University of Zurich
Department of Informatics

RERG

Software Quality Exercise 1

Model Checking, Bug Tracking, Version Control, Builder

1 Information

1.1 Dates

- Release: 08.03.2010 2pm
- Deadline: 22.03.2010 2pm
- Discussion: 29.03.2010 3.30pm

1.2 Formalities

Please package the files containing your solution into a *zip* (or *tar.gz*) file and submit it via email to `jeanneret@ifi.uzh.ch`. The subject of the email must begin with *[FS 10 SWQ]*. Answer the questions in a *pdf* document and include the source code (*java*, *pml* and *ltl* files) of your solutions. Do not include binary files such as *class* files, metadata (such as *.svn* folders or *.project* files) nor derived files like *pan.c*.

Exercises can be solved and handed in in groups of two. Every member of a group must be able to answer questions about the group's solution.

This exercise is made of 2 parts. Part 1 is dedicated to model checking, while the second is essentially a preparation for the next exercises. Note that **exercise 3.1 requires the intervention of the assistant**, thus, we strongly recommand you to solve it as early as possible.

Many exercises refer to an internal server of our research group named *daiquiri*. To access it, you need to be connected to the university network, either physically (in lab rooms or via wifi) or through a VPN connection.

Finally, assignments are written in English, but feel free to write your answers in German (or in French) if you like to do so. Nevertheless, tickets, wiki entries and code checked in the SVN should be written in English.

2 Model Checking

These exercises are based on *spin*, a model checker. You can find pointers to documentation about this tool on `http://daiquiri.ifi.uzh.ch/trac/swq10/wiki/SPIN`. The tool has been installed on the computers in the lab rooms.

Note: Here are some helpful commands for these exercises:

- Simulate a model (append `-p -g` for more verbose output)

  ```
  spin model.pml
  ```

- Verify a property on a model.

  ```
  spin -a -F property.ltl model.pml
  gcc -o pan pan.c
  ./pan -a
  ```

- Replay a trace found by the verifier (append `-p -g` for more verbose output).

  ```
  spin -t model.pml
  ```

2.1 Introductory Example: Colony of Chameleons

A friend of yours is the director of a zoo. He has described you the latest acquisition he was considering for his zoo: a rare colony of chameleons. This colony of chameleons includes 20 red, 18 blue, and 16 green individuals. Whenever two chameleons of different colors meet, each changes to the third color. The color mutation is an unique feature and your friend expects to attract many visitor with this colony. Still, the director has a doubt: if, by any chance, all 54 chameleons are in the same color, there will not be any color change any more and the colony would loose its value.

2.1.1 State Space

a) Give a rough estimation of the number of states in which the colony can be.
b) How many states present the undesired property?

2.1.2 Promela Model of the Colony

Download the file *Colony.pml* on the website of the exercises. Study this model with the help of Promela documentation.

a) What are the main components of a Promela model?
b) The behavior of chameleons is not deterministic, in the sense that they meet each other randomly. How is this non-determinism expressed in the Promela model of the colony?
c) Which other aspect of Promela is not deterministic?
d) Explain briefly what are *d_step* blocks and the reason of their presence in the Promela model of the colony. In which sense is the first instruction of such a block different than the others?
e) Despite their relation with the color change phenomenon, the duration of a color change and the probability of a change have not been modeled. Explain, in maximum 5 sentences, why they are not relevant for the problem at hand.

2.1.3 Simulation and Verification

a) Run a few simulations of the model. Have you ended up in a state where all chameleons are of the same color?
b) Express the property *there are always at least two chameleons of different color* in an LTL formula using the variables defined in the Promela model of the colony. Save this formula in a textfile named `ColorChange.ltl`. Note that you cannot define predicates in an *ltl* file; you have to define them as macros in the Promela model. For example:

```
#define noRedChameleonLeft (!nRed)
```

c) Is it a *safety* or a *liveness* property? Why?
d) Using *spin*, investigate whether the colony can reach a state where this property does not hold. Note that *spin* looks for executions that satisfy an LTL property, so, you have to negate the formula in `ColorChange.ltl`. What will you recommand to your friend?
e) In the report of *pan*, the number of *states stored* is the number of reachable states. How many states were reachable? Compare with your answer in 2.1.1.a).
f) In the report of *pan*, the number of *transitions* is the number of transitions that have been visited during the search. How many transitions were investigated by *pan*?
g) A few days later, the director of the zoo informs you: a blue chameleon escaped during the delivery. Has he any reason to worry?
h) What is the depth of the trace (that is, the number of steps) found by *pan*? Why does the trace contain more steps than the number of reachable states? What is the actual number of color changes?

2.1.4 Extending the Model

After a few weeks, you visit the zoo. Looking at this chameleon colony, you feel a strong relief: there are still chameleons of every color and they change colors happily. Still, by observing them, you figure out that your friend did not describe the behavior of the chameleons very precisely. You make the following additional observations:

- When 3 chameleons of different colors meet, they argue violently and kill eachother.
- When 2 chameleons of the same color meet, they give birth to a new chameleon of their color.
- At the day of your visit, there were 20 red, 18 blue, and 16 green individuals.

a) Extend the Promela model of the colony with 2 processes so that it reflects your new observations.
b) Run a few simulations of this model. What happens? Explain.
c) Let's make an assumption here: chameleons only give birth if there are less than N chameleons in the colony (initially, set N to 25). Modify the model accordingly.
d) Express the property *soon or later, there will be no chameleon left* in an LTL formula. Save this formula in a textfile named `Extinction.ltl`.
e) Is it a *safety* or a *liveness* property? Why?
f) Using *spin*, investigate whether the colony can die off. Are chameleons threatened with extinction?
g) Change the the number N to 300 instead of 25. As you did previously, investigate whether the colony can die off. What happens? Why?
h) Set the max search depth to 5'000'000 steps when performing the verification. Do you obtain a different result? What is the depth of the trace? Which depth has been reached during the search?

i) Change the order of process declarations in the Promela model by swapping the *fight* process with the *birth* process and re-verify the model. What is the depth of the trace and which depth has been reached during the verification? Explain the difference with your previous results. Your explanation must account the fact that the behavior of the model remains unchanged by your modification.

2.1.5 Scalability

This exercise is based on the extended model of the colony built in the exercise 2.1.4. No LTL property will be checked on the model. While keeping the max search depth to 5'000'000 steps, perform an exhaustive search on the model for $N = 50$, $N = 100$, $N = 150$, $N = 200$ and $N = 250$. Note that you have to invoke *pan* without the option `-a`, since no LTL property was specified. For every execution of *pan*, report the following information in a table:

- the time spent for the verification.
- the number of transitions explored.
- the length of the longest trace explorated.
- the number of reachable states.
- the memory used for the verification.

2.2 (Optional) Optimization through Model Checking: Traveling Professor Problem

In addition to detect flaws in a design, a model checker can be used to solve optimization problems. Consider the following problem:

> A well-known professor of Software Engineering at the University of Lugano intends to visit his colleagues across the country. There are 5 research groups working on SE in Switzerland:
> - in a service company in Basel,
> - at the University of Bern,
> - in the CERN in Geneva,
> - at the EPFL of Lausanne, and of course,
> - at the University of Zurich.
>
> The professor wants to visit all these research groups, but he does not want to visit a city twice. Your assignment is to find the shortest tour. Distances among these 5 cities are given in table 1.

a) Create a Promela model of the behavior of the traveling professor. Save it in a file called `TravelingProfessor.pml`. Consider the following hints:
   - The professor is initially in Lugano and chooses, non-deterministically, his next destination.
   - The professor keeps track of the cities he has visited and the kilometers he has made so far.
   - A Promela process can terminate (in the previous exercise, processes never ended).

	Lugano	Zurich	Basel	Bern	Lausanne	Geneva
Lugano	-	217	267	246	355	371
Zurich	217	-	85	126	227	278
Basel	267	85	-	98	201	251
Bern	246	126	98	-	101	157
Lausanne	355	227	201	101	-	63
Geneva	371	278	251	157	63	-

Table 1: Distance among cities in Switzerland

- Promela offers a *goto* construct (and it is perfectly fine to use it).
- The process should produce enough output to display the route it has chosen and the distance of the trip.

b) Run a simulation of your model. Which route has been chosen? How much distance has been traveled by the professor?

c) Express the following property *The distance traveled so far is always smaller than the distance found in b)* in an LTL formula. Save this expression in a file called `Optimum.ltl`.

d) With *spin*, verify whether the professor can find a shorter path than the one found in b).
- If no example was found by *spin*, you have already found the optimal tour.
- Otherwise, replay the execution trace found by *spin* and update the `Optimum.ltl` file accordingly. Repeat step d).

e) What is the shortest tour? What is its length?

f) Explain in a few sentences why *spin* is less efficient than dedicated algorithms to solve such optimization problems.

2.3 Back to Software Engineering: Mutual Exclusion Algorithm

In the material of the lecture (slides 10, 11 and 12 of chapter 2), you have seen the mutual exclusion problem, as well as two of its related properties. Create two models of mutual exclusion algorithms, one based on a semaphore, the other being Peterson's algorithm. Using *spin*, verify whether these algorithms (a) ensure mutual exclusion and (b) are starvation free (properties (1) and (2) respectively on slide # 10).

3 ImageJ

In this part, we introduce various tools in preparation of the next exercises. This semester, they are based on an existing software for image processing: *ImageJ*.

3.1 Bug Tracking: Trac

To support these exercises, we have created a *trac environment* on *Daiquiri*: `http://daiquiri.ifi.uzh.ch/trac/swq10/`. *Trac* provides a wiki, on which you will find pointers to the documentation of the various tools used in these exercises.

Every participant is supposed to do this part individually, even if otherwise you work in a group.

a) Register on this website. As username, choose the *sxxyyyzz* (where *xxyyyzz* is your student number without the last digit) part of your student email address. Do not forget to enter a valid email address, in order to be notified of any updates on your tickets.
b) Create a new ticket to ask for access to the version control repository. Fill in the form with as much detail as possible. Assign it to the assistant: *m1049749*. Write down the number of your ticket.
c) Wait on the assistant to confirm your access with a comment on your ticket.

3.2 Version Control: SVN

The *ImageJ* project is hosted on an SVN repository (note: the actual source code of the project has not been imported yet) located on *Daiquiri*.

a) Within an SVN repository, a project is layouted in 3 directories: `trunk`, `tags` and `branches`. What are these directories used for?
b) Check out the trunk of the project with the command:

   ```
   svn co https://daiquiri.ifi.uzh.ch/svnswq10/imagej/trunk/ ImageJ
   ```

   The error message is due to the fact that the server certificate has not been signed by a CA. Still, accept it *permanently*. Provide the *username* and the *password* you have registered with in *Trac*.
c) Open the file `ImageJ/Participants.txt` and append a line in it to register yourself (or your group):
   - If you work alone, add `SOLO L`, where *L* is your username for *trac*.
   - Otherwise, type in `GROUP L M`, where *L* and *M* are your respective usernames for *trac*.

   Check in your modification. The comment of your commit must be `Exercise Registration ticket:n`, where *n* is the number of your ticket (from 3.1.b). Write down the revision number of your commit.
d) On your ticket, write a comment confirming that your registration was successfull. Your comment should include the `changeset:m`, where *m* is the revision number of your commit (3.2.c).
e) Close your ticket.
f) Have a look on the *timeline* view of *Trac*. Report all its entries related to your registration.

3.3 Builder: Maven2

The *ImageJ* project is built by *Maven2*.

a) Within the `ImageJ` directory, execute the following command:

   ```
   mvn install
   ```

   Was the build successful?
b) In maximum 5 sentences, describe the role and the content of a `pom.xml` file.
c) What are the main phases of the *build lifecycle* of Maven2?