Q&A

• **Q**: “As it says in the lecture notes, LTL and Promela are not used in practise. Therefore it might be interesting to see some methods that are actually used.”

• **A**: SPIN (Promela and LTL checking) are used in practice!
  – I meant that Promela is not used as a programming language
informal requirements and domain knowledge

assert(...);

formalize

design model, and later implementation model

specification for MC

[]!a || b;
[]<> a;

model for MC

chan ts2c = ...

transform

fix bugs

transform

code

public void run(){
    ...
}

no more bugs → generate code

Model Checker

Promela is not used to write the final executable program (But there are also model-checkers for C, Java)

Promela is used here! (Either Promela models are modeled manually or they are created automatically from other models – as indicated here.)
Q&A

- see “success stories” http://spinroot.com/spin/success.html
- e.g. flood control: “verification of the control algorithms for the new flood control barrier built in the late nineties near Rotterdam in the Netherlands”
• “Selected algorithms for a number of space missions were verified with the Spin model checker. “

• “The model checker Spin and its Swarm verification front-end were used extensively in NASA's detailed investigation of the control software of the Toyota Camry MY05.”
Q&A

• **Q**: “Lets say our design models satisfy the formalized specification. Which techniques/tools are used later to check whether the implementation itself satisfies them too?”

• **A**: Model checking does not replace tests!
  • The model/specification may be wrong or it may not be detailed enough to capture critical details in the real system
    – for example: a processor cannot handle too many requests
  • The challenge is to test in a systematic way,
    – I ran 10,000 tests – how sure can I be sure that my system works correctly?
    – How can I obtain tests for cases I didn't test yet?

how does Spock know?

the system will fail with a probability of 0.237 percent
Q&A

• **Q:** “Graph Transformation Systems gives us an answer on how we make a graph more optimised and more easier. We wish to learn way to minimize automaton for a better communication with other engineers.”

• **A:** Graph Transformation System do different things
  – they will be introduced in the next lecture

• The minimization of automata will not be covered in this lecture.
  – you can search the literature for automata minimization

• There is also work on automatically optimizing the readability of Statecharts introducing hierarchichal states
Q: “Scenario-based Design could help to design more useful diagrams in our work after the university. Now we can make a sequence diagram but is it very useful and understandable?”

A: That depends!
- On the kind of system you will be developing about
- On the kind of behavior you need to specify
- On who you have to communicate with

Sequence Diagrams also have disadvantages
- Perhaps we would like to combine different modeling paradigms

I think it is true that engineers should have adequate languages for their different engineering tasks
- But they cannot learn too many languages
- Also diagrams are not necessarily better than text, ...
Q&A

• **Q**: “We would be interested in some example questions, which could be asked in the exam, so that we can estimate what is expected from us”

• **A**: I will tell you in the last lecture
Q: “In the lecture “Electronic Design Automation” Prof. Barke introduces several methods (heuristic, e.g.) to place components within a chip layout. He references to “Model checking” to verify, if a layout holds all electronic constraints. It could be interesting to have a glimpse on some real-life examples where the presented techniques are actually used.”

A: Model-Checking is used in chip design
   – verifying hardware circuits was one of the first application areas

Checking whether a layout satisfies constraints, or finding a correct layout, is usually done with SAT solvers
   – Finding via model checking: counter-example is solution
   – Some Model-checkers are implemented by SAT solvers
   – Some people call model-checking what's actually SAT solving
Last Time: Object System and Controller

- **Example: Production Cell**

  ![Diagram of a production cell with an object system and controller](image)

  **Controller** determines behavior of the object system:
  
  - $(ts, blankArrived, c)$
  - $(c, pickUp, a)$
  - $(c, moveToTable, a)$

  **Object System:**
  
  - $ts$
  - $a$
  - $c$
  - $b$
  - $p$
Last Time: Satisfying an MSD Specification

universal MSDs:

exists a run satisfying each MSD?  

controller:

existential MSDs:

all runs satisfy each uMSD?

...
Last Time: Universal MSDs and BA

- Example: RailCab requests permission to enter crossing

RequestEnterAtCrossing

<table>
<thead>
<tr>
<th>rc: RailCab</th>
<th>crc: CrossingControl</th>
<th>b: Barriers</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestEnter</td>
<td>closeBarriers</td>
<td>enterDenied</td>
</tr>
<tr>
<td>(c/m)</td>
<td>(h/e)</td>
<td>(c/e)</td>
</tr>
</tbody>
</table>

Diagram:

- requestEnter
- enterAllowed/enterDenied
- closed/blacked

States:
- c0
- c1
- c2
- c3
- sv

Transitions:
- rE
- cB
- eD

Symbols:
- Σ
- d
- bA
- cB
- rE
- Σ
- SV
- ed

Set: {cB, rE}
• The controller of an object system can be composed, via handshaking, of multiple smaller controllers
  – a controller can control one or many objects
  – one object can be controlled only by one controller
  – extreme: one controller per object
• Let $O_C$ be the objects controlled by a controller $C = (S, \Sigma_C, T, I)$
  – then $\Sigma_C = \Sigma \cap ((O_C \times \text{Name} \times O) \cup (O \times \text{Name} \times O_C))$ are the events of messages sent and received by the objects in $C$
• i.e., a controller cannot control messages sent and received by objects that it does not control
  – but a controller can control, also block, the sending as well as the receiving of messages of the objects it controls
Last Time: System / Environment Objects

- **Example: Production Cell**

One possible environment controller that allows all sequences of events to occur (see Assignment 8)

\[ \Sigma = \{bA, pUA, mTP, aAP, rB, mTT, aAT, p, pF, pUB\} \]

**environment controller:**

**object system:**

- Sometimes environment objects are represented with cloud-like shapes

- environment object
  - b:ArmB
  - p:Press
  - a:ArmA

- system object
  - c:Controller

- ts:TableSensor
Typically, we consider systems where we can control only a certain subset of objects; the other objects are *uncontrollable*, e.g. humans, a physical environment, or external software.

Given an object system \( O \), we define \( O_E \) and \( O_S \), \( O_E \cup O_S = O \), \( O_E \cap O_S = \emptyset \).

- \( O_S \) is called the *controllable objects* or *system objects*.
- \( O_E \) is called the *uncontrollable objects* or *environment objects*.

Our goal is to find a controller for \( O_S \) so that the global controller formed with any possible controller for objects \( O_E \) satisfies the MSD specification.

- Usually we must assume that anything can happen in the environment, but we can also formulate assumptions (later).
Last Time: Synchrony Assumption

• Our goal is to find a controller for $O_S$ so that the global controller formed with any possible controller for objects $O_E$ satisfies the MSD specification.

• Sometimes we can make the following general assumption to make the task easier:

• Be $C_E$ and $C_S$ the controllers for the environment resp. system objects, the we assume
  – if an environment event occurs, the system can take finitely many steps before it listens for the next environment event, i.e.,
    • the environment must never block system events
    • but the system must always eventually listen for environment events.
Our goal is to find a controller for $O_S$ so that the global controller formed with any possible controller for objects $O_E$ satisfies the MSD specification.

Sometimes we can make the following general assumption to make the task easier:

Be $C_E$ and $C_S$ the controllers for the environment resp. system objects, the we assume

- if an environment event occurs, the system can take finitely many steps before it listens for the next environment event, i.e.,
  - the environment must never block system events
  - but the system must always eventually listen for environment events

Can you think of criteria for the LTSs of the environment / system controllers that ensure this?
Design and Analysis of Distributed Interacting Systems

Lecture 10 – MSDs: Play-Out, Realizability, Advanced Concepts

Prof. Dr. Joel Greenyer

June 20, 2013
Agenda

• Environment & System Controllers
• Realizability
• Environment Assumptions
• The Play-Out Algorithm
• Advanced concepts:
  – Parametized Messages
  – Dynamic Object Systems and Dynamic Lifeline Binding
• Controller Synthesis
Restrictions on Controllers to Guarantee Synchrony Assumption

- **Environment events:** $\Sigma_{env} = \Sigma \cap (O_E \times Name \times O)$
- **System events:** $\Sigma_{sys} = \Sigma \cap (O_S \times Name \times O)$

**Environment Controller:** The environment must never block system events:
- Outgoing transition for every system event in every state

**System Controller:** The system can execute finitely many system events in reaction to environment events, but must always eventually listen for environment events
- Partition the set of states:
  - **active states:** only outgoing transitions labeled with system events
  - **inactive states:** for all env. events: outgoing transitions labeled with env events, no outgoing transitions labeled with system events
- The controller must always eventually reach an inactive state
Restrictions on Controllers to Guarantee Synchrony Assumption

- Environment events: \( \Sigma_{env} = \Sigma \cap (O_E \times Name \times O) \)
- System events: \( \Sigma_{sys} = \Sigma \cap (O_S \times Name \times O) \)

- **Environment Controller**: The environment must never block system events:
  - Outgoing transition for every system event in every state

- **System Controller**: The system can execute finitely many system events in reaction to environment events, but must always eventually listen for environment events
  - Partition the set of states:
    - **active** states: only outgoing transitions labeled with system events
    - **inactive** states: for all env. events: outgoing transitions labeled with env events, no outgoing transitions labeled with system events
  - The controller must always eventually reach an inactive state

when we talk about environment- and system controllers in the following, we mean controllers with these restrictions.

Simple solution: disallow cycles without inactive states
Implementation of an MSD specification

- If a system controller $C_s$ composed with any possible environment controller satisfies the MSD specification $MS$, $C_s$ is called an *implementation* of $MS$. 
Realizability

- An MSD specification $MS$ is *realizable* iff there exists an implementation of $MS$. 
Realizability – Example

- Is the following MSD specification realizable?

```
MSD PrepareCoffee
u:User  c:Controller  bu:BrewerUnit

(c/m) pressButton

boilWater (h/e)
prepareCoffee (h/e)
pourCoffee (h/e)

MSD DispenseCup

(c/m) prepareCoffee (h/e)

dispenseCup (h/e)
pourCoffee (h/e)
```

(yes)
Realizability – Example

- Is the following MSD specification realizable?

Who says that, for example, not arrivedAtTable or blankArrived occurs when expecting arrivedAtPress here? (this would lead to a safety violation)

(no)
Environment Assumptions

• Sometimes we have to assume that not arbitrary things can happen in the environment
  – environment assumptions or domain knowledge

• Idea: Formulate those with MSDs as well!

• Example: Assumptions in the Production Cell
  – if arm A is ordered to move to the press, it will eventually arrive at the press
    (unless it is ordered to move back to the table while it is “on its way” to the press)
Forbidden Messages

• while an MSD is active, no message event must occur that is
  unifiable with a **hot forbidden message** in that MSD
• if an MSD is active and a message event occurs that is
  unifiable with a **cold forbidden message**, this is a cold violation
  (which leads to the termination of that active MSD)

![Diagram]

the forbidden fragment is always at the end of an MSD. The active MSD terminates when it reaches the forbidden fragment, forbidden messages are never enabled
Environment Assumptions

• Example: Assumptions in the Production Cell
  – If a blank arrives and then arm A is ordered to move to the press, no other blank arrives before arm A has arrived back at the table
MSD Specification with Environment Assumptions

• We extend the definition of an MSD specification 
  \( MS = (O, \Sigma, D) \)
  – The set of MSDs \( D \) is partitioned into assumption MSDs \( A \) and requirement MSDs \( G \) (for guarantee)
  – \( A \cup G = D, A \cap G = \emptyset \)

• Let \( L(A) \) be the runs accepted by all assumption MSDs, and \( L(G) \) be the runs accepted by all requirement MSDs

• A controller \( C \) for the object system \( O \) satisfies an MSD specification iff
  – \( L(C) \subseteq (\Sigma^\omega \setminus L(A)) \cup L(G) \)
  – For all runs of the controller: the run either satisfies all MSDs in \( G \) or fails to satisfy at least one MSD in \( A \)
    • “if a run does not satisfy the assumptions it needs not satisfy the requirements”
MSD Specification with Environment Assumptions

• We extend the definition of an MSD specification
  \[ MS = (O, \Sigma, D) \]
  - The set of MSDs \( D \) is partitioned into assumption MSDs \( A \) and requirement MSDs \( G \) (for guarantee)
  - \( A \cup G = D \), \( A \cap G = \emptyset \)

• Let \( L(A) \) be the runs accepted by all assumption MSDs
  and \( L(G) \) by all runs accepted by all requirement MSDs.

• A controller \( C \) for the object system \( O \) satisfies an MSD specification iff
  \[ L(C) \subseteq (\Sigma^\omega \setminus L(A)) \cup L(G) \]
  - For all runs of the controller: the run either satisfies all MSDs in \( G \) or fails to satisfy at least one MSD in \( A \)
  - “If a run does not satisfy the assumptions it needs not satisfy the requirements”

usually, assumptions express what can happen in the environment (spontaneously) or how the environment must in turn react to system events

usually, requirements express how the system must react to environment events
Realizability – Example

• what now? realizable?

I think so...

this can be very hard to “see”, i.e. to check manually

<<EnvironmentAssumption>>

ArmAMoveFromTable
-ToPressAssumption

c:Controller

a:ArmA

moveToTable

arrivedAtTable

forbidden

arrivedAtPress

moveToTable

moveToPress

<<EnvironmentAssumption>>

ArmAMoveFromTable
-ToPressAssumption

c:Controller

a:ArmA

moveToPress

arrivedAtPress

forbidden

arrivedAtTable

moveToTable

moveToPress

<<EnvironmentAssumption>>

NoBlankArrivesBefore-
ArmAReturnedToTable

ts:TableSensor
c:Controller

a:ArmA

blankArrived

pickUp

moveToPress

arrivedAtPress

releaseBlank

moveToTable

arrivedAtTable

forbidden

blankArrived

moveToPress

arrivedAtTable

forbidden

blankArrived

moveToPress
Q&A

Q: “In some scenarios 1 second is long enough in some 1 second is too long. But how do we ensure that a result is delivered within a given interval in real time systems? Is it even possible or does it's solely dependent on the hardware?

A: Time is an important aspect in many systems, especially in the domain of embedded (or mechatronic) systems

Formal analysis of timed systems is possible
- discrete time: environment sends discrete ticks of a clock
- continuous time (hence, “real-time”): real-valued clock variables

Uppaal: Model-Checker for real-time systems
- unfortunately, no time to cover this in the lecture

MSDs can be extended with real-time constraints
MSDs with Real-Time constraints

- Blanks arrive in intervals, pressing, arm movement takes time

<<EnvironmentAssumption>>
BlankArrivalDelay

- `ts: TableSensor`
- `c: Controller`
- blankArrived
- `c = 0`
- `c > 6`

<<EnvironmentAssumption>>
PressPlateAssumption

- `c: Controller`
- `p: Press`
- press
- `c = 0`
- `c >= 1`
- pressingFinished
- `c <= 2`

<<EnvironmentAssumption>>
ArmAMoveFromTable
-ToPressTimeAssumption

- `c: Controller`
- `a: ArmA`
- `moveToPress`
- `c = 0`
- `c >= 2`
- arrivedAtPress
- `c <= 4`
- forbidden
- `moveToTable`
- arrivedAtTable
Agenda

• Environment & System Controllers (✓)
• Realizability (✓)
• Environment Assumptions (✓)
• The Play-Out Algorithm
• Advanced concepts:
  – Dynamic Object Systems and Dynamic Lifeline Binding
  – Parametrized Messages
• Controller Synthesis
The Play-Out Algorithm

- Our goal is to find a controller for \( O_S \) so that the global controller formed with any possible controller for objects \( O_E \) satisfies the MSD specification (under the synchrony assum.)

- Idea: Execute the MSDs!
  - Wait for any environment event to occur
  - While there are executed cuts, choose an enabled executed message to execute next, if it does not lead to a safety violation
  - Repeat the process.

what does play-out here?
do you see any problem?

play-out may get stuck if every possible next step leads to a safety violation
The Play-Out Algorithm

- What is the problem when playing-out this MSD?

Assumptions must be considered.
ScenarioTools Play-Out

ProductionCell [ScenarioTools Simulation]
Object System: (5 objects) [ts, a, p, b, c] executed with MSD specification (UML Package): ProductionCellIntegrated
- active requirement MSD: ArmATransportBlankToPress [ts->ts, c->c, a->a] (ts:1, c:3, a:2)
- active assumption MSD: NoBlankArrivesBeforeArmAReturnedToTable [ts->ts, c->c, a->a] (ts:1, c:2, a:1)
- active assumption MSD: ArmAMoveFromTableToPressAssumption [c->c, a->a] (c:1, a:1)
- Env Object: ts
- Env Object: a
- Env Object: p
- Env Object: b
- Sys Object: c

MessageEvent Selection View

<table>
<thead>
<tr>
<th>Message</th>
<th>Sending Object</th>
<th>Receiving Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>blankArrived()</td>
<td>ts:TableSensor</td>
<td>c:Controller</td>
</tr>
<tr>
<td>arrivedAtPress()</td>
<td>a:ArmA</td>
<td>c:Controller</td>
</tr>
</tbody>
</table>
ScenarioTools Play-Out

**Object System:** (5 objects) [ts, a, p, b, c] executed with MSD specification (UML Package): ProductionCellIntegrated

- active requirement MSD: ArmATransportBlankToPress [ts->ts, c->c, a->a] (ts:1, c:3, a:2)
- active assumption MSD: NoBlankArrivesBeforeArmAReturnedToTable [ts->ts, c->c, a->a] (ts:1, c:2, a:1)
- active assumption MSD: ArmAMoveFromTableToPressAssumption [c->c, a->a] (c:1, a:1)

**Environment Objects:**
- ts: TableSensor
- a: ArmA
- p: Press
- b: Blank
- c: Controller

**Message and Event Selection View:**
- blankArrived() [ts->c]
- arrivedAtPress() [a->c]

**Diagram Notes:**
- Leads to cold violation in assump. MSD
- Initializes new req. MSD but would lead to safety violation
- Progresses c/e and h/e cut in active assump. MSD
- Progresses h/m cut in req. MSD
Dynamic Systems

• Often systems consist of many components with changing relationships, e.g. RailCab:
  – which RailCab is in front?
  – which track section is a RailCab on?
  – is the RailCab currently in a station?

• The relationships influence what may/must happen
  – the relationships determine which object is responsible to do what (which “role” does which object play?)
  – certain things may/must happen only in certain situations (behavior can be “context sensitive”)
Dynamic Systems and Dynamic Bindings

- So far, a lifeline represented exactly one object
- Often we would like that the same scenario applies for different combinations of objects
  - lifelines should bind dynamically to objects

- Example: static binding
Dynamic Systems and Dynamic Bindings

- So far, a lifeline represented exactly one object
- Often we would like that the same scenario applies for different combinations of objects
  - lifelines should bind dynamically to objects

- Example: dynamic binding

(dynamic) object system

objects with links

binding expressions: determine which objects lifelines shall bind to (here: OCL)
• The dynamic binding can also exploit generalization relationships

in the class diagram:

i.e., this requirement holds also if the next track section is a crossing
Parametrized Messages

- Sometimes a choice is important, sometimes not

The parameter is not specified, isAllowed is an unbound variable. The scenario shall consider each possible value.

If the parameter is not specified, the message is also called symbolic, otherwise the message is concrete.

here a concrete value is specified
A message event in the system can be *unified* with a message in an MSD (called diagram message) if

- the message names are equal
- the source object is represented by the source lifeline
- the target object is represented by the target lifeline
Message Unification and Parametrized Messages

- A message event in the system can be *parameter unified* (or is *parameter unifiable*) with a message in an MSD iff
  - they are unifiable and
  - the diagram message is *symbolic* or
  - the diagram message is *concrete* and the parameter value carried by the message event is equal to the parameter value specified for the diagram message
Agenda

- Environment & System Controllers (✓)
- Realizability (✓)
- Environment Assumptions (✓)
- The Play-Out Algorithm (✓)
- Advanced concepts:
  - Dynamic Object Systems and Dynamic Lifeline Binding (✓)
  - Parametized Messages (✓)
- Controller Synthesis
Realizability – Example

- what now? realizable?

I think so...

this can be very hard to “see”, i.e. to check manually

we have to find an implementation or show that there is none...

... can we do this automatically?