T-79.231 Parallel and Distributed Digital Systems

Introduction

Marko Mäkelä

July 11, 2002

Autumn 2002

Lectures:	Teemu Tynjälä, 451 5138, Teemu.Tynjala@hut.fi
Exercises:	Jukka Honkola, 451 5244, Jukka.Honkola@hut.fi
Accomplishing:	Examination and obligatory home assignments (minimum amount of points; opportunity to improve the grade)
Prerequisites:	T-79.144 Logic in computer science, foundations T-79.148 Foundations of Theoretical Computer Science

(This course corresponds to T-79.179, which is lectured in Finnish.)

Teaching material

Marko Mäkelä and
Teemu Tynjälä: lecture slides (from the course home page or from Edita)
Marko Mäkelä: MARIA user manual: http://www.tcs.hut.fi/Software/maria/
David Walker: *Introduction to a Calculus of Communicating Systems*

Other reading

Tadao Murata: *Petri Nets: Properties, Analysis and Applications*
Robin Milner: *Communication and Concurrency*
Wolfgang Reisig: *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*

Announcements

- Home page http://www.tcs.hut.fi/Teaching/T-79.231/
  - weekly exercises
  - home exercises
- Discussion group nntp://news.tky.hut.fi/opinnot.tik.rhj
  - general quick announcements
  - discussing the exercises

The objective of the course

The course aims to give the necessary skills for modelling parallel and distributed systems and for formulating and verifying requirements on these systems.

- Modelling: Petri nets, transition systems and process algebra
- Formulating properties: temporal logic
- Verifying properties: reachability analysis, using tools

Related courses

T-79.186	Reactive systems
T-79.190	Testing of Concurrent Systems
T-79.193	Formal Description Techniques for Concurrent Systems
T-106.530	Embedded Systems
T-106.520	Distributed Systems
T-79.146	Logic in Computer Science: Special Topics I
T-79.185	Verification

Parallel and distributed systems: what are they?

- parallelism: events occur simultaneously (concurrency), and there are alternative execution orders (nondeterminism)

- distribution: the control of the system is not fully centralised

  > A distributed system is one on which I cannot get any work done, because a machine I have never heard of has crashed. *L. Lamport*

- asynchrony: the system components are only synchronised via message passing

- reactivity: the system operates continuously, reacting on external events

Parallel and distributed systems: designing

Systems with inherent parallelism often are so complex that they simply cannot be built with the lazy programmer's trial-and-error method. Even if the basic idea worked, special cases can cause problems.

- nondeterminism (a large number of alternatives):
  - local branching: arbitrary input and a large `switch` block with lots of `case`s
  - branching in time: an event may arrive at very many different points
- dependencies between the current and future components of the system:
  - the control logic for the distributed functions easily becomes scattered all around the program code
  - it is easiest to describe future extensions in an abstract model

Parallel and distributed systems: what for?

- speeding up: it is cheaper to obtain many slow processors than a super-fast one, and the speed of the fastest available processor may be inadequate for the application
- redundancy: the system remains operational during maintenance breaks and partial hardware failures
- modularity: it may be easier to manage several specialised processes than a single complex process that takes care of everything
- geographical distribution: preparing for large accidents and acts of war

Properties of systems

- safety: “the system never reaches a bad state”; in each state holds P
  - deadlock freedom
  - mutual exclusion etc.
- liveness: “there is progress in the system”; X occurs infinitely often
- fairness; once X has occurred, Y will occur in n steps
  - sent messages are eventually received
  - each request is served
- self-stabilisation: “the system recovers from a failure in a finite number of steps”

Methods

- Petri nets: place/transition nets and high-level nets
- Temporal logic
- MARIA and algebraic system nets
- Transition systems and process algebra
- Basics of state space reduction methods: partial order reductions and equivalences

Applications for the methods

- communication
  - verifying and testing communication protocols
  - evaluating the performance (queueing times, throughput, . . .)
- safety critical embedded systems
  - railroad interlocking
  - aircraft and air traffic control systems
- hardware design (system on chip): locally synchronous, globally asynchronous
  - processors, peripheral interfaces, memory caches and buses
  - dividing tasks between programmable logic and micro-controller firmware
- all kinds of reactive systems

Problems in software production

- How to design and implement a piece of software in such a way that
  - it can be delivered in time according to the order,
  - the costs remain within predetermined bounds,
  - the customer obtains a solution corresponding to its expectations, and
  - it is easy to extend the software?
- Software production is mostly manual work.
- Formal methods offer solutions to some of these problems.

Software crisis

The production and maintenance costs of software grow, as the software becomes increasingly more complex. The later an error is detected, the more expensive it is to correct the error. A larger number of code lines will need to be revised—or in the worst case—installations at customer sites must be replaced. Also fees for breaching contracts or the loss of reputation cost.

- telephone system: switching networks crash, feature interactions, billing problems
- errors in space probes: Mariner I, Ariane 5, ...
- errors in Pentium processors: `fpdiv`, `f0 0f`, ...
- the maintenance of badly programmed WWW servers binds an inconceivably large amount of resources
- other examples of errors: `nntp:comp.risks`

Specifying systems

- A system specification captures the assumptions and requirements of the operations.
- Specifications should be unambiguous but not necessarily complete.
- Specifications describe the allowed computations (or executions).
- A specification is a *contract* between the customer and the software supplier.
- Usually, specifications are formal descriptions of systems.

Advantages of formal description techniques

- correctness: automated verification that the system fulfils its requirements
- completeness: the specification forms a checking list
- consistency: inconsistent or unreasonable requirements can be detected from specifications in an early phase
- reuse: abstract specifications are highly independent of software and hardware environments, and the same solutions work in different projects (“design patterns”)

Many software description techniques, such as UML, often aren’t formal enough.

Marko Mäkelä

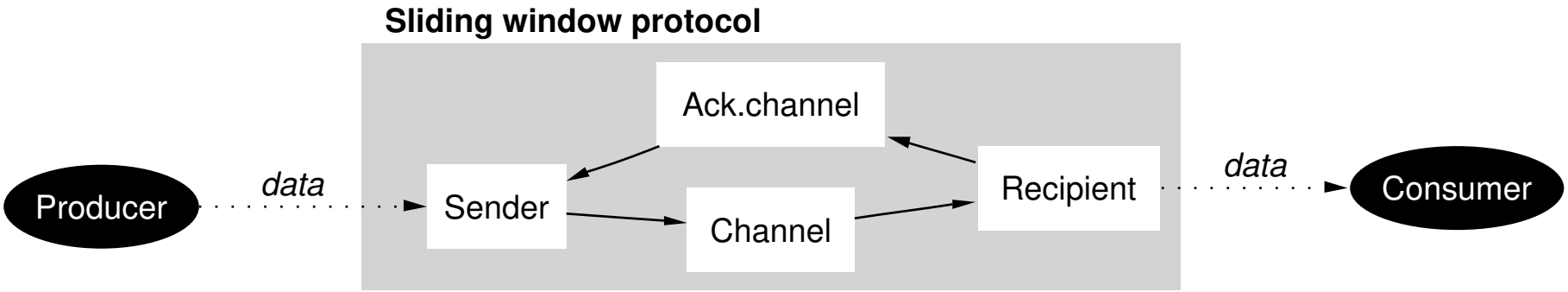
Specification in software production

- formalising the requirements: clarifying the customer needs
- planning: partitioning the system, composition and refinement of the components
- verification: does the formal model of the system fulfil the desired properties
- validation: does the system implementation correspond to the test cases derived from the model
- documenting the implementation
- analysis and evaluation: reverse engineering an existing system to understand and to develop it further

Example: reliable connection on a lossy channel

Many data communication systems are based on an unreliable connection that may distort, lose or duplicate messages. Distorted messages can be discarded by using checksums, and lost and duplicated messages can be detected by numbering the messages.

The basic solution, sliding window protocol, assigns sequence numbers to the messages of transit in such a way that the recipient can detect a lost message and ask the sender to repeat previous messages. Whenever the recipient has obtained a contiguously numbered sequence of messages, it can relay them to the consumer.



Example: Alternating bit protocol (1/6)

The sliding window protocol is a generalisation of the alternating bit protocol published in 1969. In the alternating bit protocol, there are two sequence numbers for messages: the states 0 and 1 of the alternating bit. Both the sender and the recipient have their own copy of the alternating bit. The sender holds it in the variable s, the recipient in r.

- sender: send a message tagged with s
  - if no acknowledgement tagged with s arrives in due course, repeat the message
  - if the recipient acknowledges with s, toggle s: $s \leftarrow 1 - s$.
- recipient: receive a message tagged with b—the state of s at sending time—and acknowledge it with b
  - relay the message and toggle $r \leftarrow 1 - r$, if $r = b$
  - otherwise discard the message

Our claim is that the protocol recovers from the situation that the channel loses messages or acknowledgements a finite number of times.

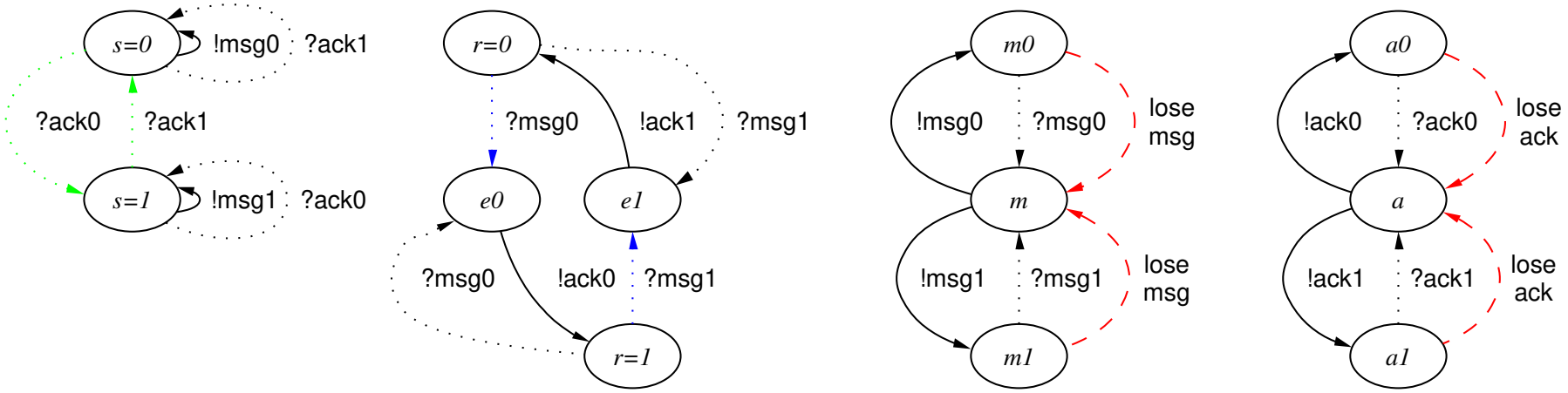
Example: Alternating bit protocol (2/6): C program

The “send” and “receive” primitives can be implemented with two subroutines, which in turn use the channel primitives “send,” “receive” and “receive_timeout.”

```
void
abp_send (char c)
{
   static char b;
   do
      send (data, b, c);
   while (receive_timeout (ack) != b);
   b = !b;
}
```

```
char
abp_receive (void)
{
   static char b_expect;
   char b, c;
   do
      b = receive (data, &c),
      send (ack, b);
   while (b != b_expect);
   b_expect = !b_expect;
   return c;
}
```

Example: Alternating bit protocol (3/6)



The figure depicts the operation of the sender, the recipient and the channels as *labelled transition systems*, or communicating finite state automata. Transmissions and receptions of messages are marked with exclamation points and question marks. Actions carrying the same name are carried out simultaneously in each automaton. The channels have capacity for at most one message at a time.

For the sake of clarity, the automata for the producer and the consumer are omitted.

Example: Alternating bit protocol (4/6): state space

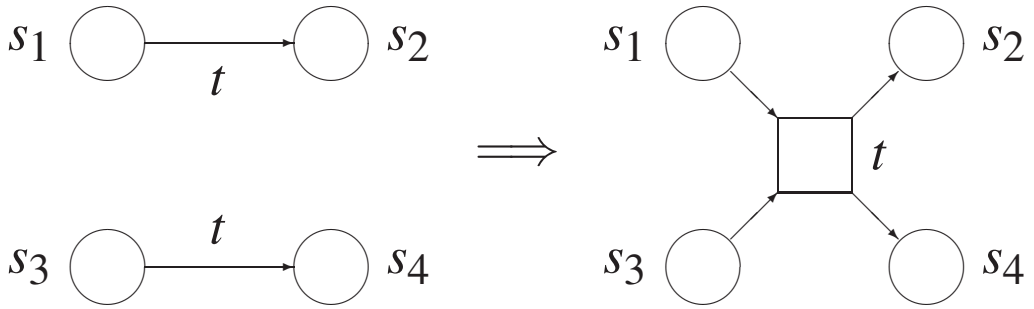
Even though the data transmitted by the protocol may be the main concern of end users, the data payload of the messages is irrelevant for observing the operation of the protocol. The less memory a model contains, the easier it can be verified, because a system with b bits of memory can assume at most 2^b states.

The timers triggering retransmissions have been abstracted away. In a fully asynchronous system, enabled timers are assumed to be able to expire at any moment. The occurrences of timer expirations could be restricted by modelling a clock, but it would make verification harder.

The state of a distributed system consists of the states of its component systems, for instance $\{s=0, s=1\} \times \{r=0, e0, r=1, e1\} \times \{m, m0, m1\} \times \{a, a0, a1\}$. In the beginning, each component system is in its initial state, which corresponds to the initial state to the whole system, e.g. $\langle s=0, r=0, m, a \rangle$.

Petri nets

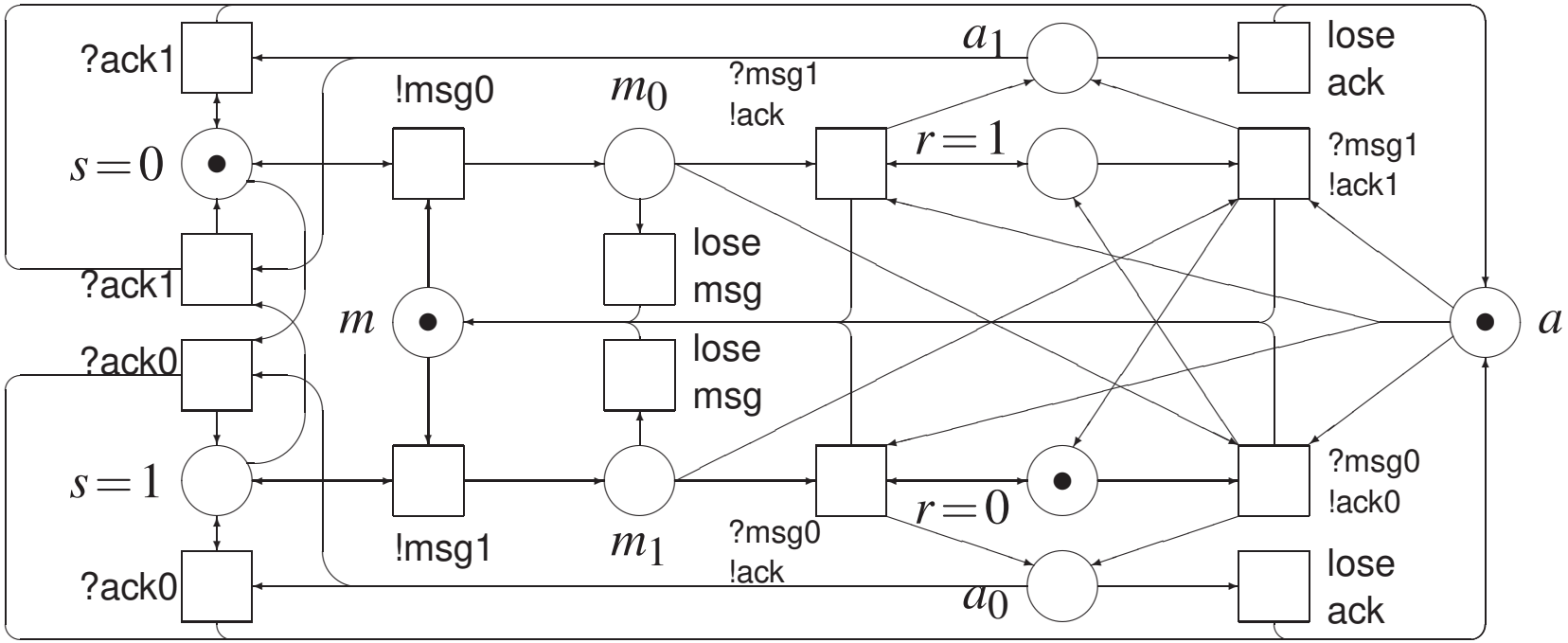
Labelled transition system diagrams may be difficult to read, since the automata are drawn as isolated entities, and the synchronisations between components are indicated with labels. Petri nets in a sense generalise labelled transition systems by depicting the synchronisations graphically:



Petri nets contain places ○ (Stelle) and transitions □ (Transition) that may be connected by directed arcs.

The state of a Petri system consists of local states (markings of places).

Example: Alternating bit protocol (5/6): Petri net



Token game (1/2)

- The state of a Petri system is formed by the distribution of tokens in the places.
- The state changes when enabled transitions are fired.
- A transition is enabled if each of its pre-places contains a token.
- When an enabled transition fires, a token is removed from its each pre-place and a token is inserted to each of its post-places.

Token game (2/2)

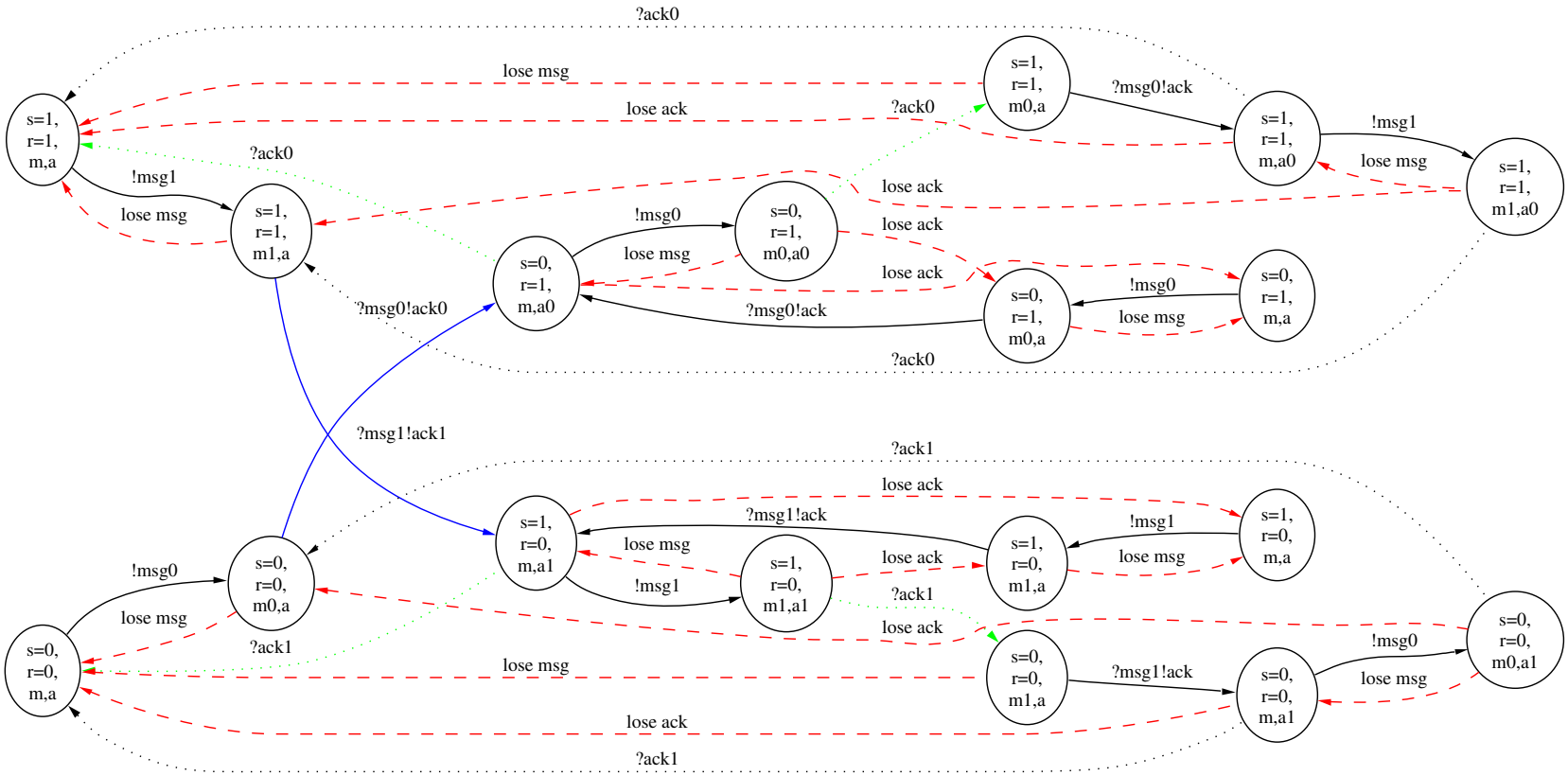
- The behaviour of the system can be described with event sequences. For instance, a successful message transmission looks like this: !msg0 ?msg0 !ack0 ?ack0.

- The event sequences can be composed into a directed graph that describes all possible events in the system.
  - state space: all states reachable from the initial state
  - reachability graph: the nodes are states and the arcs events between states

- By playing the token game, it is “fairly” easy to ensure that the alternating bit protocol works. There are at most $2 \cdot 4 \cdot 3 \cdot 3 = 72$ states.

Alternating bit protocol (6/6): reachability graph

It turns out that from $\langle s=0, r=0, m, a \rangle$, there are 18 reachable states and 40 events:



Many views of one system

Parallel and distributed systems can be described in very many ways:

- with labelled transition systems or process algebra,
- with Petri systems (place/transition systems or high-level systems),
- in some semi-formal description languages (such as UML) or
- in some programming language.

It makes sense to choose the presentation format according to the object being described and to the desired accuracy. Also formal descriptions can be presented in different equivalent notations: graphical, tabular, or plain text.