper and verify the protocol with the Workbench. Note that the syntax for “actions” in the above paper is different from the action syntax used by Milner. The actions $a?$ and $a!$ in the paper corresponds to a and \bar{a} in Milner’s book.

As mentioned in the discussion, the model of the *MAC* is slightly inaccurate. In reality, the *MAC* would be two processes: one *sender* which performs actions $send?, b!, e!, c?$, and one *receiver* which performs actions $rec!, br?, er?$.

Task 6 Redefine the protocol specification by letting each *MAC* consist of two processes, one sender (S) and one receiver (R). The idea is to let S perform all actions involving message transmissions, i.e. the “horizontal” transitions in the *MAC* diagram, while R performs all actions involving message reception, i.e. the “vertical” transitions. Let S and R run independently of each other, i.e. put $MAC = S|R$. Draw the transition graphs of S and R , and the flow graph of the protocol. Use the workbench to prove that the so modified protocol no longer satisfies the service specification, and explain why it has no sensible service specification at all. \square

Task 7 (Main Task) Redefine the protocol once again so that it works, while still keeping the S and R parts of *MAC* separate. You may need to add some ports to achieve synchronisation between S and R . Such ports may be added only between the processes that were previously integrated in a single *MAC* process, or between the medium and another process. It is *not* allowed to add ports between different *MAC*s.

Also, the service specification may not be the same as the original specification (but it should be a reasonable service!). In that case, explain why (intuitively) the service specifications differ. You should of course also define the new service specification and show that it is equivalent with the new protocol. Draw the transition graphs of S , R and the medium. Draw all relevant flow graphs.

□

5 Helpful Hints

Number 8 is by far the most important, but also the most time consuming, of these tasks. It is well known that the reason why, intuitively, the CSMA/CD protocol works is that knowledge of a collision propagates instantaneously to all stations. Both S and R need to be notified by a collision (why?). Because of this you may have to redefine the medium slightly so that no collision can occur after the e -events, or modify S so that it can respond to collisions even after the e -event (why is this so?).

6 Using the Workbench

When solving task 8 you will work with agents containing hundreds of states, and will soon realise that a simple **false** in response to an equivalence test does not reveal very much about *why* two agents (which you perhaps thought were equivalent) are in fact not equivalent. One good way of finding out more about the behaviour of a complex system is to test validity of assertions about the behaviour. The Workbench supports such property proving (see section 8 and “Model checking commands” on page 20 of the manual, and section 7 of the paper “A Semantics-Based Verification Tool . . .”).

The **dfweak** command is an alternative version of the **eq** command, that returns a logic formula if the two agents are not equal, instead of simply answering **false**. The returned formula is true of the first agent (this can be checked using the **checkprop** command) but not the second.

Many errors are simply typing mistakes. If such a mistake creates a reference to an undefined identifier the Workbench will complain, but if the mistake creates an unintended port name the Workbench can of course not notify you — it can distinguish between bound and free names, but not between “intended” and “unintended” names. Therefore, it is a good idea to check the sort (command **sort**), and perhaps the free variables (command **fv**), of any complex system before analysis.

If you receive an unexpected **false** when checking equivalence, it may be a good idea to minimise (command **min**) the two agents and inspect the results manually. The result of a minimisation is, briefly put, a representation of a minimal transition graph equivalent with the original agent.

A frequent problem is that an agent contains a deadlock which you are not

aware of. To test for deadlocks, the Workbench contains the `fd` and `fdobs` commands. If you find that these commands take a *long* time to complete, you can use the following shortcut to deadlock testing: first minimise (command `min`) the agent you want to examine, and then apply the deadlock searching (or any other experiment you may want to do, such as the `vs` command) to the minimised agent. If you are only interested in existence of deadlocks, you can check the state space (command `states`) of the minimised agent: if it has a deadlock, the minimised agent will have `nil` (i.e. 0) in its state space. Notice, however, that the converse is not necessarily true: if the minimized agent has `nil` in its state space, this does not imply the existence of deadlocks in the original agent.

The simulation commands (command `sim`) can also be of use for checking whether an agent has the expected behaviour.