Process interaction modeling

Prozessorientierte Modellierung.

Entitäten werden durch ihren **Lifecycle** modelliert, d.h. sie werden als "Arbeitsablauf"/Prozess dargestellt. Für eine single server queue kann man sich beispielsweise folgende Lifecycles vorstellen: [Figure 1]

Umsetzung als Simulation

Um daraus eine Simulation basteln zu können, wäre zusätzlich noch eine Queue und ein Kunden-Spawner zu modellieren.

Die Interaktionen zwischen den Lifecycles umzusetzen ist dann die nächste Hürde. Prozesse können in Endlosschleifen laufen (z.B. der Schalter). Damit die Simulation solche Prozesse trotzdem ausführen kann, können Prozesse "pausiert" werden: dies geschieht entweder für eine fixe Dauer (`hold`) oder bis ein anderer Prozess einen aufweckt (`passivate`).

`hold` modelliert dabei Aktivitäten, die in der echten Welt Zeit kosten würden, z.B. könnte in der Single Server Queue der Schalter während der Bedienungszeit `hold`en. `passivate` eignet sich für Wartephasen, beispielsweise wenn keine Kunden in der Queue sind. Die Queue würde den Schalter dann wieder aufwecken, wenn ein Kunde eingetroffen ist.

Eine mögliche Implementation obiger Lifecycles als prozessorientierte Simulation mit Modellierung der Interaktionen via `hold` und `passivate` könnte demnach so aussehen: [Figure 2]

Korrespondenzen zu Event Scheduling

Ähnlich wie bei der event scheduling view schreitet die Zeit nur in diskreten Schritten zwischen `hold`s voran. Der **Scheduler** muss eine Liste von den Zeitpunkten haben, an denen alle `hold`s enden. und arbeitet diese ab.

Das Wiederaktivieren eines Prozesses nach `passivate` entspricht einem Event.

DESMO-J

Grundlagen

Simulationen werden in 4 grundlegenden Schritten konstruiert:

1. Model design: identify the system to model, decide on system boundaries (what to include/exclude), and use abstraction and aggregation to arrive at a conceptual model which simplifies the real system just enough to be useful
2. Model implementation: map the conceptual model to an executable model, i.e. a computer program

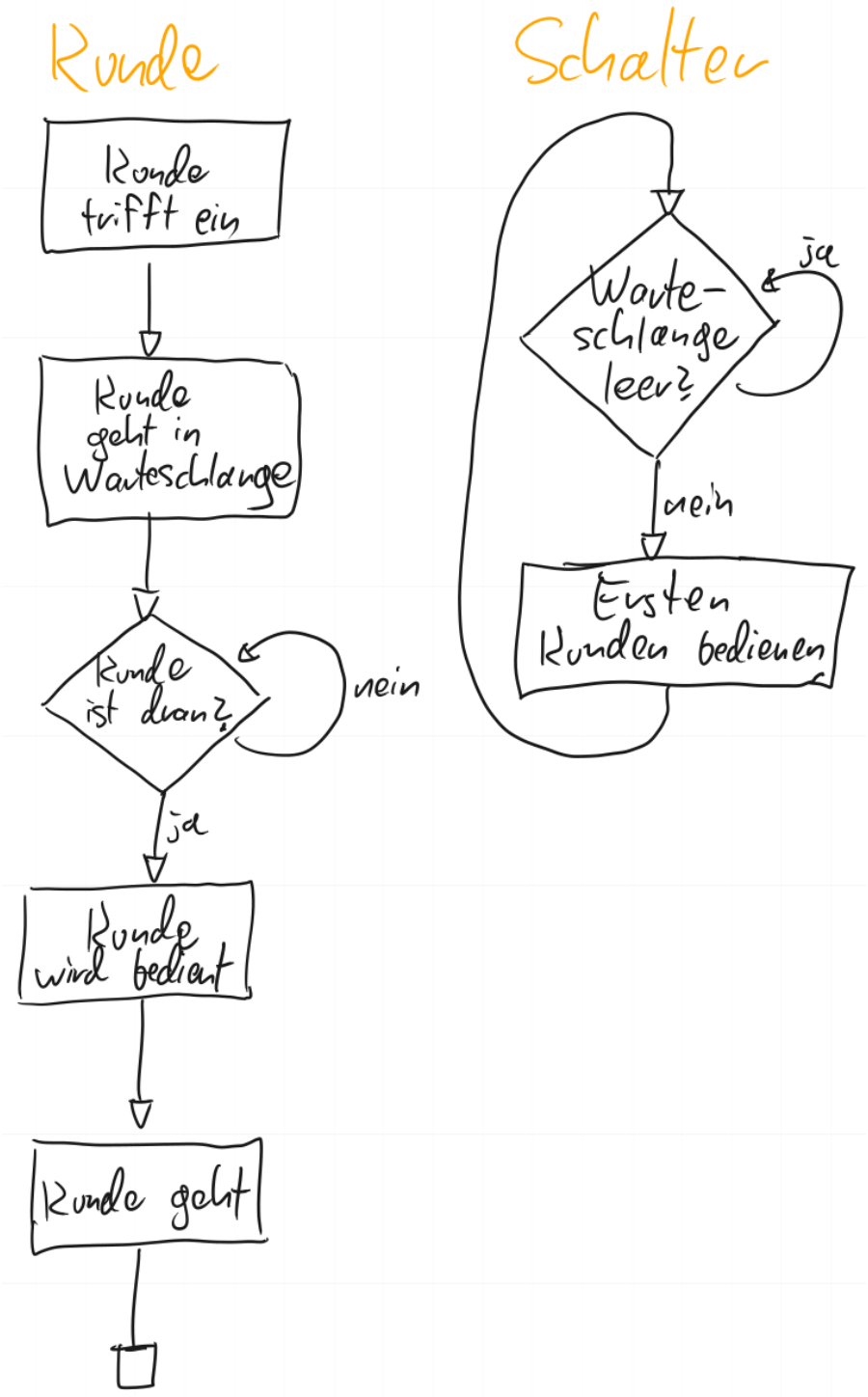

Figure 1: Real-world lifecycles

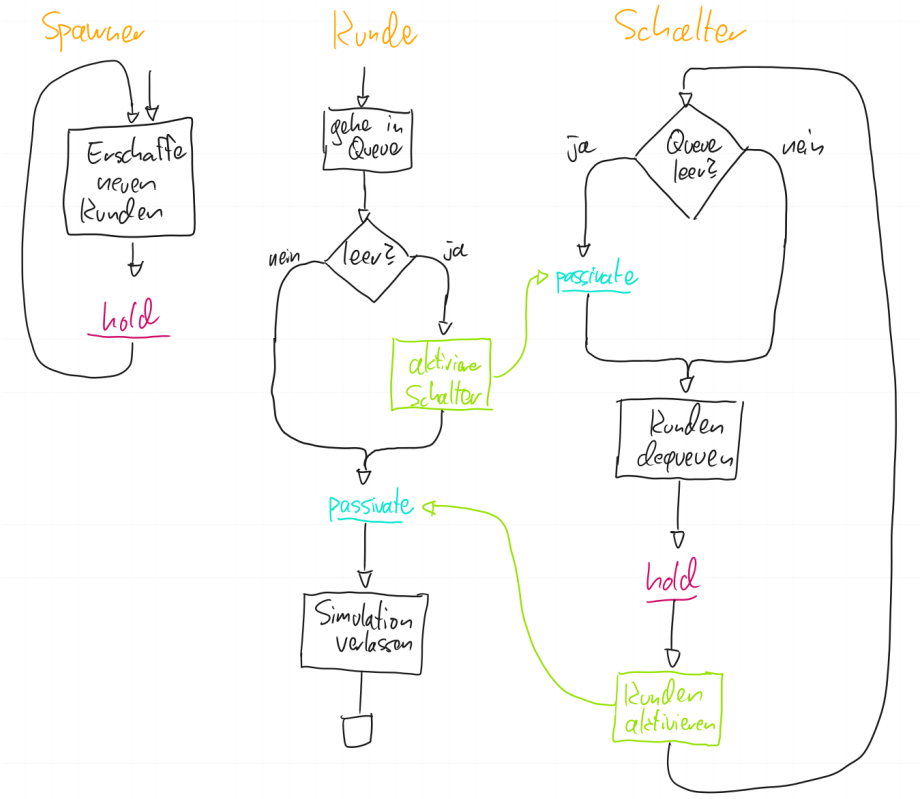

Figure 2: Prozessorientierte SSQ

3. Model validation: try to show that the model's behaviour corresponds to the real system in relevant aspects, e.g. through empirical observation (observed measures are close enough to data produced by the model)
4. Model experimentation and analysis: run experiments on the model and interpret the obtained results based on some theory about the real system

DESMO-J hilft beim zweiten Schritt.

Um das Modell zu repräsentieren, gibt es eine Klasse `desmoj.core.simulator.Model`. Das Modell wird durch die Klasse `desmoj.core.simulator.Experiment` "ausgeführt". Die `Experiment` Klasse steuert die Parameter des Durchlaufes und erzeugt einen Report am Ende.

In der `Model` Klasse müssen einige Methoden implementiert werden:

- `init()`: Setup, Modell-Komponenten (als Attribute von der `Model` Klasse) instanzieren, ...
- `doInitialSchedules()`: initiale Events erstellen bzw. initiale dynamische Komponenten (siehe unten) instanzieren
- `description()`: Kurze textuelle Beschreibung des Modells
- `main()`: `Model` und `Experiment` instanzieren und verbinden (`model.connectToExperiment(ex)`), Parameter wie `stopTime` setzen, Experiment starten

Nach Durchführung des Experiments werden HTML-Reports generiert. Deren Inhalt hängt von den verwendeten *data collectors* ab. Diese sind in weiteren von DESMO-J gelieferten Klassen wie Zufallszahlen-Generatoren mit unterschiedlichen Verteilungen, Queues usw. integriert: durch Verwendung jener Klassen werden automatisch Statistiken erhoben.

Es gibt auch ein Log/Trace, wo Textausgaben über den Zustand des Modells ausgegeben werden. Dabei gibt es zwei Levels `Trace` und `Debug`. Durch `Experiment.tracePeriod(int time)` wird festgelegt, für welche Dauer `time` ein Log gemacht werden soll. Der Default ist 0 - Logging ist langsam.

Die *dynamischen Modellkomonenten* wären z.B. eintreffende Kunden in einer Single Server Queue. Diese werden je nach Modellierungsstil unterschiedlich implementiert:

Ereignisorientierte Modellierung

Relevante Klassen:

- `desmoj.core.simulator.Entity`
- `desmoj.core.simulator.AbstractEvent`

Die Entitäten werden als Subklassen von `Entity` mit entsprechenden zusätzlichen Attributen etc. umgesetzt.

`AbstractEvent` hat weitere Subklassen:

- `Event` (an event that refers to one entity - e.g. a customer leaving the model)
- `EventOf2Entities` (an event that refers to two entities - e.g. a server entity completing the request of a customer entity)
- `EventOf3Entities` (an event that refers to three entities - e.g. a machine entity assembles two parts represented by two other entities)
- `ExternalEvent` (an event that doesn't refer to a entity, but e.g. to the model as whole - e.g. arrival of new custome who is not yet represented by an entity)

Diese haben eine `eventRoutine()` Methode, die je nach Subklasse 1, 2, 3 oder 0 Entites als Parameter nimmt. In dieser Methode sollen state changes der betreffenden Entites, Queueing neuer Events etc. durchgeführt werden.

Statistiken über z.B. die durchschnittliche Bearbeitungsdauer an einem Schalter kann man erheben, indem man die Schalter in eine `Frei-` und `Besetzt`-Queue steckt, wenn sie frei oder besetzt werden. Die Queues werden dann Statistiken über die Aufenthaltsdauer der Schalter in ihnen festhalten.

Prozessorientierte Modellierung

Relevante Klassen:

- `desmoj.core.simulator.SimProcess`

Subklassen implementieren eine `lifeCycle()`-Methode, die zu den Aktivitäten des Prozesses korrespondiert. Mit `hold()` und `passivate()` kann gewartet werden.