

Deadlocks and Safety Properties

Version 1.1

Jens Chr. Godskesen
IT University of Copenhagen

©2004, 2005 Jens Chr. Godskesen

March 1, 2005

1 Introduction

As we saw in the note about Monitors and Semaphores [3], *deadlock* is a common phenomenon when dealing with concurrency. Actually, one of the most common errors in concurrent systems are deadlocks. Deadlock is the first topic we deal with in this lecture.

The second topic is *safety properties*. They are properties stating that “something bad never happens”. Mutual exclusion is an example of a safety property (no undesired interleaving is going on) and so is absence of deadlock.

Part of this note and some of the examples in this note are inspired by [5].

The examples of models to be run in MWB in this note are presented with emphasis on readability and may thus not compile due to extra line breaks.

2 Deadlock

On the Internet you may find the following definition of deadlock

deadlock: n. 1.

[techspeak] A situation wherein two or more processes are unable to proceed because each is waiting for one of the others to do something. A common example is a program communicating to a server, which may find itself waiting for output from the server before sending anything more to it, while the server is similarly waiting for more input from the controlling program before outputting anything. (It is reported that this particular flavor of deadlock is sometimes called a starvation deadlock, though the

term starvation is more properly used for situations where a program can never run simply because it never gets high enough priority. Another common flavor is constipation, in which each process is trying to send stuff to the other but all buffers are full because nobody is reading anything.)

In terms of modeling we saw in [3] that in the very first case of the nested monitors there was a deadlock since in

```
agent Semaphore0(u,d) = u.Semaphore1<u,d>
agent Semaphore1(u,d) = u.Semaphore2<u,d> + d.Semaphore0<u,d>
agent Semaphore2(u,d) = u.Semaphore3<u,d> + d.Semaphore1<u,d>
agent Semaphore3(u,d) = d.Semaphore2<u,d>

agent Spaces(u,d) = Semaphore3(u,d)
agent Items(u,d) = Semaphore0(u,d)

agent Buf(p,g,s_d,s_u,i_d,i_u) =
  p.'s_d.'i_u.Buf<p,g,s_d,s_u,i_d,i_u>
  + g.'i_d.'s_u.Buf<p,g,s_d,s_u,i_d,i_u>

agent BB(put,get) =
  (^ s_d,s_u,i_d,i_u)( Buf<put,get,s_d,s_u,i_d,i_u>
    | Spaces<s_u,s_d> | Items<i_u,i_d> )
```

the system deadlocks if a `get`, was first provided as input. So in terms of models a deadlock is the same as stuck in a state, or in other words, a deadlocked state is one with no outgoing transitions.

2.1 Examples

Let us look at a simpler example. Let `P` be defined by

```
agent P(a,b) = a.(b.P<a,b> + a.0)
```

Issuing then in MWB the command `deadlocks P<a ,b>` results in

```
Deadlock found in 0
  reachable by 2 commitments
```

meaning that a deadlock for `P` can be found in 2 steps from its initial state in that the inactive process can be reached doing two `a`'s in a row.

If we ask `deadlocks BB<put ,get>` where `BB` is as defined above we would get as answer that a deadlock was found in one step.

Exercise Actually MWB tells there is also a deadlock reachable in 10 steps, a deadlock that we (probably?) overlooked in the corresponding Java program in the previous

lecture. What is the deadlock? Try to execute `deadlocks BB<put , get>` defined in `ErrBoundBuf` in `MWB`.

This is an example of one of the benefits of automatic model checking tools, sometimes they give you new insight of the actual system based on a thorough analysis of the model.

2.2 Deadlock due to concurrency

One thing is that a sequential process, like `P` from above, may deadlock all by itself but the most difficult deadlocks to detect occur in systems of parallel interacting processes as in `BB`.

Consider a system consisting of two users `P` and `Q` that both want to use a printer `Printer` and a scanner `Scanner`. The model may look like this

```
agent Printer(acq,rel) = acq.rel.Printer<acq,rel>
agent Scanner(acq,rel) = acq.rel.Scanner<acq,rel>
agent PS(acqP,relP,acqS,relS) =
    Printer<acqP,relP> | Scanner<acqS,relS>

agent P(acqP,relP,acqS,relS,copy) =
    'acqP.'acqS.'copy.'relP.'relS.P<acqP,relP,acqS,relS,copy>
agent Q(acqP,relP,acqS,relS,copy) =
    'acqS.'acqP.'copy.'relP.'relS.Q<acqP,relP,acqS,relS,copy>
agent PQ(acqP,relP,acqS,relS,copyP,copyQ) =
    P<acqP,relP,acqS,relS,copyP> | Q<acqP,relP,acqS,relS,copyQ>

agent Sys(cP,cQ) =
    (^ aP,rP,aS,rS)(PQ<aP,rP,aS,rS,cP,cQ> | PS<aP,rP,aS,rS>)
```

Asking `deadlocks Sys<copyP,copyQ>` tells us that the system deadlocks in two steps. The deadlock occur if first `P` gets `Printer` and then `Q` gets `Scanner` before `P` does.

Exercise Run `deadlocks Sys<copyP,copyQ>` and explain in detail from the transitions performed by the model above the deadlock reported.

3 Avoiding Deadlocks

Below we see examples of how to avoid deadlocks.

3.1 Let resources be ordered

The order in which the resources are allocated by the users in the example above is crucial for the potential of a deadlock. If for instance the resources are always taken in precisely the same order by the two users we would have no deadlock. Say,

```

agent P(acqP,relP,acqS,relS,copy) =
    'acqP.' 'acqS.' 'copy.' 'relP.' 'relS.P<acqP,relP,acqS,relS,copy>
agent Q(acqP,relP,acqS,relS,copy) =
    'acqP.' 'acqS.' 'copy.' 'relP.' 'relS.Q<acqP,relP,acqS,relS,copy>

```

Exercise Explain why there is no deadlock in the revised model. Check using MWB that there is no deadlock.

3.2 Introduce a time-out

Yet another solution may be to let the users release their obtained resource in case they do not succeed in acquiring the next resource after a certain amount of time (a time-out), say

```

agent P(acqP,relP,acqS,relS,copy) = 'acqP.GetS<acqP,relP,acqS,relS,copy>
agent GetS(acqP,relP,acqS,relS,copy) =
    'acqS.' 'copy.' 'relP.' 'relS.P<acqP,relP,acqS,relS,copy>
    + t.' 'relP.P<acqP,relP,acqS,relS,copy>

```

Exercise Explain why there is no deadlock in the revision of Sys with a time-out.

Exercise Why is it sufficient to let only one of P and Q have a time-out?

Both of these solutions of how to avoid deadlock raises however another problem; that of one of P and Q never making *progress*. The topics of progress and *starvation* are dealt with in the note [2].

4 Deadlock Conditions

There are four necessary and sufficient conditions for a deadlock to occur.

- Resources are *serially reusable*
- Processes allow *incremental acquisition*
- *No pre-emption* of resources
- There exists a *wait-for cycle* of processes

All four conditions must hold in order for a deadlock to occur. Hence, to avoid deadlocks, make sure that at least one of the conditions is not satisfied.

4.1 Serially reusable resources

Serially reusable resources are resources shared and repetitively used under mutual exclusion.

A *database table* can be a shared resources used repetitively under mutual exclusion as far as writing to the table is concerned. If for instance a database table could be accessed by several writing processes at the same time it would not cause deadlock (but how would the database content then look like?).

Clearly, monitors as well as semaphores are also a serially reusable resource.

4.2 Incremental acquisition

Incremental acquisition means that processes hold on to their granted resources while they are waiting to obtain the additional resources they desire.

Incremental acquisition is the problem we encountered in the nested monitor problem (see [3]) where the consumer holds the lock of the monitor while it is waiting for the producer to deliver a character.

One way to avoid incremental acquisition is to require that a process allocates all of its resources before it continues; or to be more precise that a process is only allowed to request (all its) resources when it has none. This is the idea of the so-called two-phase commit in database systems.

Incremental acquisition is denied by the time-out in the printer-scanner system above.

Both of these solutions clearly results in low resource utilization, and potential starvation.

4.3 No pre-emption

No pre-emption means that resources acquired by a process are only voluntarily released by it, i.e. they cannot be forcibly pre-empted.

As an example, suppose a process is waiting for yet another resource, if it then happens (say due to the operating systems) that all its allocated resources are pre-empted then deadlock cannot occur.

4.4 A Wait-for-Cycle

A Wait-for-Cycle is a *cycle* of processes where each process holds a resource its successor in the cycle is waiting for.

Exercise The printer-scanner system above has a wait-for-cycle. Letting a circle represent a process and letting a box represent a resource how will you draw the wait-for-

cycle for the printer-scanner system? To ease the reading of the cycle you may choose to write 'has a' and 'awaits' on the arcs in the cycle.

The wait-for cycle condition cannot be fulfilled if resources are always allocated in some specific order, as it was the case for the example with the scanner and the printer above. A formal argument for this goes as follows: Suppose n resources that are ordered

$$r_1 < r_2 < \dots < r_n$$

Assume a wait-for-cycle among the processes p_1, \dots, p_k such that:

$$\begin{array}{l} p_1 \text{ has } r'_1 \text{ awaits } r'_2 \quad \text{implies } r'_1 < r'_2 \\ p_2 \text{ has } r'_2 \text{ awaits } r'_3 \quad \text{implies } r'_2 < r'_3 \\ \dots \\ p_k \text{ has } r'_k \text{ awaits } r'_1 \quad \text{implies } r'_k < r'_1 \end{array}$$

It then follows that $r'_1 < r'_2 < \dots < r'_k < r'_1$ and we have a contradiction.

5 The Dining Philosophers

Perhaps one of the most famous and most well analyzed deadlock problems is the one about the *Dining Philosophers*, yet sufficiently simple it is at the same time subtle enough to be challenging.

Suppose five philosophers sitting at a circular table. Each philosopher has the same simple life cycle consisting of only two activities: they alternate between *thinking* and *eating*. On the table there is a plate in front of each philosopher and there are five forks placed such that there is one fork to the right and one fork to the left of each plate. A philosopher may pick up the fork to his left and the fork to his right, but only one at the time. At the center of the table there is a large plate of spaghetti (we assume it is never empty). A philosopher needs two forks to eat the spaghetti.

Exercise Make a drawing of the system consisting of the five philosophers, their forks, and the plates.

The problem is to develop a system where a philosopher eats only if she has two forks in the mutual exclusion hypothesis that no two philosophers may hold the same fork simultaneously.

The system consisting of the five dining philosophers, the forks, and the plates can be modeled by:

```

agent Fork(get,put) = get.put.Fork<get,put>
agent Forks(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5) =
    Fork(g1,p1) | Fork(g2,p2) | Fork(g3,p3) | Fork(g4,p4) | Fork(g5,p5)

agent Phil(getL,putL,getR,putR,eat) =
    'getL.'getR.eat.'putL.'putR.Phil<getL,putL,getR,putR,eat>
agent Phils(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5,e1,e2,e3,e4,e5) =
    Phil(g1,p1,g5,p5,e1) | Phil(g2,p2,g1,p1,e2) | Phil(g3,p3,g2,p2,e3)
    | Phil(g4,p4,g3,p3,e4) | Phil(g5,p5,g4,p4,e5)

agent Table(e1,e2,e3,e4,e5) = (^ g1,g2,g3,g4,g5,p1,p2,p3,p4,p5)
    (Forks(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5)
    | Phils(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5,e1,e2,e3,e4,e5))

```

where we have chosen not to have an explicit action for a philosopher being in the thinking mode.

It will not be an easy task to carry out a deadlock analysis manually on the example above. However, if we ask the question `deadlocks Table<e1,e2,e3,e4,e5>` in MWB we get the answers that five deadlocks were found in 140, 205, 200, 210, and 50 steps respectively. It seems that MWB (for efficiency reasons?) terminates after having found five ways of reaching deadlocked states.

A careful analysis reveals although that the five deadlocks reported occur in exactly the same state (so it is the same deadlock) where each of the five philosophers have taken the fork to their left. In principle this can be established in only 5 steps but there is no guarantee that MWB finds the smallest path to a deadlock.

Exercise Give an example of five steps in `Table(e1,e2,e3,e4,e5)` that leads to a deadlock.

Exercise According to the deadlock conditions presented in section 4 there must be a wait-for-cycle among the processes in `Table(e1,e2,e3,e4,e5)`. What is the wait-for-cycle in your solution to the previous exercise.

Suppose we didn't perform a careful deadlock analysis but went straight ahead implementing the Dining Philosophers, in that case it may be extremely difficult to find the deadlock by just running the program. Confer this weeks programming lab or

Exercise As an illustration, try to run the applet from:

<http://java.sun.com/docs/books/tutorial/essential/threads/deadlock.html>

5.1 How to avoid deadlock?

Of the four necessary and sufficient conditions for a deadlock to occur it is often convenient to try to break the condition about the existence of a wait-for-cycle.

In the case of the Dining Philosophers we may try to break the wait-for-cycle by enforcing an *asymmetry* in the philosophers behaviour. For instance we may change the definition of `Phil`s to

```
agent AsymPhil(getL,putL,getR,putR,eat) =
  'getR.'getL.eat.'putL.'putR.AsymPhil<getL,putL,getR,putR,eat>
agent Phil(getL,putL,getR,putR,eat) =
  'getL.'getR.eat.'putL.'putR.Phil<getL,putL,getR,putR,eat>
agent Phils(g1,g2,g3,g4,g5,p1,p2,p3,p4,p5,e1,e2,e3,e4,e5) =
  AsymPhil(g1,p1,g5,p5,e1) | Phil(g2,p2,g1,p1,e2) |
  Phil(g3,p3,g2,p2,e3) | Phil(g4,p4,g3,p3,e4) | Phil(g5,p5,g4,p4,e5)
```

where one of the philosophers is getting the forks in the opposite order compared to her fellow philosophers.

Exercise Why will this asymmetric model not deadlock? Or, why is the wait-for-cycle broken?

Exercise Why is it sufficient to let only one of the philosophers have asymmetric behaviour?

Notice, that the solution of introducing asymmetry in the example of the Dining Philosophers corresponds closely to the solution we imposed in the example with the users of the printer and scanner from above. There we forced `P` and `Q` to take their shared resources, the `Printer` and the `Scanner`, in the same order. Or, we could say that we imposed an ordering on the two resources.

In the solution to the Dining Philosophers we forced two of the philosophers to take their shared resource (the fork `Fork(g5,p5)`) as the first resource. This corresponds actually to imposing an ordering on the forks and requiring that the fork with the largest number (in terms of the index i in `Fork(g i ,p i)`) is taken first.

Like for the example with the scanner and the printer we may also avoid having a deadlock by breaking incremental acquisition using time out, e.g.

```
agent Phil(getL,putL,getR,putR,eat) =
  'getL.GetRPhil(getL,putL,getR,putR,eat)
agent GetRPhil(getL,putL,getR,putR,eat) =
  t.'putL.Phil<getL,putL,getR,putR,eat>
  + 'getR.eat.'putL.'putR.Phil<getL,putL,getR,putR,eat>
```

Another solution would be to let only four philosophers sit at the table simultaneously.

Exercise Why is that a solution? How to implement it?

5.2 Starvation and livelock

Any of the solutions proposed in detail above suffer from individual *starvation* of any of the philosophers, i.e. although some of the philosophers may both be eating and

thinking none of the philosophers are ever guaranteed to be eating (enter their critical section). In the worst case it may always be the same two philosophers that happens to be eating.

Exercise Why are even the asymmetric solution subject of starvation?

Another problem which occur with the solution where the time-out is used is a form of deadlock known as *livelock*, i.e. there exists executions in which no philosopher ever gets two forks (enter its critical section).

Exercise Why does the time-out solution suffer from livelock?

Exercise Why does the asymmetric solution not suffer from livelock?

We deal with avoidance of starvation and livelock in [2].

6 Safety Properties

In this section we first briefly touch upon the notion of a *property* in general, where after we focus on the specific properties called *safety properties*.

6.1 What are properties?

We may define the notion of a property of a process as follows:

A *property* for a process P is an assertion about the behaviour of P .

By the *behaviour* of a process we mean every possible execution of the process, or to be even more precise the *execution tree* of a process.

Exercise What is the execution tree of the process

$$P(a, b) = a. (b. P < a, b > + a. 0)$$

Exercise What is the execution tree of the (deadlock potential) system with the two users of a scanner and a printer?

The most common properties for *sequential programs* are typically that:

- the program terminates
- the right result is computed by the program (eg. inserting, deleting and searching correctly, sorting correctly, ...)

However, concurrent (and reactive) programs are not always supposed to terminate, so for these we may like to express other properties, for instance:

1. The buffer is not always full (is not always empty).
2. After the printer is acquired it is released later on.
3. The Dining Philosophers don't deadlock.
4. When a philosopher is thinking he eventually gets the opportunity to eat.

A property may be *valid* or *invalid* wrt. a process's execution tree. For instance,

- The assertion “ $P(a, b)$ deadlocks” is valid because $P(a, b)$ has a finite trace in its execution tree.
- The assertion “ $\text{Sys}(cP, cQ)$ doesn't deadlock” is invalid with respect to the model in Section 2.2 because it has a finite path in its execution tree of length two (when P has the printer and Q the scanner)
- The assertion that “the printer is eventually released after it is taken” is valid in the revised models of the printer and scanner system defined in Section 3.1 because in any state where the printer has just been taken there is a finite sequence of steps to a state where the printer has just been returned.

6.2 How to express properties?

Sometimes certain logics are used to express properties about processes, and CCS is equipped with a logic called Hennessy-Milner-logic (HML). It is defined by

$$\phi ::= tt \mid ff \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a] \phi$$

where $a \in \mathcal{A}$ is an action. We let Φ denote the set of all formulas defined with respect to a finite set of actions \mathcal{A} .

The first two operators are true and false respectively. The two next are the usual logical connectives of conjunction and disjunction respectively. The last two operators are referred to as *modalities*, i.e. they allow the logic formulas to assert statements about the transitions leaving a state. HML is referred to as a *modal logic*.

Intuitively, a process p satisfy $\langle a \rangle \phi$ if there exists a transition $p \xrightarrow{a} p'$ such that p' satisfy ϕ , p satisfy $[a] \phi$ if for all $p \xrightarrow{a} p'$ it holds that p' satisfy ϕ .

The semantics of HML is formally defined relative to a labeled transition system $\mathcal{L} = (\mathcal{P}, \mathcal{A}, \rightarrow)$ by a relation $\models \subseteq \mathcal{P} \times \Phi$

$p \models tt$ for any $p \in \mathcal{P}$
 $p \models ff$ for no $p \in \mathcal{P}$
 $p \models \phi \wedge \phi'$ if and only if $p \models \phi$ and $p \models \phi'$
 $p \models \phi \vee \phi'$ if and only if $p \models \phi$ or $p \models \phi'$
 $p \models \langle a \rangle \phi$ if and only if there exists $p \xrightarrow{a} p'$ such that $p' \models \phi$
 $p \models [a] \phi$ if and only if for all $p \xrightarrow{a} p'$ it holds that $p' \models \phi$

Exercise Explain why $\langle a \rangle tt$ intuitively means “can do a ” whereas $[a]ff$ intuitively means “can’t do a ”.

Exercise Why is it that p trivially satisfy $[a]\phi$ if there exists no $p \xrightarrow{a} p'$?

Exercise Explain why $\langle a_1 \rangle \dots \langle a_m \rangle tt$ characterizes processes that can perform the trace $a_1 \dots a_m$.

Exercise Explain why $\langle a_1 \rangle \dots \langle a_m \rangle ([b_1]ff \wedge \dots \wedge [b_n]ff)$ is satisfied by a process that can perform the trace $a_1 \dots a_m$ but then cannot engage in any of b_1, \dots, b_n .

Exercise Explain why $a.(b.0 + c.0)$ satisfy $\langle a \rangle \langle b \rangle tt$

Exercise Explain why $a.(b.0 + c.0)$ satisfy $[a]\langle b \rangle tt$ but $a.b.0 + a.c.0$ doesn’t.

Exercise Give a property satisfied by $a.b.0 + a.c.0$ but not by $a.(b.0 + c.0)$.

Exercise Why are tt and $[a]tt$ the same? Why are ff and $\langle a \rangle ff$ the same?

For further reading about HML see [1] and [4].

6.2.1 Tool support

We may readily check whether a CCS process satisfy a property in MWB by taking care of using the proper MWB notion for HML formulas. The general translation of an HML formula ϕ into MWB notation we denote by $\llbracket \phi \rrbracket$ and it is defined by:

$$\begin{aligned} \llbracket tt \rrbracket &= TT \\ \llbracket ff \rrbracket &= FF \\ \llbracket \phi \wedge \phi' \rrbracket &= \llbracket \phi \rrbracket \& \llbracket \phi' \rrbracket \\ \llbracket \phi \vee \phi' \rrbracket &= \llbracket \phi \rrbracket \mid \llbracket \phi' \rrbracket \\ \llbracket \langle a \rangle \phi \rrbracket &= \langle a \rangle \llbracket \phi \rrbracket \\ \llbracket [a] \phi \rrbracket &= [a] \llbracket \phi \rrbracket \end{aligned}$$

Define CCS processes

```
agent P(a,b,c) = a.b.0 + a.c.0
agent Q(a,b,c) = a.(b.0 + c.0)
```

We then may try for instance the following runs:

```
MWB>prove P<a,b,c> <a><b>TT
Model Prover says: YES!
(7 inferences)
MWB>prove Q<a,b,c> <a><b>TT
Model Prover says: YES!
(5 inferences)
MWB>prove P<a,b,c> [a]<b>TT
Model Prover says: NO.
(4 inferences)
MWB>prove Q<a,b,c> [a]<b>TT
Model Prover says: YES!
(5 inferences)
MWB>prove P<a,b,c> <a>[b]FF
Model Prover says: YES!
(4 inferences)
MWB>prove Q<a,b,c> <a>[b]FF
Model Prover says: NO.
(5 inferences)
MWB>prove P<a,b,c> <a>TT & [a](<b>TT | <c>TT)
Model Prover says: YES!
(18 inferences)
```

The information about the number of inferences has to do with how the model prover works based on what is called inference rules. We shall not pursue this further here in this note.

6.3 Recursive Formulas

The HML formulas defined above address only finite behaviour, they only state something about the behaviour until some finite level (the longest nesting of the modal operators) in the execution tree.

There seems to be a need for working with properties that not only talk about finite behaviour, how for instance to define a property describing absence of deadlock if only

finite HML formulas are allowed? Or, how to define a property describing mutual exclusion; for instance in the case with the Dining Philosophers it should not ever happen that two philosophers sitting next to each others eat at the same time.

In both cases, if the finite formula checks until say layer n then a potential failure may occur at a level after n and hence nothing confirmative as to whether the property holds or not can be ascertained. Therefore, since HML can only define properties about finite behaviour we need something extra!

In order to being able to express properties dealing with potentially infinite behaviour MWB allows for recursively defined HML formulas. We shall not study the fundamentals of recursive formulas in detail it suffice with a good intuition.

Suppose we change \mathbb{P} from above to

```
agent P(a) = a.P(a)
```

meaning that \mathbb{P} has a repetitive behaviour. It would be nice to specify a property saying that “ \mathbb{P} can repetitively do a ”.

We can do so if we allow HML to be defined also with respect to a set of variables \mathcal{X} extending the syntax with

$$\phi ::= \dots \mid X \mid \nu X. \phi$$

where $X \in \mathcal{X}$.

We are then allowed to write e.g. $\nu X. \langle a \rangle X$ which intuitively is satisfied by all the processes that can do an infinite number of a -transitions.

In MWB we may write

```
MWB>agent P(a) = a.P<a>
MWB>prove P<a> nu X.<a>X
Model Prover says: YES!
(5 inferences)
```

6.3.1 Maximal Fixed Points

In technical terms $\nu X. \langle a \rangle X$ is a *maximal fixed point* defined as the limit of

$$tt \supseteq \langle a \rangle tt \supseteq \langle a \rangle \langle a \rangle tt \supseteq \dots \langle a \rangle^k tt \supseteq \dots$$

where $\langle a \rangle^k$ denotes a sequence of k $\langle a \rangle$ -modalities and considering here a formula as the subset of processes in \mathcal{P} that satisfy it. E.g. tt denotes in that sense all processes in \mathcal{P} .

Intuitively, the set of processes satisfying $\nu X. \langle a \rangle X$ is the largest set of processes that is a solution to the equation

$$X = \langle a \rangle X$$

Clearly we should in this case not strive for the smallest set of processes satisfying the equation, because that would be the empty set. The empty set of processes is a solution because it may (following our convention from above) be denoted by ff and because ff semantically is the same as $\langle a \rangle ff$

6.4 Safety Properties

Recall that safety properties are properties stating that “something bad never happens”. This absence of badness must be checked in every state of the execution tree.

A safety property is *always* true, i.e. it holds true in every state in the execution tree.

It turns out that maximal fixed points are suitable for expressing safety properties. If ϕ is the property that must hold in every state then we may define the corresponding safety property by

$$\nu X. (\phi \wedge \bigwedge_{a \in \mathcal{A}} [a]X) \quad \textit{Safety} \quad (1)$$

Intuitively it means that ϕ must hold in the current state and no matter what the next state is ϕ must hold again. Clearly, it is crucial here that \mathcal{A} is finite because we cannot have an infinite conjunction according to the syntax of HML.

6.4.1 Examples of safety properties

Consider the deadlock free version of the example with the scanner and the printer from Section 3.1. Since the users of the printers and the scanners are supposed to copy under mutual exclusion a safety property could be defined by letting

$$\phi = [{}'cP]ff \vee [{}'cQ]ff$$

in the general formula (1).

Intuitively, ϕ means that in the current state at least one of the copying output actions ${}'cP$ and ${}'cQ$ cannot be performed, i.e. P and Q copy under mutual exclusion.

Running the prover in MWB yields:

```
MWB>prove Sys<cP,cQ>
      nu X. (([{}'cP]FF | [{}'cQ]FF) & ([t]X & [{}'cP]X & [{}'cQ]X))
Model Prover says: YES!
(122 inferences)
```

Notice, the set of actions in $\text{Sys}\langle cP, cQ \rangle$ beyond cP and cQ also includes the internal action τ .

Absence of deadlock is a safety property. The formula to be inserted into the general formula for safety properties must specify that in any state at least one transition is possible, hence

$$\phi = \bigvee_{a \in \mathcal{A}} \langle a \rangle tt$$

In MWB we get

```
MWB>prove Sys<cP,cQ> nu X. ((<'cP>TT | <'cQ>TT | <t>TT)
& ([\tau]X & ['cP]X & ['cQ]X))
Model Prover says: YES!
(173 inferences)
```

7 Exercises

1. Go through the exercises we didn't make during the lecture.
2. Make a deadlock free CCS model of the Dining Philosophers where only four philosophers may sit at the table at any one time. Hint: Use a semaphore.
3. Do the programming lab about the Dining Philosophers.
4. Write a safety property for a deadlock free model of the Dining Philosophers stating the two neighbouring philosophers eat under mutual exclusion. Prove the safety property in MWB.
5. We may choose a weaker variant of the general format of the safety property presented above; instead of requiring that any state in the execution tree satisfy ϕ it may suffice only a single path in the tree doing so. Give a HML formula stating that ϕ holds in every state along some path in the computation tree.

References

- [1] R. Cleaveland and S. Smolka. Process algebra, 1999.
- [2] J.C. Godskesen. Liveness and fairness (version 1.1), 2005. Teaching notes, IT University of Copenhagen.
- [3] J.C. Godskesen. Monitors and semaphores (version 1.1), 2005. Teaching notes, IT University of Copenhagen.
- [4] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 1985.

- [5] J. Magee and J. Kramer. *Concurrency — State Models & Java Programs*. World-wide Series in Computer Science. John Wiley & Sons, 1999.