
Supervisory Control and Simulation of a Bottling Station

Moor/Wittmann/FGDES, 2016-09-12

Abstract: This report addresses an experimental set-up in supervisory control. It (a) provides documentation for the simulator of a bottling station, including its discrete-event interface; (b) proposes a discrete-event plant model and a specification; (c) outlines a controller synthesis procedure; (d) gives instructions on how to run the controller with the plant simulation in closed-loop configuration. The overall set-up was motivated by an in-house benchmark, originally proposed by Jan Richter, Siemens AG, 2012. Back then it was agreed that the simulator should be made freely available. The main purpose of this report is to document the technological set-up and to invite the interested reader to develop and evaluate alternative control strategies. *This report is a follow-up on (Wittmann, 2012). It may receive updates/improvements in due course, please contact us for an up-to-date revision.*

Keywords: discrete-event systems, physical plant simulation, closed-loop simulation

1 Physical Plant and Simulation

The plant under consideration consists of a filling system and a transport system in a configuration shown by Figure 1. In its intended mode of operation, containers are fed to the plant from the right to pass sensor S1, are then separated by the separator V, in order to enter the roundabout at sensor S2. The roundabout can be controlled to progress in 30°-steps and to forward containers to the filling station. There, containers can be filled with products A and/or B. Using the roundabout

again, containers will be picked up by the conveyor belt and exit the station via sensor S3. A high-level factory management system places orders for products and expects acknowledgement when they have been produced.

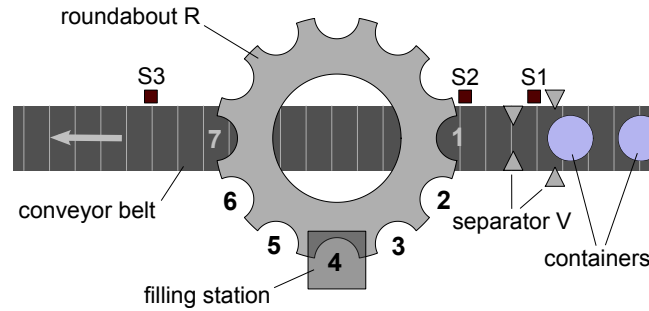


Figure 1: physical layout of the bottling station

FGDES provides the software simulation FtcPlant to mimic the behaviour of the bottling station. It is distributed with the `faudes_application`-package, freely available at www.rt.eei.uni-erlangen.de/FGdes/download.html, either pre-compiled for Linux, Mac OSX or MS Windows, or, for the case of compatibility issues, in source form. Let us know if you experience problems in obtaining/running the simulator. To this end, the simulation is started by simple double-click to show the physical layout of the bottling station. Actuators can be triggered manually, e.g, clicking the roundabout will progress it by 30° . For the subsequent controller design, it is instructive to play along and to operate the plant manually.

Discrete-event interface: textual definition

For the purpose of this report, the bottling station can be adequately modelled as a discrete-event system, with relevant events defined in textual form by Table 1.

Table 1: discrete-event interface of the bottling station

Event	Documentation
S1, S2, S3	Sensors to indicate the arrival of a bottle at the respective position, i.e., at the separator, at the roundabout and on exit. These events can be considered as edges on boolean valued signals associated with the input reading of the respective sensor.

V	The event V triggers an abstract command to let one bottle pass the entry stopper. The command is considered completed, when the bottle arrives at the roundabout S2. If no bottle is present at S1 the command is silently ignored. Likewise, as long as there is a pending V-command, subsequent V events are ignored. This behaviour is considered to be implemented by a low-level software component.
R, r	The event R triggers an abstract command to turn the roundabout by 30°, i.e., by one position. Completion is acknowledged by the event r. A corresponding low-level software component implements R by turning on the motor of the roundabout and by waiting for a key-switch to indicate arrival at the next position. By the design of this software component, one must not issue another R-command before completion.
RR	This is the same as R-command, but progresses the roundabout by 90°. Using consistently RR instead of R effectively reduces the capacity of the roundabout. This is a last resort when synthesis fails due to exhaustion of computational resources.
A, B	Abstract commands to fill either product type A or B into the container at the filling station. The filling command is implemented by a low-level software component which by design uses half of the container capacity. I.e., to fill a bottle, two commands are to be issued. The filling station will jam on filling with no container or with filling above the container capacity. The filling station will also jam if another filling process is started before the most recent filling process is completed; see also the below events a and b.
a, b, f	Acknowledgement of completion of the recent filling process A and B, resp., or, indication of an error. Here, error does not refer to the machine being jammed, but to some other unmodelled malfunction; e.g., out of supply. In the simulation, the error will only occur if the user clicks the fail-button. This feature is meant to investigate fault-tolerant control and is ignored for the remainder of this report.

In our overall set-up, the bottling station interacts with a factory management system that places orders, i.e., requests a container being filled by a certain recipe. Recipes under consideration are “only A”, “only B” or a “mixture of A and B”, specified as orders P1, P2 and P3, respectively. To this end, the simulator provides push-buttons for a human operator to place orders. It also logs acknowledgements for inspection. Events relevant for the factory management system are defined in textual form by Table 2.

Table 2: discrete-event interface for the factory management

Event	Documentation
P1, P2, P3	High-level product request for the respective product type. In the simulation, these events are triggered by push-buttons.
PI	High-level event issued by the supervisor to indicate that the recent product request has been rejected by the supervisor. Preferably, the supervisor shall not reject product requests unless necessary; e.g., due to machine break down.
p1, p2, p3	High-level events to acknowledge delivery of the respective product to the factory management. These events are to be issued by the supervisor when the respective container leaves the station at S3.
pi	High-level event to indicate the delivery of an unspecified product. Preferably, the supervisor shall not deliver unspecified products unless necessary; e.g., due to machine break down. This feature is not relevant for the purpose of this report.

Discrete-event interface: access via TCP/IP

In order to form a closed-loop system, the plant simulator `FtcPlant` exports its discrete-event interface via the TCP/IP based `Simplenet` protocol. The latter is implemented as part of the `libFAUDES` software library. For basic tests, the library includes the command-line tool `iomonitor` to connect to `FtcPlant` and to monitor events:

```
> cd ~/FTCNOMINAL
> ~/LIBFAUDES_BIN/iomonitor ftcsuper.dev
```

Here, `~/FTCNOMINAL` is to be substituted by the location of the example data that comes with the simulator `FtcPlant` and `~/LIBFAUDES_BIN` should point to the executable `iomonitor`.¹

The C++ sources of `iomonitor` (included with `libFAUDES`) may serve as a starting point for the development of a controller for the plant at hand. For other programming languages that provide methods to connect/listen to TCP/IP ports (e.g. Matlab or Python), a minimal implementation of the `Simplenet` protocol really is simple. If you want to go that path, please contact us for further directions/documentation.

¹We understand that command-line tools are old-fashioned — but this one is straight forward — if you are unfamiliar with the command line, consult your local IT-expert for a 5 minutes tutorial.

In the two subsequent sections we propose to design a controller that is represented by finite automata. For this situation, `libFAUDES` includes a ready-to-use software component to interpret finite automata and to synchronise event execution with the plant simulator `FtcPlant`; for detailed instructions see Section 4.

2 Transport System

Supervisor control theory, as originally introduced by P.G. Ramadge and W.M. Wonham, provides a framework for the design of feed-back controllers that operate discrete-event systems. From a pragmatic point of view, the framework suggests to represent plant capabilities and closed-loop requirements by finite automata and to use this input data for the computation of the controller dynamics that make ends meet. For a concise and self-contained introduction by the original authors see (Ramadge and Wonham, 1989). A textbook presentation is available in Chapter 3 of (Cassandras and Lafortune, 2008).

We start our discussion with the transport system and organise the plant model by individual components, each referring to one place on the roundabout; i.e. `Place1`, `Place2`, ..., `Place7` as illustrated by Figure 1.

Feed containers to Place1

The nominal event sequence `S1-V-S2` corresponds to a feed to `Place1`. An `S1` event during feed must be recorded for subsequent feed operations. During a feed operation, the roundabout must not move.

We propose the plant components `LFeed` and `LRaCtrl`. The former relates the events `S1`, `V` and `S2`, while the latter relates `R` and `r`; see Figure 2.

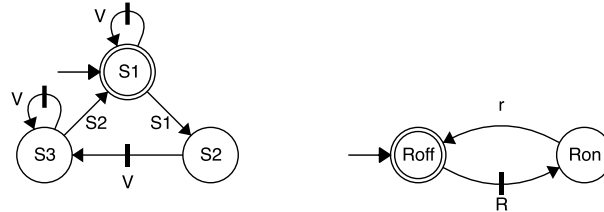


Figure 2: plant components `LFeed` (left) and `LRaCtrl` (right)

Likewise, we propose the specification `EFeedA` to relate `S1` and `V`, and the specification `EFeedB` to relate `S2`, `V`, `R` and `r`; see Figure 3.

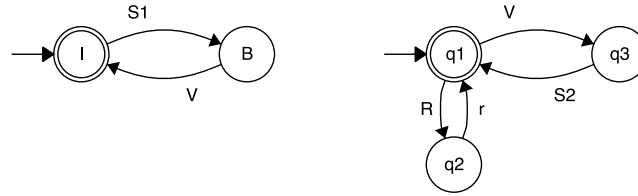


Figure 3: specification components *EFeedA* (left) and *EFeedB* (right)

Organise transport from Place1 to Place7

Except for Place1, the individual places do not have sensors. However, when a container arrives at Place7, the roundabout must not turn until the container has safely exited the station via S3. Thus, a nearby specification would refer to a non-existent sensor that indicates when Place7 becomes occupied. To state such a specification, we introduce the unobservable events $o\{i\}$, $i = 2, \dots, 7$, with the semantics of the non-existing sensors for individual places to become occupied. More precisely, if the predecessor Place $\{i-1\}$ is occupied and the roundabout turns, $o\{i\}$ is defined to occur just before the roundabout stops with r . The component that models the feed to Place $\{i\}$ is named LPlace $\{i\}$, with the special case for LPlace2 where arrival at the predecessor Place1 is reported by the actual sensor S2; see Figure 4.

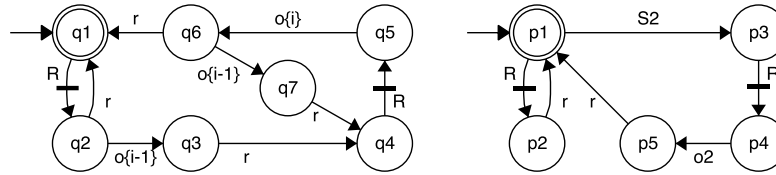


Figure 4: plant model LPlace $\{i\}$ for transport to Place $\{i\}$ (left: $i \neq 2$, right $i = 2$)

Since the additional sensor events are declared unobservable, synthesis for partial observation will virtually figure how to keep track and provide us a supervisor that can be implemented in the absence of the missing sensor information, while it is perfectly fine to refer to the additional events in the specification.

We can now model the exit sensor S3 by relating o_7 , R and S3 and specify, that R must not occur between o_7 and S3, i.e., that containers will safely leave Place7 and exit the station. The respective automata LExit and EExit are given in Figure 5. Here, EExit specifies that the plant component LExit must not attain the dedicated error state Err, and, hence, we do not need to model its future behaviour (i.e., we do not need to discuss whether containers that pass Place7 by the roundabout are dropped or whether they re-appear at S2; neither do we need to be concerned with how many containers may be on the conveyor belt between Place7 and S3).

We finalise the transport system by the two more specifications given in Fig-

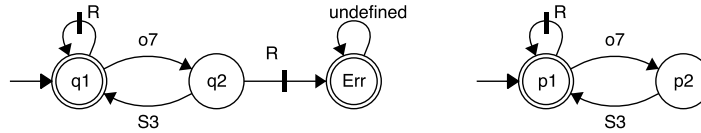


Figure 5: components *LExit* (left) and *EExit* (right)

ure 6. *ELazy* is cosmetic and tracks whether or not the roundabout is empty to prevent idle rotation. The intention of *EProcl* is to make the unobservable event *o4* visible by the additional observable event *X* and thereby provide an interface for the control of the filling station. In principle, there are two possible outcomes of this approach. If our intuition is right and the supervisor turns out able to track occupancy of *Place4* accurately, the closed loop will actually reach a state that corresponds to the report state *Rep* and therefore must enable *X* to attain a marked state. Since *X* is not related to a physical event, we can adjust execution semantics such that enabling *X* amounts to executing *X*. On the other hand, if we got it wrong, the synthesis procedure will implicitly prevent *o4* to avoid blocking. This can be validated easily.

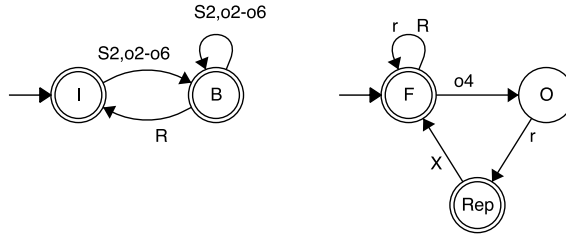


Figure 6: specification components *ELazy* (left) and *EProcl* (right)

Controller Synthesis

The example data includes all of the above plant components and specifications (in *libFAUDES* file-format with file-names matching the identifiers used in this report) and the *luafaudes*-script *transport.lua* to organise controller synthesis. The script implements a straightforward monolithic design along the following stages.

- (i) Compute the parallel composition of all plant components to form an over-all plant model:

$$LTransport = LFeed \parallel LRaCtrl \parallel LPlace1 \parallel \dots \parallel LPlace7 \parallel LExit.$$

This amounts to 1947 reachable states.

- (ii) Compute the parallel composition of all specification components to form an overall specification:

$$ETransport = EFeedA \parallel EFeedB \parallel EExit \parallel ELazy \parallel EProcl.$$

This amounts to 44 reachable states.

- (iii) Apply inverse projection to the overall alphabet to obtain matching alphabets. For the plant, this amounts to self-looping with the additional event X, which is regarded controllable for the synthesis procedure. For the specification, this amounts to self-looping with all unobservable events except for o4;
- (iv) Compute a relatively closed, controllable and observable sublanguage of the specification. Since all unobservable events are uncontrollable, observability matches prefix-normality and we can go for the supremal closed-loop behaviour; see e.g. (Lin and Wonham, 1988; Cho and Marcus, 1989). The script reports 1520 reachable states and we refer to this result as KTransport.
- (v) Project to the observable alphabet to obtain a basis for the implementation of a supervisor. After state minimisation, the script reports 608 states. This is the main result of this section and we refer to it as HTransport.

Subsequent design stages for the filling process will refer to the events X to enable filling, R to prevent progressing the roundabout during filling, and S3 to issue acknowledgement. Given the marking chosen for our plant model, non-blocking control requires that a supervisor must not prevent the roundabout from becoming empty. Thus, an abstraction suitable for the design of non-blocking supervisors must track whether the roundabout is empty. When considering natural projections as abstractions, this suggests the minimum high-level alphabet $\Sigma_{hi} = \{S2, R, X, S3\}$. The provided script computes the projection to the latter alphabet. After state minimisation, the resulting state count amounts to 192. We did not go through the automaton in detail, however, the state count makes sense. This result is referred to as HTransAbs. Although well motivated by intuition, it remains the question whether there is a formal guarantee that any supervisor designed for HTransAbs will be applicable to the actual plant HTransport. For the example at hand, this guarantee is provided by a test proposed in (Moor, 2014). Alternatively, one could add S1 to the high-level alphabet to obtain a so called natural observer; see Wong and Wonham (1996). The latter abstraction exhibits a state count of 384, effectively encoding one additional place for transit from S1 to S2.

3 Filling System and Factory Management

For the filling process we begin with a model to relate the events A , B , a and b ; see the plant component $LProc$ in Figure 7. Neither of the four events is related to the transport system, and, hence, the plant component $LProc$ can be shuffle-composed with the abstract closed-loop model $HTransAbs$ to extend the overall-plant accordingly.

To ask for a supervisor that coordinates transport, product types and filling, we introduce the two specifications $EProcA$ and $EProcB$; see again Figure 7. Here, specification $EProcA$ prevents the roundabout to start while filling is in progress. Each one of the three recipes is enabled by the newly introduced high-level events $D1$, $D2$ and $D3$, which are used to interface the process with the factory management. As with the interface event X from the previous section, the $D_$ events are considered controllable.

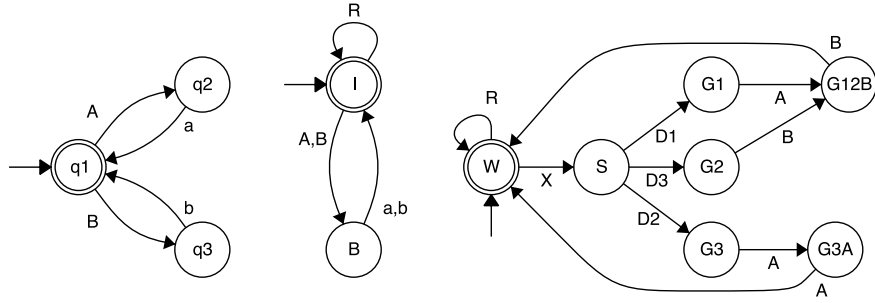


Figure 7: filling system $LProc$ with specifications $EProcA$ and $EProcB$ (left to right)

We propose to synthesise a controller right at this stage to effectively get rid of the low-level filling events A , B , a and b and to reduce the relevant state count. Along the same line of thought as in the previous section, the `luafaudes` script `process.lua` implements the below stages.

- (i) Obtain an overall plant model and an overall specification by

$$LProduct = HTransAbs \parallel LProc$$

$$EProduct = EProcA \parallel EProcB$$

- (ii) Augment $LProduct$ by $D_$ -self-loops.
- (iii) Compute the supremal controllable sublanguage, denote the realisation by $KProduct$. The script reports a state count of 896.
- (iv) Test whether $\Sigma_{hi} = \{S2, R, D1, D2, D3, S3\}$ is an adequate alphabet to obtain an abstraction by natural projection. As it turns out, the natural observer condition is satisfied. After state minimisation, the realisation exhibits 192 states. This result is referred to a $HProduct$.

We did not inspect the automata in detail, but we expect from the state count that HProduct is effectively identical to HTransport with the X-transitions substituted by D_-transitions.

Acknowledgement and request buffers

The factory management places orders P1, P2 and P3 when the container enters the bottling station and expects acknowledgement when a container leaves the station. This can be arranged by two FIFO-buffers.

The order buffer is fed P_- events and outputs them by enabling the oldest corresponding D_- event. The depth of this buffer should be three or four, depending how one reads “orders placed on container entry”. Here, P_- events are considered uncontrollable plant events which on overflow are rejected from the buffer by a P1 event. Figures 8 shows the additional plant component LReq and the order buffer EReqBuf for a reduced variety of products and a reduced depth.

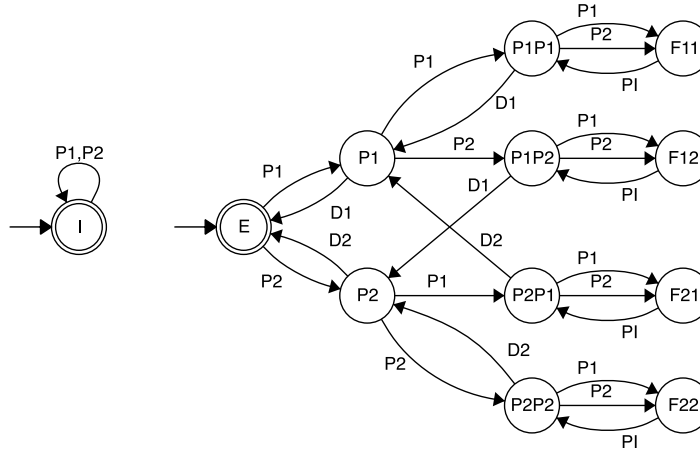


Figure 8: request buffer EReqBuf (two products, depth 2)

The acknowledgement buffer is fed D_-events and outputs them by enabling the corresponding oldest p_- event. Here, we disable p_-events altogether except after S3. Since the supervisor controls the execution of p_-events, appropriate execution priorities ensure prompt acknowledgement. Figures 9 shows the acknowledgement buffer and the additional specification, again, with for a reduced variety of products and a reduced depth.

The actual state counts for the plant at hand with three different products are 202 and 121 for an acknowledgement buffer with depth 4 and for a request buffer with depth 3, respectively. The provided example data includes a luafaudes-script

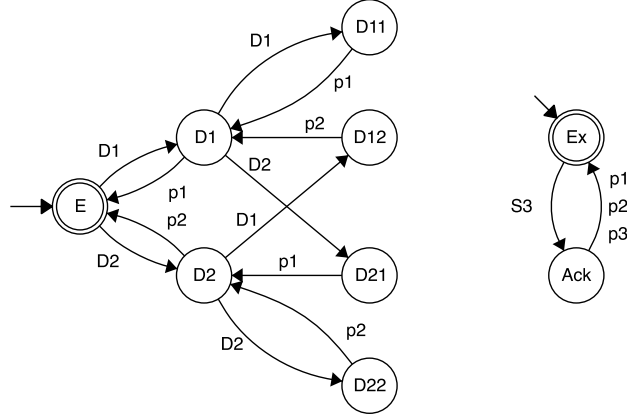


Figure 9: acknowledge buffer *EAckBuf* (two products, depth 2) and exit specification *EExit*

to generate the automata representations in a systematic manner. We do not see an alternative for the order buffer. For the acknowledgement buffer, however, one may instead augment the transport system model by tracking the container contents for the places $\text{Place}\{i\}$ for $i = 4, 5, 6, 7$.

For controller synthesis, we start with the acknowledgement mechanism since it refers to $S3$ and we may get rid of this event once acknowledgement has been implemented. The provided script organises the design as follows.

- (i) Use *HProduct* as abstract plant model and compose the specification *EAcknow*:

$$\begin{aligned} \text{LAcknow} &= \text{Hproduct} \\ \text{EAcknow} &= \text{EAckBuf} \parallel \text{EExit} . \end{aligned}$$

- (ii) Augment *LAcknow* by p_self -loops.
- (iii) Compute the supremal controllable sublanguage, denote the realisation by *KAcknow*. The script reports a state count of 4480.
- (iv) An abstraction of *KAcknow* for the purpose at hand only needs to represent the fact that any $D_$ -event will become eventually enabled. Thus, $\Sigma_{hi} = \{D1, D2, D3\}$ would be a first guess for a high-level alphabet. This, however, leads to a single state which in the setting of plain supervisory control would allow for supervisors that disable the three products altogether and, hence, provoke a blocking situation. The next best choice is to add the uncontrollable sensor event $S2$, i.e., we use $\Sigma_{hi} = \{S2, D1, D2, D3\}$ to obtain the projection *HAcknow* with a state count of 10. It passes the test provided by Moor (2014). It is, however, not a natural observer. The script demonstrates that a natural observer can be obtained by adding another event, namely R .
- (vi) Using *HAcknow* as plant, and applying the usual procedure for the spec-

ification `EReqBuf`, we obtain the supremal closed-loop behaviour with realisation `KRequest` at a state count of 2020. It turns out identical to the product of plant and specification. Hence, we can alternatively use the specification as a supervisor.²

The overall supervisor consists of the components `HTransport`, `KProduct`, `KAcknow` and `KRequest`, to be executed based on parallel-composition semantics with some refinements required to accommodate the technological set up.

4 Closed-Loop Configuration

We give instructions on how to run a closed-loop simulation using `simfaudes` to implement the controller. `simfaudes` is a command-line tool to simulate the behaviour of finite automata with the option to synchronise with external hard- or software. The tool is part of the `libFAUDES` software package, with documentation at www.rt.techfak.fau.de/FGdes/faudes/reference/simulator_index.html. For convenience, `simfaudes` is also distributed together with the plant simulator `FtcPlant`.

For the purpose at hand, a configuration of `simfaudes` consists of two files. The first file refers to the supervisor components that are to be simulated and to an attributed list of events. Here, the attributes are used to resolve ambiguities, i.e., when more than one event is enabled, which one to execute and at which physical time to do so. To this end, the provided example data includes the configuration file `ftcsuper.sim` with the below contents.

Simulator configuration `ftcsuper.sim`

```
<Executor>

<Generators>
<!-- run my supervisors -->
" htransport.gen "
" kproduct.gen "
" kacknow.gen "
" krequest.gen "
</Generators>

<SimEvents>
<!-- prefer feed over roundabout -->
V <Priority> 10 </Priority>
R <Priority> 5 </Priority>
```

²There is a subtle catch here: buffer overflow requires `P1` to preempt `P1`, `P2` and `P3`. The provided script circumvents this issue by effectively taking `P_`-events as controllable. Thus, the result requires verification in that `P1`, `P2` and `P3` are indeed only disabled in favour of `P1`.

```

<!-- do not miss high-level feedback -->
p1 <Priority> 20 </Priority>
p2 <Priority> 20 </Priority>
p3 <Priority> 20 </Priority>
</SimEvents>

</Executor>

```

The generators to simulate are given as filenames to reside in the filesystem next to the configuration file. They have been synthesised with the scripts outlined in the previous sections and are included with the example data. The default priority of each event is zero, if more than one event is enabled the highest priority wins. The proposed configuration imposes no timing constraints, transitions are executed immediately.

The second configuration file is optional and sets up how `simfaudes` is meant to synchronise with external devices. For the use case at hand, we configure `simfaudes` to synchronise events with the plant simulator `FtcPlant` via the TCP/IP based `Simplenet` protocol. This is done with the additional device configuration file `ftcsuper.dev`.

```

Device configuration ftcsuper.dev

<SimplenetDevice name="Supervisor">

  <TimeScale value="1000"/>
  <ServerAddress value="localhost:40001"/>
  <Network name="FtcLoop">
    <Node name="Plant"/>
    <Node name="Supervisor"/>
  </Network>

  <EventConfiguration>

    <!-- physical actuator/sensor events -->
    <Event name="V" iotype="output"/>
    <Event name="R" iotype="output"/>
    <Event name="RR" iotype="output"/>
    <Event name="r" iotype="input"/>
    <Event name="S1" iotype="input"/>
    <Event name="S2" iotype="input"/>
    <Event name="S3" iotype="input"/>
    <Event name="A" iotype="output"/>
    <Event name="a" iotype="input"/>
    <Event name="B" iotype="output"/>
    <Event name="b" iotype="input"/>
    <Event name="f" iotype="input"/>

    <!-- factory management events -->
    <Event name="P1" iotype="input"/>
    <Event name="P2" iotype="input"/>
  </EventConfiguration>
</SimplenetDevice>

```

```

<Event name="P3" iotype="input"/>
<Event name="p1" iotype="output"/>
<Event name="p2" iotype="output"/>
<Event name="p3" iotype="output"/>
<Event name="I" iotype="output"/>

</EventConfiguration>
</SimplenetDevice>

```

Events tagged with `iotype="output"` will be executed by `simfaudes` as soon as they are enabled and according to their respective priority. The plant simulation `FtcPlant` will connect to `simfaudes` on TCP port 40001 to receive notifications whenever `simfaudes` executes an output-event. You can connect to the same port (by e.g. `telnet` or `nc`) to observe the notifications. Vice versa, `simfaudes` will never execute `iotype="input"` unless being notified by a server. On start-up, `simfaudes` will search for servers via UDP broadcast. It will discover that `FtcPlant` provides notifications on TCP port 40000 and will subscribe for relevant input events.

To run `simfaudes` with the provided configuration files, enter the following at the command prompt.

```

> cd ~/FTCNOMINAL
> ~/LIBFAUDES_BIN/simfaudes -dr -d ftcsuper.dev ftcsuper.sim

```

Again, `~/FTCNOMINAL` is to be substituted by the location of the example data that comes with `FtcPlant` and `~/LIBFAUDES_BIN` should point to the executable `iomonitor`.

The `libFAUDES` GUI `DESTool` also provides means to simulate the supervisor in closed-loop configuration. Figure 10 shows `DESTool`'s animation-tab with projectfile `ftcsuper.pro` loaded. `DESTool` is distributed as developer preview via the FGDES homepage with documentation available at www.rt.techfak.fau.de/FGdes/destool.

Summary

FGDES distributes the animated plant simulation `FtcPlant` of a bottling station to serve as a test case for design methods in supervisory control. The simulator was originally motivated by an in-house benchmark, however, we believe that it may be useful in general and therefore provide documentation by this report. For demonstration purposes, the report also provides a basic solution in that it proposes component models, specifications and a controller design. If you plan to use our simulator, do not hesitate to ask for technical support. And please forward your

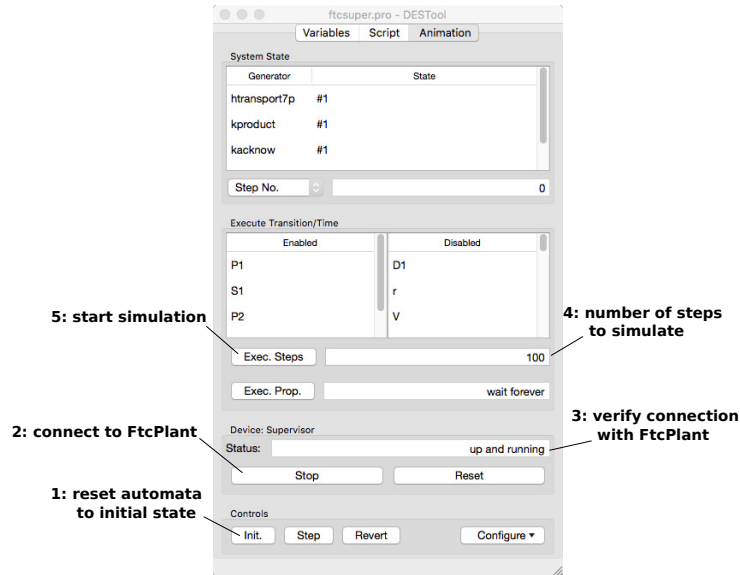


Figure 10: running the supervisor by DESTool

controller design, which we can package with a future revision of FtcPlant to promote your solution.

References

- C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008.
- H. Cho and S. I. Marcus. On supremal languages of classes of sublanguages that arise in supervisor synthesis problems with partial observation. *Maths. of Control, Signals & Systems*, 2:47–69, 1989.
- F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44:173–198, 1988.
- T. Moor. Natural projections for the synthesis of non-conflicting supervisory controllers. In *Workshop on Discrete Event Systems (WODES)*, Paris, 2014.
- P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

Th. Wittmann. Abstraction based controller design for discrete event systems: an application example. *Interner Bericht Nr. 8*, 2012.

K. C. Wong and W. M. Wonham. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 1996.