Prof. Dr. A. Podelski,
Dr. B. Westphal
C. Schilling

Summer term 2018

Softwaretechnik/Software Engineering

http://swt.informatik.uni-freiburg.de/teaching/SS2018/swtvl

Exercise Sheet 5

Early submission: Wednesday, 2018-07-04, 12:00 Regular submission: Thursday, 2018-07-05, 12:00

Exercise 1 – OCL (5 Points + 5 Bonus)

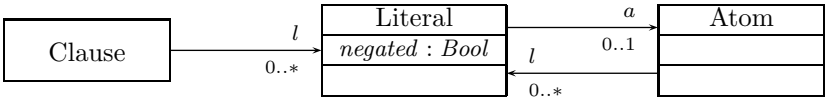

Figure 1: Class diagram for a clause representation.

Recall the class diagram for clause representation from Exercise Sheet 4 as shown in Figure 1.

(i) Assume that the possible system states from Exercise Sheet 4, Task 3.(ii) for the class diagram in Figure 1 have been shown to the customer and they realized that some of the examples are actually undesired. The developer's understanding from a discussion of the restriction with the customer has been formalised in the following Proto-OCL formula.

$$F := \forall\, self \in allInstances_{\text{Clause}} \bullet \forall\, \ell_1 \in l(self) \bullet \forall\, \ell_2 \in l(self) \bullet \ell_1 \neq \ell_2 \implies a(\ell_1) \neq a(\ell_2)$$

To validate the formalisation, present system states $\sigma_1, \sigma_2, \sigma_3$ such that

a) formula F evaluates to *true* for σ_1,

b) formula F evaluates to *false* for σ_2,

c) formula F evaluates to $\perp$ (undefined) for σ_3.

Represent each of the system states by an object diagram. (3)

(ii) Choose one of your system states σ_2 or σ_3 from Task (i) and give a proof, using the interpretation function as defined in the lecture, that the formula actually evaluates to the value you claimed. (2)

To save time, skip the steps of quantifier instantiation. Instead, start with the expression $\mathcal{I}[\![\ell_1 \neq \ell_2 \implies a(\ell_1) \neq a(\ell_2)]\!](\sigma_i, \beta)$*, where* σ_i *is your choice between* σ_2 *and* σ_3*, and* β *is a suitable valuation. Argue whether it is sufficient to only look at one specific valuation. If not, explain how the results for the other valuations look like and how they affect the result.*

(iii) *Writing Proto-OCL formulae is a creative act and harder than understanding a given formula or evaluating a formula on a given system state. Here is a little challenge for the students in the mood to give it a try.*

Use Proto-OCL to formalise the following three requirements on the system states to be used in the software:

a) Each Literal instance has an Atom instance (as 'a').

b) Each Clause instance has exactly two (different) Literal instances (as 'l').[1]

c) If a Literal instance is an 'l' of an Atom instance, then this Atom instance is also an 'a' of this Literal instance.

Demonstrate that each of your solutions is plausible:

- Give an example of a nontrivial system state.
- Write down the expected outcome for this system state when considering the respective requirement.
- Argue why your formula evaluates to the expected outcome. (5 Bonus)

Hint: You may use the function symbol 'size $: 2^\tau \to \mathbb{Z}_\bot$ *' with*

$$size_{\mathcal{I}} : \mathcal{I}[\![2^\tau]\!] \to \mathcal{I}[\![\mathbb{Z}_\bot]\!]$$
$$size_{\mathcal{I}}(x) = \begin{cases} |x| & x \neq \bot \\ \bot & otherwise. \end{cases}$$

Extra challenge (but no extra bonus): Do not use the function 'size'.

[1] The motivation for this restriction is that the customer wants to use an *efficient* SAT solver. For formulas with at most two variables, there exist polynomial-time algorithms to solve the SAT problem. For more information, we refer to `https://en.wikipedia.org/wiki/2-satisfiability`.

Exercise 2 – CFA Models (10 Points)

Mutual exclusion is the problem of making sure that when multiple processes or entities access a single shared resource, they do it such that only at most one process or entity obtains access to the shared resource at any given time. We call the part of each process that performs work on the shared resource its *critical section*. Mutual exclusion thus should ensure that, at any given time, at most one process can be in its critical section.
For instance, a factory may have a single machine for a production process step that is fed by multiple incoming production lines. The machine can service only one production line at a time. An initial solution is the use of lock variables, that is, shared variables that indicate whether the resource is in use. A process waits until the value of the lock variable indicates that the resource is free, sets the value of the variable to indicate that the resource is locked, performs the required work, and then resets the variable to indicate that the resource is free again.

For the following tasks, consider the CFA model of mutual exclusion by using lock variables shown in Figure 2. The model contains two identical processes `P1` and `P2` and a shared resource `Worker`.

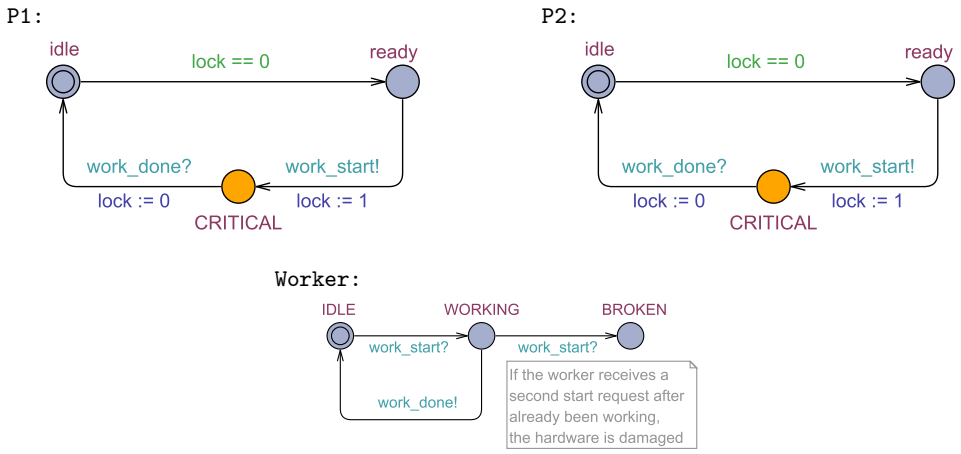

Figure 2: CFA Model $\mathcal{C}$ of mutual exclusion by using lock variables.

(i) Draw the reachable part of the transition graph of the CFA model. Make sure to clearly indicate the initial configuration(s). (5)

(ii) Is mutual exclusion satisfied in the model? Prove your answer by using the transition graph. (1)

(iii) Does the model have a *deadlock*? Prove your answer by using the transition graph. (1)

(iv) Use Uppaal to simulate the model and generate a trace that demonstrates a scenario where mutual exclusion *is* satisfied. Make sure to submit your model and trace files. (2)

(v) Give a query to check mutual exclusion on your Uppaal model. Use the verifier to check whether your query is satisfied. Document your results. (1)

See Appendix A for instructions how to use Uppaal.

Exercise 3 – CFA Model Verification (5 Points)

A more sophisticated approach to ensure mutual exclusion is the *Bakery algorithm*[2]. It works by assigning each process requesting access to the critical section a number, much like the numbers printed on tickets for queuing at a bakery. Each process waits until no other process has a lower number. Since obtaining a number happens concurrently, two processes may obtain the same number. In that case, the bakery algorithm breaks the tie by giving priority to the process with the lowest identifier (PID).

(i) The files `bakeryA.xml`, `bakeryB.xml` and `bakeryC.xml` provide models of three different proposals to implement the Bakery algorithm. The corresponding developers are not available and have not provided convincing arguments for why their implementation is correct; we doubt that all of them are.

Analyse the three models for whether they satisfy mutual exclusion. (3)

Hint: You may use any approach you like for the analysis.

For those models that do not satisfy mutual exclusion, (construct,) save, and submit a trace file that demonstrates this fact; make sure to specify which trace file corresponds to which model. For those models that do satisfy mutual exclusion, provide a strongly convincing argument why they do.

One approach to this task is to ***write a query*** *that serves to determine which of the three models guarantee mutual exclusion, and to use the* Uppaal *tool accordingly. You can determine whether process k is in its critical section by using the expression* `Process(k).CRITICAL`*. Explain in your own words how your query works, what you observe with your query, and how it relates to the property of mutual exclusion.*

(ii) For each of the three above models we would like to *investigate* how the complexity of verifying resp. falsifying the property behaves with respect to the number of processes. You can adjust the number of processes in a model by setting the value of the constant `N` in the global declarations.

Measure the number of explored states, and the time and virtual memory used for checking mutual exclusion[3] of each model configured for $\texttt{N} = 2, 3, 4, \ldots$ processes, and present the results.

What is the maximum number of processes that can be analysed on the machines in the computer pool? Analyse your results and state a hypothesis about the complexity of the verification and falsification relative to the number of processes. (2)

Make sure to keep the computing environment conditions stable between measurements. Do not forget to indicate the specifications of the machine on which you performed your measurements, and state by which procedure you obtained your timing measurements (to make the experiment reproducible).

See Appendix A for instructions how to use Uppaal and the command line verifier, and how to show the machine specifications.

[2] `https://en.wikipedia.org/wiki/Lamport's_bakery_algorithm`

[3] If you followed an approach not using Uppaal queries in the previous task, just use the query "`A[] not deadlock`" (it checks for absence of deadlocks in the model). Use the command line verifier to obtain the number of explored states, and time and memory consumption.

A Command Line Usage Instructions

UPPAAL and a command line verifier are available on the Linux machines in the computer pool. To run UPPAAL, use the following command:

```
/usr/local/ufrb/uppaal/uppaal-4.1.19/uppaal
```

To run the UPPAAL command line verifier, use the following command:

```
/usr/local/ufrb/uppaal/uppaal-4.1.19/bin-Linux/verifyta -u <model> <query>
```

where `<model>` is the file where your model is stored and `<query>` is the name of a text file containing the query to verify. The output of running `verifyta` looks like the following:

```
Options for the verification:
  Generating no trace
  Search order is breadth first
  Using conservative space optimisation
  Seed is 1467050321
  State space representation uses minimal constraint systems

Verifying formula 1 at line 1
 -- Formula is satisfied.
 -- States stored : 145 states
 -- States explored : 109 states
 -- CPU user time used : 430 ms
 -- Virtual memory used : 25288 KiB
 -- Resident memory used : 4908 KiB
```

To show the CPU model name, use the following command:

```
less /proc/cpuinfo
```

To show the available main memory, use the following command (look for line `MemTotal`):

```
less /proc/meminfo
```