Model-based Software Engineering

Summer Term 2016

Mini-Project 2 Assignment:
State machine networks: build a code generator and interpreters

Note:

- *The deadline for submitting the solutions for the mini-project 2 is Monday, June 27th, 23:59 CET.*
- *You can work on the mini-project in groups of three to four students. Use the lecture's Stud.IP forum for forming groups if necessary.*
- *Use the lecture's Stud.IP forum for discussion about the project, for asking questions and submitting your solutions. There is a dedicated category for mini-project 2 with an area for discussion and an area for submissions.*
- *Each submission must use its own topic. The first post of your topic must contain (1) the names of all group members, (2) a screenshot of your editor, (3) a download link to a ZIP file that contains the eclipse projects that comprise your solution, and (4) a download link to a PDF file that describes your solution. This description should be in the style of presentation slides and must include a brief description on how to use your editor and an explanation of your solutions to each of the tasks below.*
- *Presentation: The results of this mini-project can be presented in the tutorial session on Tuesday, June 28th. A mini-project can be presented by one member of a mini-project group, who can obtain the presentation bonus by doing so (see bonus point rules explained in the first lecture). If you want to present, send me an email with your presentation slides in PDF format at least by 23:59 on the Monday before the tutorial. Presentations should take between 5 to 10 minutes.*

State machine networks

In the first mini-project, you should have *defined a language* and *built graphical and textual editors* for networks of simple communicating state machines. In this mini-project, you shall now exercise three different ways of *executing* the state machine networks that you can model with your tools.

Before explaining the tasks in detail, we first describe the semantics of the state machine networks a bit more precisely.

State machine network semantics

When talking about the semantics of state machine networks, the following concepts are important

- *Current state*: In a running state machine network, every state machine has exactly one current state. The outgoing transitions of a current state are called *enabled*.
- *Channel buffer*: When sending messages via an asynchronous channel, the sending and the receiving of the message can happen asynchronously. Messages sent and not received over a channel are stored in a channel buffer. A channel buffer for an asynchronous channel is a counter that counts how many messages were sent but not yet received over that channel. Each asynchronous channel has its own channel buffer (counter).

The *global execution state* of the state machine network can be described by a mapping of each state machine in the state machine network to its current state and by a mapping of each asynchronous channel to a natural number value (including zero), which represents the number of messages buffered by this channel.

The state machine network is *initialized* and can then progress by making *steps* (with possibly a non-deterministic choice among different possible steps[1]). The steps are atomic and happen sequentially:

- *Initialization*: The global execution state of a state machine network is initialized as follows (before any steps happen):
  - The current state of each state machine is its initial state.
  - Each asynchronous channel is mapped to zero (to indicate that there is no message in the channel buffer).
- *Step*: A step consists in *firing* (or *taking*) one or two transitions; we distinguish the following cases
  - *Firing a message that sends over an asynchronous channel*: If a transition is enabled that sends over an asynchronous channel, this transition can fire. After firing, the current state of that state machine is the target state of that transition. Furthermore, the channel buffer counter is increased by 1.

[1] Where there is a non-deterministic choice between steps, a code generator or interpreter can implement some deterministic of random strategy to choose the particular step to be performed.

- *Firing a message that receives over an asynchronous channel*: If a transition is enabled that receives over an asynchronous channel and the channel buffer counter is greater than zero, this transition can fire. After firing, the current state of that state machine is the target state of that transition. Furthermore, the channel buffer counter is decreased by 1.
- *Firing a pair of transitions that are synchronized via a synchronous channel*: if in two state machines transitions are enabled where one sends and the other receives over the same channel, then both transitions can fire synchronously. After firing, the current states of the two state machines are the target states of the respective transitions.

  (Note that there could be multiple transitions enabled that send and receive via the same synchronous channel. Then all possible combinations of firing one sending and one receiving transition are valid steps.)

If no next step is possible, the execution has reached a *deadlock* and terminates.

If the above definition remains unclear or ambiguous, please ask clarification question in the forum.

1 Build a code generator

Build a code generator that transforms a state machine network model into Java code that can execute a sequence of steps that is valid w.r.t. the above semantics description. You can extend the code generation mechanism suggested by Xtext as was also presented in the lecture to implement a simple code generator for Petri nets. The generated code should print of log information about the execution steps (optionally also about the global execution state).

Test your code generator with at least three different state machine networks and validate that the execution sequence is one that is valid w.r.t. the execution semantics described above. Test also state machine networks with deadlocks to see what happens. You may also check if your code can run into an execution where it will always send but never receive over some channel (and this the buffer will grow ad infimum).

2 Build an interpreter

In addition to the code generator also build an interpreter as described in the lecture: Extend the state machine metamodel with run-time specific concepts and implement the interpreter logic in the form of Java code.

Optional: Benchmark the performance of the generated code and your interpreter.

3 Model the interpreter logic using Henshin

Instead of implementing the interpreter logic in Java code, now model the initialization and step logic using graph transformation rules, using the Henshin tool (`https://www.eclipse.org/henshin/`).

Then use the Henshin state space explorer feature to explore all possible states of execution for at least one example state machine network. (Choose state machine networks where the channel buffer counter will not grow ad infimum.)