Design and Analysis of Distributed Interacting Systems

Organization
Prof. Dr. Joel Greenyer

June 5, 2013
Oral Exams

• Oral exams take place between the August 13 and 15
• We will assign slots to students between 9.00 and 12.00 or 14.00 and 17.00

• Please send us an email by August 1\textsuperscript{st} and state
  – which of the days you prefer
  – whether you prefer the morning or afternoon
  – which times are impossible for you

• Please contact me soon if you require an exam outside of this period
Design and Analysis of Distributed Interacting Systems

Lecture 8 – LTL Model Checking Summary

Prof. Dr. Joel Greenyer

June 5, 2013
• Example: $\varphi = a \mathbf{U} b$

here the acceptance condition is $\{F_1\}$ with

$$F_1 = \{q_1, q_3, q_4, q_5\}$$
• Every GBA can be translated to a regular BA that accepts the same language

• For a $GBA = (Q, \Sigma, T, I, \{F_1, \ldots, F_n\})$ we construct a $BA = (Q \times \{0, \ldots, k\}, \Sigma, T', I \times \{0\}, F \times \{k\})$ with $((q, x), \sigma, (q', y)) \in T'$ if $(q, \sigma, q') \in T$ and
  – if $q \in F_i$ and $x = i - 1$ then $y = i$,
  – if $x = k$ then $y = 0$,
  – $x = y$ otherwise

• Example:

$$F_1 = \{q_1\},$$
$$F_2 = \{q_1, q_2\}$$
Summary LTL Model Checking, Complexity

\[ M \models \varphi \]

iff \[ L(M) \subseteq L(\varphi) \]

iff \[ L(M) \cap ((2^{AP})^\omega \setminus L(\varphi)) = \emptyset \]

iff \[ L(M) \cap L(\neg \varphi) = \emptyset \]

iff \[ L(B_M \otimes B_{\neg \varphi}) = \emptyset \]

What we need:
1. Checking emptiness of the language accepted by a BA
2. Product construction for BAs
3. Represent KS as BA
4. Represent LTL formula as BA

• Calculate of \( cl(\varphi)/cs(\varphi) : O(|\varphi|), |\varphi| \) is \# of operators in \( \varphi \)
• Create \( B_{\neg \varphi} : O(2^{\varphi}) \)
• Create \( B_M : O(|M|), |M| \) is the number of states and transitions of Kripke Structure M.
• Create \( B_M \otimes B_{\neg \varphi} : O(|M| \times 2^{\varphi}) \)
• Check \( B_M \otimes B_{\neg \varphi} \) emptiness: linear, i.e. \( O(|M| \times 2^{\varphi}) \)

→ Hence, overall complexity is linear in size of the model, but exponential in the size of the formula.
Verification in the Development Process

• How do we envision formal verification methods to be used in practice?
  – LTS and Promela are not modeling/programming languages that are used in practice...
  – Temporal Logics are difficult to write/understand by many software engineers, even experts

• Alternative:
  – check C or Java programs directly
    • for C: CBMC, BLAST, …; for Java: Java Pathfinder, BANDERA, …
    • check properties specified in program code or temporal logics
  – model-based software development
    • use adequate models (UML+extensions, DSLs) for the design
    • transform models to the input for a model-checker
    • generate runnable code from the models
Model-based Software Development and Formal Verification

informal requirements and domain knowledge

formalize

formalized specification

could be different models for different development stages, e.g. platform-independent, later platform-specific

design model, and later implementation model

no more bugs → generate code

public void run(){
    ...
}

code
Model-based software development: Goal: Provide engineers with *adequate languages* to describe their system/software.

*adequate language*: can capture all relevant information, unambiguous, understandable, suited for the development task, machine-processable
Model-based Software Development and Formal Verification

informal requirements and domain knowledge

formalize

formalized specification

transform

design model, and later implementation model

no more bugs → generate code

Model Checker (MC)

specification for MC

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

model for MC

public void run(){
  ...
}
Example of Transformation: SDL to Promela

- SDL (Specification and Description Language) to Promela:
Example of Transformation: Realtime Statecharts to Uppaal

- Realtime Statecharts: Timed extension of Statecharts (with clock variables that increase over time)
When is the system “correct”?

- **Correct:**
  - system does what the stakeholders want it to do
  - it does not do what the stakeholders do not want it to do

- Is the system really correct when model checking returns true?
  - requirements may be incorrectly formalized
  - domain knowledge could be overly optimistic or incorrectly formalized
  - bugs in the specification/model transformations
  - bugs in the code generator
  - bugs in the model checker

**must be validated!**
Design and Analysis of Distributed Interacting Systems

Lecture 8 – Live Sequence Charts (LSCs)

Prof. Dr. Joel Greenyer

June 5, 2013
Note: Also the specification may be incorrect

Model-checking may also reveal that the specification must be fixed! → then long and costly iterations required!
What is an adequate language to specify?

Model-based software development:


informal requirements and domain knowledge

formalize

formalized specification

design

design model, and later implementation model

no more bugs → generate code

code

public void run(){
    ...
}
Idea: Play-In / Play-Out

- Idea: “Play with the system and tell it what to do in certain cases”
  - called “Play-In” of behavior
  - record “rules” that can be executed later via “Play-Out”

- Example: Coffee machine
  - “when I insert a coin and press 'coffee', then the machine should dispense a cup of coffee”
  - “when I insert a coin and press 'tea' then...”
  - “when I press coffee and the coffee bean container is empty, the service lamp must light up and no coffee must be dispensed”
Live Sequence Charts – Informal Introduction

• Example: Coffee machine
  – “when I insert a coin and press 'coffee', then the machine should dispense a cup of coffee”

• Live Sequence Chart (LSC)

“Prechart”: sequence of events that must be satisfied if the prechart is satisfied.

“Main Chart”: sequence of events that must be satisfied if the prechart is satisfied.
Live Sequence Charts – Informal Introduction

• Example: Coffee machine
  – “when I press coffee and the coffee bean container is empty, the service lamp must light up and no coffee must be dispensed”

• Live Sequence Chart (LSC)

   ![Live Sequence Chart Diagram]

   LSC: SeriveLampOnWhenCoffeeBeansEmpty

   user

   coffee machine

   forbidden events: something must not happen while the scenario is “active”
Come, Let's Play

  (http://www.wisdom.weizmann.ac.il/~playbook/)
  (also free PDF of the book available)

  (http://www.wisdom.weizmann.ac.il/~harel/papers/LiberatingProgramming.pdf)

  (http://www.wisdom.weizmann.ac.il/~harel/papers/Behavioral%20programming%20.pdf)
Play-Engine

• Tool and many interesting examples:
  – Bakery oven

figure from Harel, D. & Marelly, R. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine
Play-Engine

- Tool and many interesting examples:
  - Cell Phone

figure from Harel, D. & Marelly, R. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine
Play-Engine

• Tool and many interesting examples:
  – Calculator

figure from Harel, D. & Marelly, R. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine
• Recent new version of the tool

see http://www.weizmann.ac.il/mediawiki/playgo/index.php/Main_Page
Play-In / Play-Out – Critical Thoughts

• (+) Play-In seems useful especially to describe the interactions of a user with an interface
• (-) Play-in difficult when there are more and more exceptions
  – manual modification of the LSCs necessary
• (+) LSCs are (relatively) intuitive for expressing many kinds of properties (what may, must (not) happen, loops, choice, ...)
• (-) There can be many unforeseen “overlappings” and contradictions between different scenarios
  – but this can also be the case when we use e.g. temporal logic – it is an inherent problem of the requirements
• (+) Play-out can help to analyze the behavior emerging from the combination of different scenarios and check for contradictions
Modal Sequence Diagrams (MSDs)

• Developed by D. Harel and S. Maoz as a formal interpretation of UML Sequence Diagrams

• Based on the concepts of LSCs
  – a bit different: no prechart
  – formalizing especially assert and negate constructs in UML SDs

• Supported by ScenarioTools
  – with a slightly modified semantics
  – which we will introduce in the following

Modal Sequence Diagrams (MSDs)

- **Continued in Lecture 9**
- (All slides hereafter were not covered in the lecture; a more systematic introduction to MSDs was given in Lecture 9)
Modal Sequence Diagrams (MSDs)

- An MSD specification consists of
  - a set of objects, called the *object system*
    - a subset of objects are *system objects*, the remaining objects are *environment objects*
  - objects can exchange *messages*
    - a message is identified by the sending object, a message name, the receiving object
    - we consider only synchronous messages: sending and receiving is one event, also called *message event*
  - The set of all message is the *alphabet*, $\Sigma$
  - a set of MSDs; there are two kinds of MSDs
    - *existential MSDs*: describe runs of a system that must be possible
    - *universal MSDs*: properties that must be satisfied by all runs
  - a subset of MSDs can be *assumption MSDs*, the remaining MSDs are called *requirement MSDs*
Modal Sequence Diagrams (MSDs)

- Example: Production Cell

plates leave system on deposit belt
blanks enter system on feed belt

ArmA
ArmB
Press
Controller

TableSensor

object system:

environment object

system object

a: ArmA
b: ArmB
c: Controller
p: Press
ts: TableSensor
Modal Sequence Diagrams (MSDs)

- A Modal Sequence Diagram (MSD)
  - Each lifeline represents an object in the object system
- In universal MSDs, message can have
  - an execution kind: monitored or executed
  - a temperature: hot or cold

The semantics will become clearer in a few moments...
Universal MSD – Semantics by Example

- **may happen**
- **can be “violated”**
- **must not be “violated” (safety)**

- **monitored**
  - cold
  - hot

- **executed**

**must eventually happen (liveness)**

---

**ArmATransportBlankToPress**

- ts: TableSensor
- c: Controller
- a: ArmA

- **blankArrived**
- **pickUp**
- **moveToPress**
- **arrivedAtPress**
- **releaseBlank**
- **moveToTable**
- **arrivedAtTable**
• If a message occurs in the system that can be *unified* with the first message in an MSD, an *active copy* of the MSD (also called *active MSD*) is created.
  
  – a message in the system can be *unified* with a message in an MSD if the message names are equal and the source/target objects are represented by the source/target lifeline

• The *cut* remembers which MSD messages were unified
• If the cut is in front of a message on its sending and receiving lifeline, the message is *enabled*
  – if an executed message is enabled, the cut is *executed*, otherwise it is *monitored*
  – if a hot message is enabled, the cut is *hot*, otherwise it is *cold*
• If a message occurs that can be unified with an enabled message, the cut progresses
If a message occurs that can be unified with an enabled message, the cut progresses.
Universal MSD – Semantics by Example

- If a message occurs that can be unified with an enabled message, the cut progresses
Universal MSD – Semantics by Example

- If a message occurs that can be unified with an enabled message, the cut progresses
If a message occurs that can be unified with an enabled message, the cut progresses.
• If a message occurs that can be unified with an enabled message, the cut progresses
• If a message occurs that can be unified with an enabled message, the cut progresses
• The active MSD terminates when the cut progressed beyond the last message
• If a message occurs that can be unified with a message in an active MSD that is not currently enabled, this is a violation
  – violations are allowed in a cold cut, then then active MSD terminates. This is called a cold violation.
  – violations are forbidden in a hot cut: safety violation
An active MSD must not remain forever in an executed cut
– liveness violation
An active MSD may remain forever in a monitored cut
Universal MSD – Iterative vs. Invariant Semantics

- If the first message of an MSD appears also later in the diagram, there may be multiple active copies of this MSD – *invariant semantics*
- There is also an interpretation of MSDs where only one active MSD copy of an MSD is allowed – *iterative semantics*

run: m1, m2, m1, ...

---

```
<table>
<thead>
<tr>
<th>a:A</th>
<th>b:B</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>
```

invariant interpretation:
• If the first message of an MSD appears also later in the diagram, there may be multiple active copies of this MSD
  – *invariant* semantics
• There is also an interpretation of MSDs where only one active MSD copy of an MSD is allowed
  – *iterative* semantics

run: m1, m2, m1, ...

iterative interpretation:
Concurrently Active MSDs

PressPlateAfterArmARelasesBlankPlate

- c:Controller
- a:ArmA
- p:Press
- b:ArmB

- releaseBlank
- press
- pressingFinished
- pickUp

ArmATransportBlankToPress

- ts:TableSensor
- a:ArmA
- c:Controller

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- h/e
- releaseBlank
- moveToTable
- arrivedAtTable
Concurrently Active MSDs

PressPlateAfterArmARelasesBlankPlate


- releaseBlank
- press
- pressingFinished
- pickUp

h/e

ArmATransportBlankToPress

ts:TableSensor  c:Controller  a:ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- h/e
- moveToTable
- arrivedAtTable
**Concurrently Active MSDs**

**PressPlateAfterArmARelasesBlankPlate**
- **c:Controller**
- **a:ArmA**
- **p:Press**
- **b:ArmB**

- releaseBlank
- press
- pressingFinished
- pickUp

**ArmATransportBlankToPress**
- **ts:TableSensor**
- **c:Controller**
- **a:ArmA**

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- moveToTable
- arrivedAtTable

h/e
h/m
Concurrently Active MSDs

PressPlateAfterArmAReleasesBlankPlate

- c:Controller
- a:ArmA
- p:Press
- b:ArmB

- releaseBlank
- press
- pressingFinished
- pickUp

ArmATransportBlankToPress

- ts:TableSensor
- c:Controller
- a:ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- moveToTable
- arrivedAtTable
- h/m
Concurrently Active MSDs

**PressPlateAfterArmARelasesBlankPlate**
- c:Controller
- a:ArmA
- p:Press
- b:ArmB
  - releaseBlank
  - press
  - pressingFinished
  - pickUp

**ArmATransportBlankToPress**
- ts:TableSensor
- a:ArmA
- c:Controller
  - blankArrived
  - pickUp
  - moveToPress
  - arrivedAtPress
  - releaseBlank
  - moveToTable
  - arrivedAtTable
  - h/m
  - h/e
Concurrently Active MSDs

PressPlateAfterArmAReleasesBlankPlate

- c: Controller
- a: ArmA
- p: Press
- b: ArmB

- releaseBlank
- press
- pressingFinished
- pickUp

ArmATransportBlankToPress

- ts: TableSensor
- c: Controller
- a: ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- moveToTable
- arrivedAtTable
Concurrently Active MSDs

PressPlateAfterArmAReleasesBlankPlate

- c:Controller
- a:ArmA
- p:Press
- b:ArmB

- releaseBlank
- press
- pressingFinished
- pickUp

ArmATransportBlankToPress

- ts:TableSensor
- c:Controller
- a:ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- moveToTable
- arrivedAtTable