Karpfen Language, Runtime, and IDE Reference Documentation

Syntax, Semantics, UI

The karpfen project

https://karpfen.io

Commit 3b6029f of 16 November 2026

About this document. This is the reference of karpfen: the modeling languages KMeta, KModel and KStates, the runtime that executes the models, and karpfen IDE. It is generated from the reference chapters in the repositories karpfen-IDE, karpfen-dsl-tools and karpfen-runtime and has the text of the documentation at https://karpfen.io/docs/. The chapters and their numbers are those of the repositories.

Contents

1 Getting started **6**
- 1.1 What karpfen is . . . 6
- 1.2 What you need . . . 7
- 1.3 Starting the IDE . . . 7
  - 1.3.1 From an installer . . . 7
  - 1.3.2 From the sources . . . 8
- 1.4 The lamp example . . . 8
- 1.5 Looking around . . . 9
- 1.6 Running the lamp . . . 9
- 1.7 Sending events . . . 10
- 1.8 Observing and pulling . . . 10
- 1.9 Stopping the run . . . 11
- 1.10 Changing the model . . . 11
- 1.11 Trying a bridge . . . 11
- 1.12 Closing and cleaning up . . . 12

2 Modeling languages **13**
- 2.1 KMeta . . . 15
  - 2.1.1 Types . . . 15
  - 2.1.2 Properties . . . 16
  - 2.1.3 Relations . . . 16
  - 2.1.4 Lists . . . 16
  - 2.1.5 The root type . . . 17
  - 2.1.6 Rules of a metamodel . . . 17
  - 2.1.7 Event definitions . . . 17
  - 2.1.8 Behaviour of a type . . . 18
- 2.2 KModel . . . 18
  - 2.2.1 Objects . . . 19

2.2.2 Keys . . . 19
2.2.3 Property values . . . 19
2.2.4 Embedded objects . . . 20
2.2.5 References . . . 20
2.2.6 Rules of a model . . . 21
2.3 KStates . . . 21
2.3.1 The state machine . . . 22
2.3.2 States . . . 22
2.3.3 Actions . . . 26
2.3.4 Transitions . . . 31
2.3.5 Events . . . 33
2.3.6 Macros . . . 34
2.3.7 Histories . . . 35
2.3.8 Tick limits . . . 36
2.3.9 Breakpoints . . . 36
2.3.10 Optional features . . . 37
2.3.11 State machine management and lifecycle . . . 37
2.3.12 Pages . . . 40
2.3.13 Compound states and container states . . . 43
2.4 Checks . . . 45
2.4.1 What is checked when . . . 46
2.4.2 Syntax . . . 46
2.4.3 KMeta . . . 46
2.4.4 KModel . . . 47
2.4.5 KStates . . . 48
2.4.6 Project . . . 51
2.5 Using the library . . . 55
2.5.1 Adding the library to a build . . . 55
2.5.2 Reading a text . . . 56
2.5.3 Throwing converters . . . 58
2.5.4 Printing back to text . . . 59
2.5.5 Building models in code . . . 60
2.5.6 Optional features . . . 61
2.5.7 Verifying a project . . . 61
2.5.8 Flattening a composition . . . 62
2.5.9 Logging . . . 63

3 Execution **64**
3.1 Execution semantics . . . 65
3.1.1 Example state machine . . . 65
3.1.2 Ticks . . . 67
3.1.3 Active states . . . 68
3.1.4 Transitions . . . 69
3.1.5 Actions . . . 71
3.1.6 EVAL and macros . . . 75
3.1.7 Events . . . 76
3.1.8 Splits and joins . . . 77
3.1.9 Histories . . . 78
3.1.10 Tick limits . . . 79

3.1.11 Forcing a state . . . 80
3.1.12 Errors in a tick . . . 81
3.1.13 State machine management and lifecycle . . . 81
3.1.14 Pages . . . 85
3.1.15 Where the engine goes . . . 91
3.2 Runtime features . . . 95
3.2.1 Switching features . . . 95
3.2.2 History . . . 96
3.2.3 TickLimit . . . 96
3.2.4 TickByTick . . . 96
3.2.5 Breakpoint . . . 97
3.2.6 EventRecording . . . 99
3.2.7 EventInjection . . . 100
3.2.8 EventReplayController . . . 100
3.2.9 TrueParallel . . . 101
3.3 Running the runtime . . . 102
3.3.1 Requirements . . . 103
3.3.2 Running on your machine . . . 103
3.3.3 Running in Docker . . . 103
3.3.4 Configuration . . . 104
3.3.5 Trace log . . . 105
3.3.6 Stopping the runtime . . . 108
3.4 HTTP API . . . 108
3.4.1 Conventions . . . 108
3.4.2 Environments . . . 109
3.4.3 Uploads . . . 110
3.4.4 Rejected uploads . . . 112
3.4.5 Settings . . . 113
3.4.6 Clients . . . 113
3.4.7 Running . . . 114
3.4.8 Observation . . . 115
3.4.9 Health . . . 117
3.4.10 Features . . . 117
3.4.11 Whole session . . . 118
3.5 WebSocket . . . 118
3.5.1 Connecting . . . 119
3.5.2 Sending events . . . 119
3.5.3 Pushed messages . . . 120
3.5.4 Revisions . . . 123
3.5.5 Keeping a copy of the model . . . 123
3.5.6 Delivery . . . 123
3.6 Examples . . . 123

4 The IDE **126**
4.1 Projects . . . 129
4.1.1 Kinds of files . . . 129
4.1.2 Environment descriptors . . . 130
4.1.3 Folders without a descriptor . . . 133
4.1.4 Files that belong to no context . . . 133

4.1.5 The project tree . . . 133
4.1.6 The project file . . . 134
4.1.7 Language features . . . 135
4.1.8 Settings and logs outside the project . . . 136
4.2 Text editor . . . 136
4.2.1 Colours and marks . . . 136
4.2.2 Completion . . . 137
4.2.3 Hover . . . 137
4.2.4 Go to definition . . . 138
4.2.5 Outline . . . 138
4.2.6 Problems . . . 139
4.2.7 Find and replace . . . 139
4.2.8 Saving and changes outside the IDE . . . 139
4.2.9 Editing a descriptor . . . 139
4.2.10 Keys . . . 139
4.3 Diagrams . . . 140
4.3.1 Views . . . 141
4.3.2 Reading a statechart . . . 141
4.3.3 Reading a class or object diagram . . . 142
4.3.4 Editing a statechart . . . 142
4.3.5 Editing a class diagram . . . 148
4.3.7 Code cells . . . 151
4.3.8 Arranging by hand . . . 151
4.3.9 Renaming between diagram and text . . . 152
4.3.10 While a model runs . . . 152
4.3.11 Limits . . . 153
4.4 Validating . . . 153
4.4.1 Checks while you type . . . 153
4.4.2 What a validation adds . . . 153
4.4.3 Starting a validation . . . 154
4.4.4 The dialog . . . 154
4.4.5 The group Global . . . 154
4.4.6 Validation before a run . . . 155
4.5 Running . . . 155
4.5.1 The runtime container . . . 155
4.5.2 Starting a run . . . 155
4.5.3 The run dialog . . . 156
4.5.4 The Runtime tool window . . . 156
4.5.5 Stopping a run . . . 162
4.5.6 A failed run . . . 163
4.5.7 Several runs . . . 165
4.5.8 Captured logs . . . 165
4.6 Debugging . . . 165
4.6.1 Switching breakpoints on . . . 166
4.6.2 Setting breakpoints . . . 166
4.6.3 The debugger line . . . 166
4.6.4 What a pause shows . . . 167
4.6.5 Breakpoints of definitions . . . 167
4.6.6 Breakpoints in pages . . . 167

4.6.7 Limits . . . 167
4.7 Bridges . . . 167
4.7.1 The bridge file . . . 168
4.7.2 From message to event and back . . . 169
4.7.3 Settings . . . 169
4.7.4 The bridge form . . . 171
4.7.5 Starting bridges with a run . . . 172
4.7.6 The bridges of a run . . . 172
4.7.7 Trying the HTTP bridge . . . 172
4.8 Command line . . . 173
4.8.1 Commands . . . 173
4.8.2 Starting it . . . 173
4.8.3 Validating . . . 174
4.8.4 Running . . . 175
4.8.5 What a run reads from the project . . . 178
4.8.6 Exit statuses . . . 178

1 Getting started

karpfen describes a system as a few text files and runs it. The example of this chapter is a lamp with a wall switch: you open it in the IDE, run it, send events to it and connect it to HTTP. Keys are written for macOS first and for Windows and Linux second, as in Cmd+S / Ctrl+S.

1.1 What karpfen is

karpfen has three parts.

Part	What it is	Reference
The modeling languages	KMeta for the types of a domain, KModel for the objects, KStates for the state machines that change the objects.	2 Modeling languages
The runtime	A server that runs a model with its state machines in ticks. Events come in from outside, and clients watch the model change.	3 Execution
The IDE	A desktop program that edits the files, checks them while you type, draws them as diagrams, runs them in its own runtime and shows the run.	4 The IDE

A system in karpfen is a folder of these files.

File	What it holds
Metamodel, `.kmeta`	The types of the domain with their properties and relations.
Event metamodel, a second `.kmeta`	One type per event. The properties of a type are the payload of its event.
Model, `.kmodel`	The objects of the domain, each with a key such as `lamp`.
State machine, `.kstates`	The behaviour of one object. Entering a state runs actions that change the model. Transitions wait for events and conditions.
Fragment, `.frag.kstates`	A page of a state machine: a piece of behaviour that a container state of another state machine runs as its inner states, see 2.3.12 Pages.
Environment descriptor, `.kenv.json`	Which metamodel, events, model and state machines run together, and which object each state machine is attached to.
Bridge, `.kbridge.json`	A connection of the running model to HTTP, MQTT or ROS.

The descriptor and the bridge files belong to the IDE. 4.1 Projects describes them.

1.2 What you need

- Docker Desktop or another local Docker engine, running. The IDE runs every model in a Docker container of its own.
- The repository karpfen-IDE. It holds the lamp example in `examples/lamp`. The folder `examples/pages` holds a lamp composed of fragments, see 4.1.2.5 Fragments.
- To start the IDE from the sources: the submodules of the repository and a JDK 21. Gradle also runs on JDK 17 or newer and downloads a JDK 21 if none is installed. An installed IDE brings its own Java and needs neither.

```
git clone --recurse-submodules https://github.com/karpfenproject/karpfen-IDE.git
cd karpfen-IDE
```

In a clone without the submodules, run `git submodule update --init --recursive`.

1.3 Starting the IDE

The IDE starts from an installer or from the sources.

1.3.1 From an installer

The releases of karpfen-IDE hold one installer per system. GitHub builds them from a release tag.

System	File	Installing	Program
macOS, Apple silicon	`karpfen-ide-<version>-macos-arm64.dmg`	Open the file and drag `karpfen-ide` to Applications.	`/Applications/karpfen-ide.app`
macOS, Intel	`karpfen-ide-<version>-macos-x64.dmg`	Open the file and drag `karpfen-ide` to Applications.	`/Applications/karpfen-ide.app`
Windows	`karpfen-ide-<version>-windows-x64.msi`	Open the file.	`karpfen-ide` in the Start menu, or `C:\Program Files\karpfen-ide\karpfen-ide.exe`
Debian and Ubuntu	`karpfen-ide-<version>-linux-x64.deb`	`sudo apt install ./karpfen-ide-<version>-linux-x64.deb`	`/opt/karpfen-ide/bin/karpfen-ide`

The installers are not signed. macOS and Windows stop an unsigned program, and you have to allow it by hand once.

On macOS the first start fails with a message that macOS could not verify the app. Open System Settings, go to Privacy & Security and click Open Anyway next to the entry for `karpfen-ide`. Then start the app again. If macOS says that the app is damaged, it shows no such entry. Remove the mark of the download in a terminal and start the app again:

```
xattr -dr com.apple.quarantine /Applications/karpfen-ide.app
```

On Windows the installer stops at the message "Windows protected your PC". Click More info and then Run anyway.

On Debian and Ubuntu nothing has to be allowed.

Start the program and open the folder `examples/lamp` of the repository with Open Folder.

1.3.2 From the sources

From the root of the repository, on macOS and Linux:

```
./gradlew :app:ide:run --args="$PWD/examples/lamp"
```

On Windows:

```
gradlew.bat :app:ide:run --args="%CD%\examples\lamp"
```

The path of the folder must be absolute. Without a folder the IDE opens on its welcome view, with Open Folder and the recent folders. The first build takes a few minutes. The command keeps running while the IDE is open.

To build the application as a program of its own, run `./gradlew :app:ide:createDistributable`. It lands in `app/ide/build/compose/binaries/main/app/`. 4.8 Command line names the program file on each system.

1.4 The lamp example

The folder `examples/lamp` holds a small system. A living room has a lamp and a wall switch.

- A press of the switch turns the lamp on and off. The lamp always comes on at the full power of its bulb, 40 W.
- While the lamp is on, a dimmer turns it up and down in steps of 10 W, between 0 W and the full power.
- Switching the lamp off forgets the dimming.
- A smash breaks the lamp at any time and throws away its spare bulb.
- The switch has its own state machine that moves up and down with each press.
- The descriptor sets a tick delay of 500 ms, so each change can be watched, and lets an event wait up to 5 seconds.

File	Content
`lamp.kmeta`	The types `Bulb`, `Switch`, `Lamp` and `Room`.
`lamp-events.kmeta`	The events `press`, `dim` with the payload `bright`, and `smash`.
`lamp.kmodel`	The room `living room` with the lamp `lamp`, its bulbs `bulb` and `spare`, and the switch `wallSwitch`.
`lamp.kstates`	The state machine of the lamp, with the states `off`, `on` with `full`, `brighter` and `dimmer` inside, and `broken`.

File	Content
`switch.kstates`	The state machine of the switch, with the states `down`, `up` and `back down`.
`lamp.kenv.json`	The descriptor. It attaches `lamp.kstates` to `lamp` and `switch.kstates` to `wallSwitch`.
`http.kbridge.json`, `mqtt.kbridge.json`, `ros.kbridge.json`	One bridge of each kind.

1.5 Looking around

After the start of 1.3 Starting the IDE the window shows the folder. The Project tool window lists the descriptor `lamp.kenv.json` at the top, the five model files in the middle and the three bridge files at the bottom. The status bar shows `Problems` without an icon and `Runtime: stopped`.

1. Double-click `lamp.kstates`. The file opens with coloured keywords, and the Outline lists its states and transitions.
2. Click Split in the bar below the editor. The statechart appears next to the text. It starts at the initial state `off`, with `on` and its three inner states. A transition caption reads like `1: switch/press`.
3. Click a state in the diagram. The editor selects its name in the text.
4. Drag the state `broken`. A grid of dots appears, and the state moves on it. In the Project tool window `lamp.kstates` can now be expanded and shows `lamp.kstates.vis` below it.
5. Open `lamp.kmeta` and click Split. A class diagram appears. A line with a filled diamond at `Lamp` leads to `Bulb`, and a line with an open arrowhead leads from `Lamp` to `Switch`.
6. Open `lamp.kmodel` and click Split. An object diagram appears. The box `bulb : Bulb` shows `maxWatts = 40`.
7. In `lamp.kstates`, Cmd+click / Ctrl+click on `Lamp` in `STATEMACHINE ATTACHED TO "Lamp"`. `lamp.kmeta` opens at the type.
8. Open `lamp.kenv.json` and rest the mouse on the key `tickDelayMs`. A tooltip explains the key.
9. In `lamp.kstates`, delete the `{` after `STATE "broken"`. A red wavy underline appears, and the status bar shows `Problems:` with a number and an error icon. Undo with Cmd+Z / Ctrl+Z, and the file is clean again.

1.6 Running the lamp

1. Choose Run > Run, or press Shift+F10. On a Mac keyboard you may have to hold Fn.
2. The dialog Run lists the three bridge files and the box Capture run to file. Leave the boxes unticked and click Run.

The Runtime tool window opens at the bottom and takes three quarters of the window's height. Its second line shows a run named `lamp`, whose label moves from `preparing` to `running`. On the first run its first line reads `Docker: building the runtime image` for a few minutes, and then `Docker: running at 127.0.0.1:` with a port.

The Log pane starts with `lamp: off`, `wallSwitch: down` and the line `Deployment done` with the number of steps. A click on that line shows the steps. The Model pane reads `Observation on`. Click Expand all. The lamp shows `mode = "off"`, the bulb `maxWatts = 40`, `watts = 0` and `lit = false`, and the switch `position = "down"`.

If Docker is not running, the run turns `failed`, and the first line of the Runtime tool window names the problem in red. Start Docker and click Run again.

1.7 Sending events

The Events pane sends events into the run. Choose a domain under 1. Domain, an event under 2. Event, and click the Send button. After each send, look at the Model pane and the Log pane.

Step	Do this	What happens
1	Click the domain `switch`, the event `press`, and `Send press`.	The lamp goes on at full power: `lamp: on > full`, `mode = "on"`, `lit = true`, `watts = 40`. The switch is `up`.
2	Click the event `dim`. Replace `false` in the payload by `3`.	The form shows an error, because `bright` is a boolean, and the Send button is greyed out.
3	Write `false` again and click `Send dim`.	The log shows `lamp on -> dimmer`, and the bulb has `watts = 30`.
4	Send it three more times, then once more.	The bulb goes down to `watts = 0`. The last send changes nothing.
5	Change the payload to `"bright": true` and send it.	`lamp on -> brighter`, `watts = 10`. Each further send adds 10 W, up to 40 W.
6	Dim a step, then send `press` twice.	The lamp goes off with `watts = 0`, then comes on at `watts = 40` again.
7	Click the domain `service` and send `smash`.	`lamp: broken`, `mode = "broken"`, `lit = false`, `watts = 0`. The spare bulb disappears from the tree.
8	Send `press` once more.	The switch moves, but the lamp stays broken.

An event waits for a state machine that takes it. A `dim` sent while the lamp is off takes effect if a press turns the lamp on within 5 seconds. Otherwise it expires.

1.8 Observing and pulling

1. In the Model pane, turn the Observation switch off. The status reads `Observation off`.
2. Send `press` twice. The tree and the active states stay as they are. The simplified log still shows the transitions.
3. Click Pull. The tree shows the current model, with two more entries in `presses`.
4. Turn observation on again. The status reads `live at revision` with a number.

1.9 Stopping the run

Click Stop in the line of the runs, or press Cmd+F2 / Ctrl+F2. The run turns `stopped`, the Model pane keeps the last model, and the Send button is greyed out. The container keeps running, so the next run starts fast. Stop runtime in the first line stops and removes the container.

1.10 Changing the model

Give the lamp a way back from `broken`.

1. Add an event at the end of `lamp-events.kmeta`:

```
type "repair" "Repairs a broken lamp" {
}
```

2. In `lamp.kstates`, add a transition at the end of the `TRANSITIONS` block. Ctrl+M inside the quotes of a state name proposes the states.

```
TRANSITION "broken" -> "off" {
    CONDITION { EVENT("service", "repair") }
}
```

3. Click Run again in the line of the runs, and Run in the dialog Run. A run saves every open file first.
4. Send `smash`, then `repair`, both in the domain `service`. The lamp goes from `broken` back to `off`. The spare bulb stays gone, because nothing creates it again.

If a file has an error, the run does not start, and a dialog lists the errors. 4.4 Validating describes what blocks a run.

1.11 Trying a bridge

1. Click Run again, tick `http.kbridge.json` in the dialog Run and click Run. The Log pane lists the bridge below the deployment.
2. In a terminal, press the wall switch and make the lamp dimmer:

```
curl -X POST http://127.0.0.1:8090/press
curl -X POST http://127.0.0.1:8090/dim -d '{"bright": false}'
```

   The lamp goes on, then down to `watts = 30`, as if the events had been sent in the Events pane.

The other two bridge files need a partner. Without it the bridge shows as failed, and the run goes on.

File	Needs	Does
`mqtt.kbridge.json`	An MQTT broker on `localhost:1883`.	Takes presses on `home/lamp/press` and dim steps on `home/lamp/dim`. Publishes the power on `home/lamp/watts` and the active states on `home/lamp/state`.

File	Needs	Does
`ros.kbridge.json`	A rosbridge server on `localhost:9090`, from the ROS package `rosbridge_suite`.	Takes presses on `/lamp/press` and dim steps on `/lamp/dim`. Publishes the power on `/lamp/watts` and the active states on `/lamp/state`.

4.7 Bridges describes the bridge files and their form.

1.12 Closing and cleaning up

Quit with Cmd+Q, or close the window. The IDE asks about unsaved changes, stops every run and removes the runtime container. To undo your changes to the example, run `git restore examples/lamp` and delete the `.vis` files and the file `.karpfenproject` that the IDE wrote into the folder.

The container is also removed when the IDE is stopped from outside, for example with Ctrl+C on the Gradle command. Only a process that is killed outright leaves it behind.

The image stays, so the next run starts in seconds. To free the disk space, remove it. The first command removes containers left behind by a killed IDE, and prints an error when there are none.

```
docker rm --force --volumes $(docker ps --all --quiet --filter label=io.karpfen.ide.runtime
    ↪ =true)
docker rmi $(docker images --quiet --filter label=io.karpfen.ide.runtime=true)
```

2 Modeling languages

karpfen has three textual languages. KMeta describes the types of a domain: what a lamp, a bulb and a switch are, which values they carry and how they belong together. KModel describes objects of those types and their values. KStates describes a state machine. A state machine is attached to a type and runs on one object of that type, where it reads and changes the model.

File	Language	Content
`.kmeta`	2.1 KMeta	A metamodel: the types of a domain with their properties and relations.
`.kmodel`	2.2 KModel	A model: one root object with the objects it holds, all of types of one metamodel.
`.kstates`	2.3 KStates	One state machine, attached to a type of the metamodel or attachable to several.

A project has one domain metamodel, one model and any number of state machines. A model is read against its metamodel: every object is of a type the metamodel defines, and every value fits a property of that type. A state machine names its type after `ATTACHED TO`. Which object of the model it runs on is decided when the project runs, by the object's key, see 3.3 Running the runtime and 4.5 Running. An attachable state machine has a name instead, and objects get instances of it through their types and through the actions of other state machines, see 2.3.11 State machine management and lifecycle.

A project may have a second `.kmeta` file, the event metamodel. Each of its types is one event, and the properties of the type are the payload the event carries, see 2.1.7 Event definitions. A state machine waits for an event by its domain and its name, and reads the payload of the event that let it change state.

The library checks each text on its own while it reads it, and it checks the state machines of a project against the metamodels and the model. 2.4 Checks lists every problem it reports. 2.5 Using the library shows how to read, check and write the texts from Kotlin code.

The three languages share their lexical rules. Spaces, tabs and line breaks between the parts of a text are free, so the layout of a text is up to the writer. A comment starts with `//` and runs to the end of the line. It may stand wherever a space may stand, also at the end of a line. There is no other kind of comment. Keywords are case sensitive. Names, keys and values are string literals between quotes, and so are numbers and booleans. The only numbers outside quotes are the tick counts of KStates.

Rule	KMeta	KModel	KStates
Quotes of a string	`"`	`"` or `'`	`"`, and `'` only inside `EVAL` code
Spelling of the keywords	lower case	lower case	upper case; the action operations and `WITH` and `AS` also lower case; `reference` only lower case
Quote characters the reader removes from the edges of a value	every `"`	every `"` and `'`	one pair of `'` directly inside the `"`

A string ends on its line: it cannot hold a line break. Inside a string, a backslash and the

character after it belong to the string, so `\"` does not end it. The value keeps both characters as written: `"say \"hi\""` is not turned into `say "hi"`. Since the readers remove quote characters at the edges of a value, a value should not start or end with a quote character. In KStates, `"'on'"` and `"on"` are the same value.

The lamp is the example of this chapter. A room holds a lamp and a wall switch. The lamp holds a bulb and knows the switch it is wired to. The metamodel has four types:

```
// The types of the lamp example.
type "Bulb" "A light bulb that burns with watts, up to maxWatts" {
    prop("maxWatts", "number")
    prop("watts", "number")
    prop("lit", "boolean")
}
type "Switch" "A wall switch" {
    prop("position", "string")
    prop("presses", list("string"))
}
type "Lamp" "A lamp with a bulb, wired to a switch" {
    prop("mode", "string")
    has("bulb", "Bulb")
    knows("control", "Switch")
}
type "Room" "The room the lamp stands in" {
    has("lamp", "Lamp")
    has("switch", "Switch")
}
```

The model is a living room. Every object has a key, such as `"lamp"`, by which a `knows` relation and a run refer to it:

```
make object "living room":"Room" {
    has("lamp") -> make object "lamp":"Lamp" {
        prop("mode") -> "off"
        has("bulb") -> make object "bulb":"Bulb" {
            prop("maxWatts") -> "40"
            prop("watts") -> "0"
            prop("lit") -> "false"
        }
        knows("control") -> "wallSwitch"
    }
    has("switch") -> make object "wallSwitch":"Switch" {
        prop("position") -> "down"
    }
}
```

The state machine of the lamp runs on the object `"lamp"`. It starts in the state `off`. An event `press` in the domain `switch` turns the lamp on, and the next one turns it off again. On entering a state, the state machine writes the mode and the values of the bulb:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            ENTRY {
                SET("mode", "off")
                SET("bulb->lit", "false")
```

```
                SET("bulb->watts", "0")
            }
        }
        STATE "on" {
            ENTRY {
                SET("mode", "on")
                SET("bulb->lit", "true")
                SET("bulb->watts", EVAL { return $(bulb->maxWatts) })
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The paths `"mode"` and `"bulb->watts"` start at the object the state machine runs on, the lamp. `$(bulb->maxWatts)` in the `EVAL` code is the current value of that property, 40 here. The sections of this chapter extend these three texts. The full lamp project is the walkthrough of the IDE, see 1 Getting started.

2.1 KMeta

A KMeta text is a metamodel. It defines types, and each type has properties, which hold values, and relations, which lead to objects of other types. A file has the extension `.kmeta`. The grammar of the language is `src/main/antlr/io/karpfen/dsl/grammar/kmeta/Kmeta.g4` in the karpfen-dsl-tools repository.

2.1.1 Types

A type starts with the keyword `type`, its name and a description, both as strings, and a body in braces. The body holds the rules of the type, one `prop`, `has` or `knows` per rule, in any order. A body may be empty.

```
type "Switch" "A wall switch" {
}
```

The name identifies the type. Models and state machines refer to it, and names are case sensitive. The description is text for the reader. It is required, but it may be the empty string `""`. A text defines one type or more. Between the description and the body, a type may name the state machine its objects behave after, see 2.1.8 Behaviour of a type.

2.1.2 Properties

`prop("key", "type")` gives a type a property: a value of one of the primitive types `string`, `number` and `boolean`. The type names are written in lower case.

```
type "Bulb" "A light bulb" {
    prop("maxWatts", "number")
    prop("watts", "number")
    prop("lit", "boolean")
}
```

A number is a decimal number, such as `40` or `0.5`. A boolean is `true` or `false`. A string is any text. How a model writes these values is described in 2.2.3 Property values.

2.1.3 Relations

A relation leads from an object to an object of another type, or of the same type. There are two kinds.

`has("key", "Type")` embeds an object. The embedded object belongs to the object that holds it: in a model it is written inside its owner, and it exists once, at that place. `knows("key", "Type")` refers to an object that is embedded somewhere else in the model. A model names that object by its key.

```
type "Bulb" "A light bulb" {
    prop("watts", "number")
}
type "Switch" "A wall switch" {
    prop("position", "string")
}
type "Lamp" "A lamp with a bulb, wired to a switch" {
    has("bulb", "Bulb")
    knows("control", "Switch")
}
type "Room" "The room the lamp stands in" {
    has("lamp", "Lamp")
    has("switch", "Switch")
}
```

Here the bulb is part of the lamp, and the lamp refers to the switch, which is part of the room. The target type of a relation must be a type of the metamodel. It may be defined before or after the type that uses it.

2.1.4 Lists

`list("type")` in place of the type makes a property or a relation hold a list. A list keeps the order of its elements and may be empty.

```
type "Lamp" "A lamp" {
    prop("mode", "string")
}
type "Switch" "A wall switch" {
    prop("presses", list("string"))
    knows("lamps", list("Lamp"))
}
```

```
type "Room" "A room with several lamps" {
    has("lamps", list("Lamp"))
    has("switch", "Switch")
}
```

`presses` holds a list of strings, `lamps` of the room embeds a list of lamps, and `lamps` of the switch refers to a list of lamps. A property or relation without `list` holds at most one value.

2.1.5 The root type

The last type of a text is the root type of the metamodel. In the examples of 2.1.3 and 2.1.4 it is `Room`. Write the type of the root object of the model last. The readers do not check that the root object of a model has the root type.

2.1.6 Rules of a metamodel

A metamodel must keep these rules. 2.4.3 KMeta lists the code of each problem.

- The text defines at least one type.
- No two types have the same name.
- No two rules of one type have the same key, whether they are properties, relations or both.
- A `prop` has the type `string`, `number` or `boolean`, written in lower case, or a list of one of them.
- A `has` or `knows` names a type of the metamodel, not a primitive type.
- A `behaves after` names an attachable state machine of the project that lists the type, and stands only in the domain metamodel. The verifier of a project checks this rule, since it needs the state machines.

Every property and relation is optional in a model. KMeta has no default values, no inheritance between types and no numbers of elements other than one and a list.

2.1.7 Event definitions

The events of a project are defined in a second KMeta text, the event metamodel. Each type of it is one event: the name of the type is the name of the event, and the properties and relations of the type are the payload the event carries. An event without payload is a type with an empty body.

```
// The events of the lamp example.
type "press" "Presses the wall switch" {
}
type "dim" "Turns the dimmer one step: brighter when bright is true, dimmer when it is
    ↪ false" {
    prop("bright", "boolean")
}
type "smash" "Breaks the lamp" {
}
```

The event metamodel names no domain. A state machine names the domain together with the event, as in `EVENT("switch", "dim")`, and reads the payload as `event->bright`, see 2.3.5

Events.

The event metamodel is read with the types of the domain metamodel at hand. A payload may therefore embed or refer to a domain type without defining it again, for example `has("position", "Point")` in an event type when `Point` is a type of the domain metamodel. The event types do not become part of the domain metamodel. The last type of an event metamodel is its root type, which has no meaning for events.

2.1.8 Behaviour of a type

`behaves after "name"` between the description and the body of a type gives every object of the type an instance of the attachable state machine of that name: each object of the model when a run starts, and each object that a rule of a state machine creates. The keywords are written in lower case.

```
type "Bulb" "A light bulb" behaves after "glowing" {
    prop("watts", "number")
    prop("lit", "boolean")
}
```

The state machine must be attachable to the type, as in `STATEMACHINE "glowing" ATTACHABLE TO "Bulb"`, see 2.3.1 The state machine. Types without the clause have no state machine of their own unless a run or a rule attaches one. An event type cannot have the clause. 2.3.11 State machine management and lifecycle explains when to use it.

2.2 KModel

A KModel text is a model: objects of the types of one metamodel, with their values. A file has the extension `.kmodel`. The grammar of the language is `src/main/antlr/io/karpfen/dsl/grammar/kmodel/Kmodel.g4` in the karpfen-dsl-tools repository.

The examples of this section are models of the lamp metamodel:

```
type "Bulb" "A light bulb that burns with watts, up to maxWatts" {
    prop("maxWatts", "number")
    prop("watts", "number")
    prop("lit", "boolean")
}
type "Switch" "A wall switch" {
    prop("position", "string")
    prop("presses", list("string"))
}
type "Lamp" "A lamp with a bulb, wired to a switch" {
    prop("mode", "string")
    has("bulb", "Bulb")
    knows("control", "Switch")
}
type "Room" "The room the lamp stands in" {
    has("lamp", "Lamp")
    has("switch", "Switch")
}
```

2.2.1 Objects

`make object "key":"Type" { ... }` makes an object of a type. The braces hold its statements: `prop` statements give values to its properties, `has` statements embed objects, and `knows` statements refer to objects. The statements may come in any order, and an object may have none.

```
make object "wallSwitch":"Switch" {
    prop("position") -> "down"
}
```

A text holds exactly one object at its top, the root object. Every other object is embedded in it, directly or deeper. A complete model starts with an object of the root type of the metamodel, see 2.1.5 The root type. The readers accept a root object of any type of the metamodel, so the small examples of this section start with a lamp, a bulb or a switch.

2.2.2 Keys

The first string of `make object` is the key of the object, its id. A `knows` statement names an object by its key, and a run names the object a state machine runs on by its key. No two objects of a text have the same key, embedded objects included.

The empty key `""` makes an object without an id. Several objects may have the empty key, but no `knows` statement can refer to them.

```
make object "lamp":"Lamp" {
    has("bulb") -> make object "":"Bulb" {
        prop("watts") -> "0"
    }
}
```

2.2.3 Property values

`prop("key") -> "value"` gives the property `key` a value. The value is always written as a string and read by the type of the property.

Type	Value	Examples
`string`	Any text.	`"down"`, `""`
`number`	A number as Java reads a `double`.	`"40"`, `"-2.5"`, `"1e3"`
`boolean`	`true` or `false`, in any case.	`"true"`, `"FALSE"`

```
make object "bulb":"Bulb" {
    prop("maxWatts") -> "40"
    prop("watts") -> "0"
    prop("lit") -> "false"
}
```

A list property gets one statement per element, in the order of the list. A single statement gives a list of one element. A list without a statement is empty.

```
make object "wallSwitch":"Switch" {
    prop("position") -> "down"
    prop("presses") -> "up"
    prop("presses") -> "down"
}
```

A property without a statement has no value in the model.

2.2.4 Embedded objects

`has("key") -> make object ...` embeds a new object in the relation `key`. The embedded object is written in place, with its own statements.

```
make object "living room":"Room" {
    has("lamp") -> make object "lamp":"Lamp" {
        has("bulb") -> make object "bulb":"Bulb" {
            prop("maxWatts") -> "40"
        }
    }
    has("switch") -> make object "wallSwitch":"Switch" {
    }
}
```

The embedded object should have the type the relation declares. A list relation gets one `has` statement per object.

2.2.5 References

`knows("key") -> "id"` refers to the object whose key is `id`. The object may stand anywhere in the model, before or after the statement, and must have the type the relation declares.

```
make object "living room":"Room" {
    has("lamp") -> make object "lamp":"Lamp" {
        knows("control") -> "wallSwitch"
    }
    has("switch") -> make object "wallSwitch":"Switch" {
    }
}
```

A list relation gets one `knows` statement per object it refers to. This metamodel has list relations of both kinds:

```
type "Lamp" "A lamp" {
    prop("mode", "string")
}
type "Switch" "A wall switch" {
    knows("lamps", list("Lamp"))
}
type "Room" "A room with several lamps" {
    has("lamps", list("Lamp"))
    has("switch", "Switch")
}
```

```
make object "hall":"Room" {
```

```
    has("lamps") -> make object "ceiling":"Lamp" {
    }
    has("lamps") -> make object "wall":"Lamp" {
    }
    has("switch") -> make object "hallSwitch":"Switch" {
        knows("lamps") -> "ceiling"
        knows("lamps") -> "wall"
    }
}
```

2.2.6 Rules of a model

A model must keep these rules. 2.4.4 KModel lists the code of each problem.

- Every object is of a type of the metamodel.
- Every `prop` names a property of the type of its object, every `has` an embedding relation (`has` in KMeta) and every `knows` a referring relation (`knows` in KMeta).
- A number property gets a number and a boolean property `true` or `false`.
- A property or relation that is not a list gets at most one statement.
- No two objects have the same key, unless the key is empty.
- A `knows` names the key of an object of the model, and that object has the type of the relation.
- An object embedded with `has` has the type of the relation. Another type is reported as a warning.

2.3 KStates

A KStates text is a state machine. It has states, which carry actions that read and write the model, and transitions, which lead from state to state when their conditions hold. A file has the extension `.kstates` and holds one state machine. The grammar of the language is `src/main/antlr/io/karpfen/dsl/grammar/kstates/Kstates.g4` in the karpfen-dsl-tools repository.

The exact behaviour of each construct at run time is in 3.1 Execution semantics.

The examples of this section run on the lamp metamodel of 2 Modeling languages, in which the room also holds a list of spare bulbs:

```
type "Bulb" "A light bulb that burns with watts, up to maxWatts" {
    prop("maxWatts", "number")
    prop("watts", "number")
    prop("lit", "boolean")
}
type "Switch" "A wall switch" {
    prop("position", "string")
    prop("presses", list("string"))
}
type "Lamp" "A lamp with a bulb, wired to a switch" {
    prop("mode", "string")
    has("bulb", "Bulb")
    knows("control", "Switch")
}
type "Room" "The room the lamp stands in, with spare bulbs" {
```

```
    has("lamp", "Lamp")
    has("switch", "Switch")
    has("spares", list("Bulb"))
}
```

2.3.1 The state machine

A text starts with `STATEMACHINE ATTACHED TO "Type"`, where `Type` is a type of the domain metamodel. The braces after it hold the block `STATES`, then the block `TRANSITIONS`, then the block `MACROS`, in this order. `STATES` and `TRANSITIONS` are required, `TRANSITIONS` may be empty, and `MACROS` may be left out.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
    }
    TRANSITIONS { }
}
```

The state machine runs on one object of the type it is attached to. Every path of the state machine starts at this object, unless it starts with one of the names that 2.3.3.2 Paths lists. Which object of the model it is, a run decides by the key of the object.

A text may instead start with `STATEMACHINE "name" ATTACHABLE TO "Type", ...`. Such an attachable state machine is a definition with a name and one or more types. A run does not attach it by a key. The runtime builds an instance of it for each object that a `behaves after` of its type, an `ATTACH` clause or an `EXCHANGE` names it for, see 2.3.11 State machine management and lifecycle. The name is required, and each type is listed once.

A third form, `STATEMACHINE "name" FRAGMENT`, is a fragment: a piece of a state machine that runs inside a container state of another one, see 2.3.12 Pages.

```
STATEMACHINE "glowing" ATTACHABLE TO "Bulb" {
    STATES {
        INITIAL STATE "dark" { }
    }
    TRANSITIONS { }
}
```

2.3.2 States

`STATE "name" { ... }` defines a state. No two states of a state machine have the same name, whatever their depth. Transitions refer to states by their names.

2.3.2.1 Nesting

A state may hold inner states, which are written inside its braces after everything else of the state. While an inner state is active, the states around it are active too. A transition that starts at an outer state also applies while the state machine is in one of its inner states.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            STATE "full" { }
            STATE "dimmed" { }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "full" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "full" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

A press switches the lamp off from `full` and from `dimmed`, since the transition from `on` applies in both. A transition into a state with inner states enters the state itself and none of its inner states.

2.3.2.2 The initial state

The keyword `INITIAL` in front of `STATE` marks where the state machine starts. Exactly one state of a file carries the mark, at any depth, in a state machine and in a fragment alike. A file without a mark or with two marks has the error `kstates.initial-state-count`.

At the start the state machine enters a stack of states, the outermost first: the start stack. It is built level by level. On the top level it takes the state that is marked or that holds the mark somewhere inside it, and the first state of the level when there is none. It goes on below the chosen state the same way while that state has inner states, and stops at a state without inner states.

So the mark may stand on an inner state. The state machine then starts in the marked state and in every state around it, wherever these stand among their siblings:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        STATE "off" { }
        STATE "on" {
            STATE "full" { }
            INITIAL STATE "dimmed" { }
        }
    }
    TRANSITIONS {
        TRANSITION "dimmed" -> "full" {
            CONDITION { EVENT("switch", "dim") }
```

```
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "off" -> "full" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The start stack is `on`, `dimmed`. `off` is the first state of the top level, but `on` holds the mark, and inside `on` the marked `dimmed` wins over `full`, which stands first.

A marked state with inner states leads on to its first inner state, and on to the first inner state of each level below:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "on" {
            STATE "full" {
                STATE "steady" { }
                STATE "flickering" { }
            }
            STATE "dimmed" { }
        }
        STATE "off" { }
    }
    TRANSITIONS {
        TRANSITION "steady" -> "flickering" {
            CONDITION { EVENT("service", "flicker") }
        }
        TRANSITION "full" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The start stack is `on`, `full`, `steady`.

A container state counts as a state whose inner states are the states of its fragment, see 2.3.12 Pages. When the start stack reaches a container state, because it is marked or because it stands first on the way down, it goes on with the start of the fragment: the state of the fragment that its own mark chooses, by the same rule. The examples use this fragment, which starts in `dimmed`:

```
STATEMACHINE "dimmer" FRAGMENT {
    STATES {
        STATE "full" { }
        INITIAL STATE "dimmed" { }
        FINAL STATE "dark" { }
    }
```

```
    TRANSITIONS {
        TRANSITION "dimmed" -> "full" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "full" -> "dark" {
            CONDITION { EVENT("switch", "dim") }
        }
    }
}
```

A marked container state starts the state machine inside its fragment:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        STATE "off" { }
        INITIAL CONTAINER STATE "on" USE "dimmer" { }
    }
    TRANSITIONS {
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The start stack is `on`, `on/dimmed`. The states of a fragment carry the name of their container state in front, see 2.3.12.3 Names. A container state that the start reaches on the way down goes on the same way:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "on" {
            CONTAINER STATE "dimming" USE "dimmer" { }
            STATE "bright" { }
        }
        STATE "off" { }
    }
    TRANSITIONS {
        TRANSITION "dimming" -> "bright" {
            CONDITION { DONE }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The start stack is `on`, `dimming`, `dimming/dimmed`.

The mark of a fragment belongs to the fragment. The state machine that uses it does not see it, so it does not count as the mark of that state machine and does not pull its start into the

container state. It says where the fragment starts, each time its container state is entered:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        CONTAINER STATE "on" USE "dimmer" { }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The start stack is `off`. A press enters `on` and at once `on/dimmed`, and so does every later press from `off`. Had `on` its inner states written in its braces, the mark on `dimmed` would be the one mark of the file, and the state machine would start in `on`, `dimmed` as in the first example.

The start goes down into a state with inner states written in its braces, and a transition does not: a transition into such a state enters the state itself and none of its inner states, see 2.3.2.1 Nesting. Every entry into a container state, by the start or by a transition, goes on with the start of its fragment. The one exception is a container state with a history that remembers a state of its fragment: it goes on with the remembered state, see 2.3.7 Histories. What runs when the states are entered, and in which tick, is described in 3.1.3 Active states and for every way of entering a state in 3.1.15 Where the engine goes, and, for container states, in 3.1.14 Pages.

2.3.2.3 Parts of a state

A state holds these parts, each of them optional, in this order:

1. `BREAKPOINT { ... }`, see 2.3.9 Breakpoints
2. `TICK LIMIT { ... }`, see 2.3.8 Tick limits
3. `ENTRY { ... }`, see 2.3.3.1 Entry and do blocks
4. `DO { ... }`
5. the inner states

In front of `STATE` stand `INITIAL` and then a history, `SHALLOW HISTORY` or `DEEP HISTORY`, see 2.3.7 Histories, each of them optional. Right before `STATE` may stand `CONTAINER` or `FINAL`, which make a container state or a final state with fewer parts, see 2.3.12 Pages.

2.3.3 Actions

2.3.3.1 Entry and do blocks

`ENTRY { ... }` holds the actions a state runs once when the state machine enters it. `DO { ... }` holds the actions it runs in every tick in which it is the innermost active state. A block holds action rules, `IF IN SCOPE` blocks and `WITH` blocks, which run in the order they are written.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            ENTRY {
                SET("mode", "off")
                SET("bulb->watts", "0")
            }
        }
        STATE "warming up" {
            ENTRY {
                SET("mode", "on")
            }
            DO {
                SET("bulb->watts", EVAL { return min($(bulb->watts) + 10, $(bulb->maxWatts)
    ↪ ) })
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "warming up" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

While the lamp is in `warming up`, every tick gives the bulb 10 W more, up to its maximum. A state has no block for leaving it.

2.3.3.2 Paths

A path names a property or a relation. It is a string of keys joined by `->`, and it starts at the object the state machine runs on: `"mode"` is the mode of the lamp, `"bulb->watts"` the watts of its bulb and `"control->position"` the position of the switch the lamp knows. A path follows `has` and `knows` relations alike. A path that goes on after a list relation goes on from the first object of the list. Spaces around a key are ignored, so a key that is used in paths should not contain `->`.

A path starts somewhere else if its first key is one of these names:

First key	Where the path starts
`event`	At the payload of the event in scope, see 2.3.5 Events.
The name of a `WITH` block around it	At the result of its macro, see 2.3.3.7 WITH blocks.
A parameter of the macro, in the code of a macro	At the value of the parameter, see 2.3.6 Macros.

2.3.3.3 Action rules

An action rule names an operation and its operands in parentheses. The first operand is always a string.

Rule	What it does
`SET("path", value)`	Writes the value into a property that is not a list.
`APPEND("path", value)`	Adds the value at the end of a list property.
`SETLIST("path", value)`	Replaces a list property with the list that its `EVAL` or `MACRO` returns.
`DROPLIST("path")`	Empties a list property.
`SETOBJ("source", "relation", target)`	Sets the relation of the object at `source`, which holds one object, to the target.
`APPENDOBJ("source", "relation", target)`	Adds the target to the list relation of the object at `source`.
`DROPOBJ("path")`	Removes the object at the path from the model, with every reference to it.
`EVENT("domain", name)`	Raises an event of the name in the domain, without payload.
`EXCHANGE("path", name)`	Ends the state machine of the object at the path and attaches an instance of the attachable state machine of the name.

The source of `SETOBJ` and `APPENDOBJ` is a path to an object; `"self"` and `""` name the object the state machine runs on. The target of a `has` relation must be a new object, which a `MACRO` or an `EVAL` builds. The target of a `knows` relation must be an object of the model: a path to it, or a macro that returns `reference("Type")`. Every operation except `EVENT` may also be written in lower case, such as `set`.

A `SETOBJ` or `APPENDOBJ` on a `has` relation may end with `ATTACH name`, after its parentheses. The clause names the attachable state machine that the object the rule creates gets. `EXCHANGE("path", name)` replaces the state machine of an object of the model. In both, the name is a value: a string is the name itself, not a path, and an `EVAL` or a `MACRO` returns it as a string. `ATTACH` may also be written `attach`. 2.3.11 State machine management and lifecycle shows them at work.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            ENTRY {
                SET("mode", "off")
                DROPLIST("control->presses")
            }
        }
        STATE "on" {
            ENTRY {
                SET("mode", "on")
                APPEND("control->presses", "on")
                EVENT("lamp", "switched on")
            }
        }
        STATE "reset" {
            ENTRY {
                SETLIST("control->presses", EVAL { return ["reset"] })
```

```
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "reset" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "reset" -> "off" { }
    }
}
```

A state machine attached to the room changes relations. It wires the lamp to the switch, adds a new spare bulb and removes the bulb from the lamp:

```
STATEMACHINE ATTACHED TO "Room" {
    STATES {
        INITIAL STATE "rewiring" {
            ENTRY {
                SETOBJ("lamp", "control", "switch")
                APPENDOBJ("self", "spares", MACRO("new bulb", "lamp->bulb->watts"))
                DROPOBJ("lamp->bulb")
            }
        }
    }
    TRANSITIONS { }
    MACROS {
        MACRO "new bulb" {
            TAKES("watts", "number")
            RETURNS("Bulb")
            DEFINITION {
                EVAL { return {"watts": $(watts)} }
            }
        }
    }
}
```

2.3.3.4 Values

The last operand of a rule, its value, has one of three forms.

Form	Example	Value
A string	`"40"`	For `SET` and `APPEND`, the text read as the type of the property: a number, `true` or `false`, or any text. For `SETOBJ` and `APPENDOBJ`, a path to an object. For `EVENT`, the name of the event.
`EVAL { code }`	`EVAL { return $(bulb->maxWatts) / 2 }`	What the code returns, see 2.3.3.5 EVAL code.
`MACRO("name", "path", ...)`	`MACRO("half", "bulb->maxWatts")`	What the macro returns, see 2.3.6 Macros.

A number or a boolean is written as a string too, as in `SET("bulb->watts", "0")`. Without

the quotes it is a syntax error.

2.3.3.5 EVAL code

`EVAL { ... }` holds code that the runtime runs as Python. It can be the value of a rule, a clause of a condition or the definition of a macro. The code returns its result with `return`.

The runtime prepares the code before it runs it. Each `$(path)` becomes the current value at the path, so `$(bulb->watts) + 10` adds 10 to the watts the bulb has at this moment. Each `@name(arguments)` becomes a call of the macro `name`, which takes one argument per `TAKES` of the macro:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "dimmed" {
            ENTRY {
                SET("bulb->watts", EVAL {
                    full = $(bulb->maxWatts)
                    return @half(full)
                })
            }
        }
    }
    TRANSITIONS { }
    MACROS {
        MACRO "half" {
            TAKES("watts", "number")
            RETURNS("number")
            DEFINITION {
                EVAL { return $(watts) / 2 }
            }
        }
    }
}
```

The code may run over several lines. The library reads it as KStates text up to the closing brace, so the braces in the code must pair up. Braces and `//` inside string literals of the code do not count. Outside string literals, `//` starts a KStates comment that runs to the end of the line: `return a // b` loses `// b`, and a closing brace after it on the same line. Write `int(a / b)` for the integer division and `#` for a Python comment.

2.3.3.6 IF IN SCOPE blocks

`IF IN SCOPE("path", ...) { ... }` runs the actions inside only if every path it names can be read at this moment. A path that starts with `event` can be read if an event is in scope and its payload has that field. Any other path can be read if it leads to a value from the object the state machine runs on. A state that the state machine enters with and without an event can read the payload this way:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            ENTRY {
                SET("bulb->watts", EVAL { return $(bulb->maxWatts) })
                IF IN SCOPE("event->bright") {
                    SET("bulb->lit", EVAL { return $(event->bright) })
```

```
                }
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

Several paths are separated by commas, and the block runs only if all of them can be read. Blocks may be nested.

2.3.3.7 WITH blocks

`WITH MACRO("name", ...) AS "binding" { ... }` runs the macro once and gives its result the name `binding` for the actions inside the block. The code of an `EVAL` inside reads the result as `$(binding)`, or a value of an object result as `$(binding->key)`, and the arguments of a macro call inside may start with the name too. `with` and `as` may also be written in lower case.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "on" {
            ENTRY {
                WITH MACRO("brightest", "bulb->maxWatts") AS "best" {
                    SET("bulb->watts", EVAL { return $(best) })
                    SET("bulb->lit", EVAL { return $(best) > 0 })
                }
            }
        }
    }
    TRANSITIONS { }
    MACROS {
        MACRO "brightest" {
            TAKES("limit", "number")
            RETURNS("number")
            DEFINITION {
                EVAL { return $(limit) }
            }
        }
    }
}
```

Both rules see the same result, even when the macro returns another value each time it runs.

2.3.4 Transitions

`TRANSITION "from" -> "to" { ... }` leads from one state to another. The braces may hold a `BREAKPOINT` block and then a `CONDITION` block. A transition without a `CONDITION` block fires whenever the runtime tries it. A transition may lead from a state to the state itself.

2.3.4.1 Conditions

`CONDITION { ... }` holds one clause or more. The transition fires only if all clauses hold. The runtime checks them from top to bottom and stops at the first one that fails.

Clause	Holds when
`EVENT("domain", "name")`	An event of this name waits in this domain, see 2.3.5 Events.
`VALUE("path")`	The boolean property at the path is true, or the string property at the path is `true` in any case. `VALUE("true")` always holds and `VALUE("false")` never.
`EVAL { code }`	The code returns true.
`DONE`	The container state the transition leaves has reached a final state of its fragment, see 2.3.12 Pages.

A condition has at most one `EVENT` clause. Write it first: the clauses after it can read the payload of the event.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "on" {
            STATE "dimmer" {
                ENTRY {
                    SET("bulb->watts", EVAL { return $(bulb->watts) - 10 })
                }
            }
        }
    }
    TRANSITIONS {
        TRANSITION "on" -> "dimmer" {
            CONDITION {
                EVENT("switch", "dim")
                EVAL { return not $(event->bright) }
                EVAL { return $(bulb->watts) - 10 >= 0 }
            }
        }
    }
}
```

The lamp gets dimmer on a `dim` event whose payload has `bright` set to false, unless the bulb is already at 0 W.

2.3.4.2 Order and NOT LOOPING

The runtime tries the transitions of the innermost active state first and those of the states around it after. It tries the transitions of one state in the order they are written, and the first one whose condition holds fires. The order of the `TRANSITIONS` block is therefore a priority.

`NOT LOOPING` after the target keeps a transition from firing twice in a row. When the runtime skips it for this reason, it tries the next transition. It has an effect only on a `TRANSITION`, not on a split or a join.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "on" {
            DO {
```

```
                SET("bulb->watts", EVAL { return $(bulb->watts) - 1 })
            }
        }
        STATE "check" { }
    }
    TRANSITIONS {
        TRANSITION "on" -> "on" NOT LOOPING {
            CONDITION { VALUE("bulb->lit") }
        }
        TRANSITION "on" -> "check" { }
        TRANSITION "check" -> "on" { }
    }
}
```

While the bulb is lit, the state machine enters `on` again, but not twice in a row: in between it takes the transition to `check`.

2.3.4.3 Splits and joins

A split leads from one state into two or more states that are active side by side, each in a branch of its own. A join leads from such states back into one state.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "bulb test" { }
        STATE "switch test" { }
        STATE "tested" { }
    }
    TRANSITIONS {
        SPLIT "off" -> "bulb test", "switch test" {
            CONDITION { EVENT("service", "test") }
        }
        JOIN "bulb test", "switch test" -> "tested" { }
    }
}
```

`SPLIT "from" -> "a", "b", ...` names two targets or more, each once. `JOIN "a", "b", ... -> "to"` names two sources or more, each once. Both take `NOT LOOPING`, a `BREAKPOINT` block and a `CONDITION` block like a `TRANSITION`. The runtime takes a split only while no branches are active, and a join only when each of its sources is active in a branch of its own; a join comes before the transitions of the branches. See 3.1 Execution semantics for the branches.

2.3.5 Events

An event has a domain and a name, and it may carry a payload. The domain and the name are any strings: in the lamp, presses arrive in the domain `switch` and the smash in the domain `service`. Events come from the `EVENT` actions of state machines and from outside the runtime, see 3.4 HTTP API, 3.5 WebSocket and 4.7 Bridges.

The clause `EVENT("domain", "name")` holds when an event of this domain and name waits. The event that fires the transition is then in scope: the clauses after the `EVENT` clause, the state the transition leads to and the states inside it read its payload with paths that start with `event`, such as `event->bright` or `$(event->bright)`. The event stays in scope while that state is active.

The payload of an event has the type of the same name in the event metamodel, see 2.1.7 Event definitions. The event `dim` of the lamp carries `bright`:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "on" { }
        STATE "brighter" {
            ENTRY {
                SET("bulb->watts", EVAL { return $(bulb->watts) + 10 })
            }
        }
    }
    TRANSITIONS {
        TRANSITION "on" -> "brighter" {
            CONDITION {
                EVENT("switch", "dim")
                VALUE("event->bright")
            }
        }
        TRANSITION "brighter" -> "on" { }
    }
}
```

2.3.6 Macros

A macro is a named piece of code with typed parameters. The block `MACROS` at the end of a state machine holds the macros, each written as `MACRO "name" { ... }` with these parts in this order:

1. `TAKES("parameter", type)` for each parameter, none or more
2. `RETURNS(type)`, exactly once
3. `DEFINITION { EVAL { code } }`, exactly once

Type	Meaning
`"string"`, `"number"`, `"boolean"`	A primitive value.
`"Type"`	An object of a type of the domain metamodel or of the event metamodel. As a result, a new object.
`list("type")`	A list of values of the type.
`reference("Type")`	Only in `RETURNS`: an object that is already part of the model.

The code reads a parameter as `$(parameter)`, and a value of an object parameter as `$(parameter->key)`. A macro that returns a new object returns a Python dictionary of its values, such as `{"watts": 40}`, see the example of 2.3.3.3 Action rules.

`MACRO("name", "path", ...)` calls a macro as the value of a rule or in a `WITH` block. Each argument is a path, and the macro gets the value at the path. A call passes one argument per `TAKES`. If it passes fewer, the runtime reads each missing one from the path named like its parameter. In the code of an `EVAL`, `@name(arguments)` calls a macro with values, see 2.3.3.5 EVAL code.

```
STATEMACHINE ATTACHED TO "Lamp" {
```

```
    STATES {
        INITIAL STATE "half" {
            ENTRY {
                SET("bulb->watts", MACRO("share", "bulb->maxWatts", "mode"))
            }
        }
    }
    TRANSITIONS { }
    MACROS {
        MACRO "share" {
            TAKES("full", "number")
            TAKES("mode", "string")
            RETURNS("number")
            DEFINITION {
                EVAL { return $(full) / 4 if $(mode) == "eco" else $(full) / 2 }
            }
        }
    }
}
```

2.3.7 Histories

A history makes a state with inner states remember which of them was active when the state machine left it. When the state machine enters the state again, it goes on to the remembered state. `SHALLOW HISTORY` remembers the inner state directly in it, `DEEP HISTORY` the innermost state at any depth. The history is written after `INITIAL` and before `STATE`.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        SHALLOW HISTORY STATE "on" {
            INITIAL STATE "full" { }
            STATE "dimmed" { }
        }
        STATE "off" { }
    }
    TRANSITIONS {
        TRANSITION "full" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "dimmed" -> "full" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The lamp starts at full power. A lamp that was switched off while dimmed comes on dimmed again.

2.3.8 Tick limits

A tick limit bounds how many ticks a state stays active. The block `TICK LIMIT { ... }` holds a delay, a timeout or both, in either order.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            TICK LIMIT {
                DELAY: 2 TICKS
                TIMEOUT: AFTER 20 TICKS TRANSITION TO "off"
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

`DELAY: n TICKS` keeps the state machine in the state until the state was active for more than `n` ticks. `TIMEOUT: AFTER n TICKS TRANSITION TO "state"` leads to the named state once the state was active for more than `n` ticks. Here the lamp stays on for more than two ticks before a press can switch it off, and without a press it goes off once it has been on for more than 20 ticks. A count is a whole number, and `TICK` and `TICKS` are read alike, so `1 TICK` and `5 TICKS` are both fine.

2.3.9 Breakpoints

A breakpoint marks where the debugger of the IDE halts a run, see 4.6 Debugging. In a state, `BREAKPOINT { ENTRY EXIT }` halts when the state machine enters the state, when it leaves it, or both. In a transition, split or join, `BREAKPOINT { START END }` halts at the start of the transition, at its end, or both. A block names one of its two words or both, in either order. In a transition the block comes before the `CONDITION` block.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            BREAKPOINT { EXIT }
        }
        STATE "on" { }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            BREAKPOINT { START END }
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

2.3.10 Optional features

Five parts of KStates are optional features. A tool decides which of them a text may use. The IDE keeps this choice in the settings of a project, see 4.1 Projects. The library allows all of them unless it is told otherwise, see 2.5.6 Optional features.

Feature	Id	Constructs	The problem is reported at
Histories	`history`	`SHALLOW HISTORY`, `DEEP HISTORY`	`SHALLOW`, `DEEP`
Tick limits	`tickLimit`	`TICK LIMIT { ... }`	`TICK`
Breakpoints	`breakpoints`	`BREAKPOINT { ... }` in states, transitions, splits and joins	`BREAKPOINT`
Macros	`macros`	the `MACROS` block and every `MACRO(...)` call, also in a `WITH` block	`MACROS` once for the block, `MACRO` of each call
Splits and joins	`parallel`	`SPLIT` and `JOIN`	`SPLIT`, `JOIN`

When a feature is switched off, the text is still read as before, but each of its constructs is an error with the code `kstates.feature-off`. Switching a feature on again turns these errors off. An `@name(...)` call in `EVAL` code is not a construct of the feature `macros`.

At run time, histories, tick limits and breakpoints take effect only where the runtime has the matching feature switched on, see 3.2 Runtime features. Without it, the state machine runs as if they were not written. Dynamic attachment, see 2.3.11 State machine management and lifecycle, and pages, see 2.3.12 Pages, are no optional features: a text may always use fragments, container states, final states and `DONE`, and the runtime always understands them.

2.3.11 State machine management and lifecycle

An attached state machine runs on one object that a run names by its key. That fits a lamp that exists from the start. It does not fit objects that come and go while the model runs, such as the spare bulbs of the room, which a state machine creates. For them a project writes attachable state machines and says in the model and in the actions which object gets which one.

Use the attached form, `ATTACHED TO "Type"`, for the state machine of one known object, chosen when the project runs. Use the attachable form, `"name" ATTACHABLE TO "Type", ...`, for behaviour that many objects share, objects that exist from the start as well as objects that rules create. Each object gets an instance of its own, which starts in the initial state. List several types when the same behaviour fits objects of several types; every path of the state machine must then be valid from each of them.

`behaves after "name"` in the header of a KMeta type gives every object of the type an instance of the named state machine: each object of the model at the start of a run, and each object a rule creates. Use it for the usual behaviour of a type.

```
type "Bulb" "A light bulb that burns with watts, up to maxWatts" behaves after "glowing" {
    prop("maxWatts", "number")
    prop("watts", "number")
    prop("lit", "boolean")
```

```
}
type "Switch" "A wall switch" {
    prop("position", "string")
    prop("presses", list("string"))
}
type "Lamp" "A lamp with a bulb, wired to a switch" {
    prop("mode", "string")
    has("bulb", "Bulb")
    knows("control", "Switch")
}
type "Room" "The room the lamp stands in, with spare bulbs" {
    has("lamp", "Lamp")
    has("switch", "Switch")
    has("spares", list("Bulb"))
}
```

Every bulb glows while it has watts:

```
STATEMACHINE "glowing" ATTACHABLE TO "Bulb" {
    STATES {
        INITIAL STATE "dark" {
            ENTRY { SET("lit", "false") }
        }
        STATE "lit" {
            ENTRY { SET("lit", "true") }
        }
    }
    TRANSITIONS {
        TRANSITION "dark" -> "lit" {
            CONDITION { EVAL { return $(watts) > 0 } }
        }
        TRANSITION "lit" -> "dark" {
            CONDITION { EVAL { return $(watts) == 0 } }
        }
    }
}
```

A bulb that blinks switches its light on every tick:

```
STATEMACHINE "blinking" ATTACHABLE TO "Bulb" {
    STATES {
        INITIAL STATE "off" {
            ENTRY { SET("lit", "false") }
        }
        STATE "on" {
            ENTRY { SET("lit", "true") }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" { }
        TRANSITION "on" -> "off" { }
    }
}
```

`ATTACH name` after a `SETOBJ` or `APPENDOBJ` on a `has` relation names the state machine of the object the rule creates, in place of the one its type behaves after. Use it when a created object should start with other behaviour than the usual one. `EXCHANGE("path", name)` ends the

state machine of an object of the model and attaches an instance of the named one, which starts in its initial state. Use it to change the behaviour of an object that exists already. On an object without a state machine, `EXCHANGE` attaches the named one.

The state machine of the room adds a blinking spare bulb when the switch is pressed, and makes the bulb of the lamp blink on the next press:

```
STATEMACHINE ATTACHED TO "Room" {
    STATES {
        INITIAL STATE "waiting" { }
        STATE "stocked" {
            ENTRY {
                APPENDOBJ("self", "spares", MACRO("new bulb", "lamp->bulb->maxWatts"))
    ↪ ATTACH "blinking"
            }
        }
        STATE "party" {
            ENTRY {
                EXCHANGE("lamp->bulb", "blinking")
            }
        }
    }
    TRANSITIONS {
        TRANSITION "waiting" -> "stocked" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "stocked" -> "party" {
            CONDITION { EVENT("switch", "press") }
        }
    }
    MACROS {
        MACRO "new bulb" {
            TAKES("watts", "number")
            RETURNS("Bulb")
            DEFINITION {
                EVAL { return {"maxWatts": $(watts), "watts": $(watts)} }
            }
        }
    }
}
```

The name after `ATTACH` and the second operand of `EXCHANGE` are values in the three forms of 2.3.3.4 Values: a string is the name itself, and `EVAL { return $(mode) }` takes the name from the model, or from a `WITH` binding by its name. The verifier checks a name written as a string: that an attachable state machine of the project has it, and that the state machine lists the type of the object. A name that an `EVAL` or a `MACRO` computes is checked when the action runs. `ATTACH` on a `knows` relation is an error, since that relation links an object that exists already; `EXCHANGE` is the rule for such an object. 2.4 Checks lists the codes.

What happens at run time, when an instance starts, in which order the instances of a tick run, and what becomes of the state machine of an object that is removed, is described in 3.1.13 State machine management and lifecycle.

2.3.12 Pages

A state machine grows long when it holds every step of a lamp, its self test included. Pages split it: a fragment holds a piece of behaviour, such as the self test, in a file of its own, and a container state of another state machine runs the fragment as its inner states. The state machine and the fragments it uses, directly or through other fragments, form a composition, which the library flattens into one state machine before it runs.

2.3.12.1 Fragments and final states

A fragment starts with `STATEMACHINE "name" FRAGMENT`. It has no type: it runs on the object of the state machine that uses it, so its paths start at that object. Its body is that of every state machine: `STATES`, `TRANSITIONS` and `MACROS`. Exactly one of its states is `INITIAL`, and at least one state at its top level is a final state.

`FINAL STATE "name" { }` marks the end of the fragment. It stands only at the top level of a fragment, it has no actions, no inner states, no tick limit and no history, and it is never `INITIAL`. Its braces may hold a `BREAKPOINT` block and nothing else. No transition, split or join leaves a final state: the state machine stays in it until a transition of the container state leads out.

The fragment `bulb test` lights the bulb at half its power and passes once the bulb is lit, or fails on a smash:

```
STATEMACHINE "bulb test" FRAGMENT {
    STATES {
        INITIAL STATE "powered" {
            ENTRY {
                SET("bulb->watts", MACRO("half", "bulb->maxWatts"))
            }
        }
        FINAL STATE "passed" { }
        FINAL STATE "failed" { }
    }
    TRANSITIONS {
        TRANSITION "powered" -> "passed" {
            CONDITION { VALUE("bulb->lit") }
        }
        TRANSITION "powered" -> "failed" {
            CONDITION { EVENT("service", "smash") }
        }
    }
    MACROS {
        MACRO "half" {
            TAKES("watts", "number")
            RETURNS("number")
            DEFINITION {
                EVAL { return $(watts) / 2 }
            }
        }
    }
}
```

2.3.12.2 Container states and DONE

`CONTAINER STATE "name" USE "fragment" { }` runs the named fragment. A container state has no `ENTRY` and no `DO` block and no inner states of its own, since its fragment does the work. Its braces may hold a `BREAKPOINT` block and a `TICK LIMIT` block, and it may be `INITIAL` and carry a history like any other state. It may stand at any depth: at the top level, inside another state and inside a fragment. A fragment never uses itself, neither directly nor through other fragments.

Entering a container state starts its fragment the way a state machine starts, see 2.3.2.2 The initial state: the state machine enters the marked state of the fragment and the states around it. A container state with a history enters the state it remembers instead, once it remembers one, see 2.3.7 Histories. While the fragment runs, the container state is active around it, so the transitions of the container state apply in every state of the fragment, as for a state with inner states. A transition that leaves the container state ends the fragment.

The clause `DONE` holds when the container state the transition leaves has reached a final state of its fragment. It takes no arguments and stands on a `TRANSITION` or `SPLIT` whose source is a container state:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        CONTAINER STATE "self test" USE "bulb test" {
            TICK LIMIT {
                TIMEOUT: AFTER 50 TICKS TRANSITION TO "off"
            }
        }
        STATE "on" { }
    }
    TRANSITIONS {
        TRANSITION "off" -> "self test" {
            CONDITION { EVENT("service", "test") }
        }
        TRANSITION "self test" -> "on" {
            CONDITION { DONE }
        }
        TRANSITION "self test" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

On a test event the lamp runs the bulb test. When the test has reached `passed` or `failed`, the lamp goes on. A press or 50 ticks end the test early. `DONE` does not tell the final states apart; a fragment that ends in several ways writes the outcome into the model in its states, and the transitions of the container state read it.

On a `JOIN`, `DONE` speaks of every source that is a container state and holds when each of them has reached a final state; sources that are no container states do not count for it. At least one source of the join is a container state. A split into two container states and a join when both are done run two fragments side by side:

```
STATEMACHINE "switch test" FRAGMENT {
    STATES {
```

```
        INITIAL STATE "waiting" { }
        FINAL STATE "pressed" { }
    }
    TRANSITIONS {
        TRANSITION "waiting" -> "pressed" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        CONTAINER STATE "bulb check" USE "bulb test" { }
        CONTAINER STATE "switch check" USE "switch test" { }
        STATE "checked" { }
    }
    TRANSITIONS {
        SPLIT "off" -> "bulb check", "switch check" {
            CONDITION { EVENT("service", "test") }
        }
        JOIN "bulb check", "switch check" -> "checked" {
            CONDITION { DONE }
        }
    }
}
```

2.3.12.3 Names

The states and transitions of a file refer only to the states of that file. A file never names a state inside a fragment it uses, and a fragment never names a state of the state machine that uses it.

After flattening, a state of a fragment carries the name of its container state in front, with a slash: the state `powered` of the bulb test above runs as `self test/powered`. When a fragment uses a fragment again, the names nest, as in `self test/wiring/powered`. The trace of a run, the debugger and the runtime show these names. A fragment used by two container states runs as two copies, and nothing is shared between them; in the second lamp above the bulb test runs as `bulb check/powered`. A flattened name must not equal another state name of the composition, so a state name of a file that contains a slash may clash with one.

Fragments are found by their names alone. The fragments, the attachable state machines and their names form one name space in a project: no two of them have the same name, and the folders and file names of the texts play no part. `ATTACH`, `EXCHANGE` and `behaves after` name attachable state machines only, never a fragment.

2.3.12.4 Macros in a composition

The macros of all files of a composition form one set, so a file may call a macro that another file of its composition defines. A macro of a fragment that runs twice is in the set once. Two macros of one name with different definitions cannot both be in the set, which is an error. A file is easier to read and to reuse when it defines the macros it calls: a call of a macro that only another file defines is a warning at the call.

The checks of a composition are in 2.4.5 KStates for a single file and in 2.4.6.5 Pages across files. What a container state, a final state and `DONE` do at run time is described in 3.1 Execution semantics.

2.3.13 Compound states and container states

A compound state is a state with inner states written in its braces, see 2.3.2.1 Nesting. A container state runs the states of a fragment as its inner states, see 2.3.12 Pages. Both group states under an outer state whose transitions apply in each of them. They differ in where the inner states live and in how far the outer text can reach into them.

	Compound state	**Container state with its fragment**
Where the inner states are written	In the braces of the state, in the same file.	In a fragment, a file of its own. Several container states may use one fragment, and each use runs a copy of it.
Names	One name space for the whole file. Transitions cross the border of the state freely, into and out of any inner state.	The file and the fragment each have their own name space. The using text cannot name a state of the fragment, and the fragment cannot name a state outside itself, so no transition crosses the border of the container state. After flattening, a state of the fragment is named with the name of the container state and a slash in front, see 2.3.12.3 Names.
The `INITIAL` mark	The single mark of the file may stand on an inner state, and the state machine then starts inside the compound state.	The fragment has a mark of its own, which the using text does not see. It decides where the fragment starts.
What a transition into the state enters	The state itself and none of its inner states.	At once the start of the fragment, or the remembered state when the container state has a history that remembers one: a container state is never active without a state of its fragment inside it.
Actions	`ENTRY` and `DO` blocks.	None. The states of the fragment carry them.

	Compound state	**Container state with its fragment**
Finishing	No final states and no `DONE`. A transition from an inner state leads out.	The fragment ends in a `FINAL STATE`, and `DONE` on a transition from the container state holds once it has reached one, see 2.3.12.2 Container states and DONE.
Leaving	A transition from the compound state leaves it from any of its inner states.	A transition from the container state leaves it from any state of the fragment and ends the fragment.
Histories	`SHALLOW HISTORY` and `DEEP HISTORY` remember the inner state, see 2.3.7 Histories.	The same, for the states of the fragment. At run time the two differ in when the remembered state is entered, see 3.1.14.6 Histories, tick limits and breakpoints.
Tick limits and breakpoints	Allowed, as on every state.	Allowed on the container state. The states of the fragment have their own, and each copy of the fragment has its own states.
Macros	The macros of the file.	The macros of all files of the composition form one set, see 2.3.12.4 Macros in a composition. A file should call only the macros it defines itself.

The lamp below switches between full and dimmed light while it is on, and a dim in `dimmed` switches it off. Written with a compound state:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            STATE "full" { }
            STATE "dimmed" { }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "full" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "full" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "dimmed" -> "off" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

Written with a container state, the light levels move into a fragment:

```
STATEMACHINE "fading" FRAGMENT {
    STATES {
        INITIAL STATE "full" { }
        STATE "dimmed" { }
        FINAL STATE "faded" { }
    }
    TRANSITIONS {
        TRANSITION "full" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "dimmed" -> "faded" {
            CONDITION { EVENT("switch", "dim") }
        }
    }
}
```

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        CONTAINER STATE "on" USE "fading" { }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { DONE }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

The compound state lets `off` lead straight into `full` and `dimmed` lead straight out to `off`. The container state cannot: the lamp enters `on`, which starts the fragment in `on/full`, and the fragment cannot name `off`, so it ends in its final state `faded`, and `DONE` leads out. This takes one more step at run time, see 3.1.14.5 Final states and DONE.

Which one to use: a compound state suits a group of states that belongs to one state machine and needs transitions across its border. A container state suits a piece of behaviour with a clear start and end that is long enough for a file of its own, or that several state machines or several places of one state machine run, such as a self test.

2.4 Checks

The library checks texts in three steps. Each problem it finds is a diagnostic: a range of the text, a severity, a code and a message. The code names the kind of problem and stays the same across versions, while the message may change. An error means the text is wrong. A warning means the text is accepted but very likely does not do what was meant.

2.4.1 What is checked when

The reader of a language parses a text and reports its syntax errors. The parser recovers from an error and reads on, so one text may show several syntax errors.

The reader then checks the text on its own: the KMeta reader the types, the KModel reader the objects against the metamodel, and the KStates reader the states, transitions and macros of the state machine. Where a syntax error may have cost the reader part of the text, it leaves out the checks that would report what the lost part might hold. A text with a syntax error therefore gets no report of an unknown state or macro in KStates, of an unknown target type or a missing type in KMeta, or of a `knows` to a missing object in KModel.

The verifier checks the state machines of a project against the domain metamodel, the event metamodel and the model: paths, operations, values, macro calls, events in scope, the states that can be reached, the names of the attachable state machines that `behaves after`, `ATTACH` and `EXCHANGE` use, and the compositions of fragments. It reports its problems in the text of the state machine they concern, and the problems of `behaves after` in the text of the metamodel. It leaves out a state machine with a syntax error, and each of its checks needs the texts it relies on without errors: the checks of types a domain metamodel without errors, the checks of event fields an event metamodel without errors, the checks of values a model without errors and the object the state machine runs on, and the checks of states and events a state machine whose state names are unique and known.

The IDE shows the problems of the readers and of the verifier, see 4.4 Validating. From code, the readers and the verifier are called as 2.5 Using the library shows.

2.4.2 Syntax

These codes are the same in every language.

Code	Severity	Meaning	Example
`syntax.unrecognized-input`	error	The text holds characters that start nothing the language knows. The reader skips them.	`type 'Bulb' "A bulb" { }` in KMeta, which has no single quotes
`syntax.error`	error	A word or sign stands where the language does not allow it, or a required one is missing.	`type "Bulb" { }`, which lacks the description
`reader.failure`	error	The reader failed and read nothing, for example on states or objects nested deeper than the parser can follow.	

2.4.3 KMeta

Code	Severity	Meaning	Example
`kmeta.no-types`	error	The text defines no type, so the metamodel has no root type.	a text that holds only comments
`kmeta.unknown-property-type`	error	A `prop` names a type other than `string`, `number` and `boolean`.	`prop("watts", "Number")`

Code	Severity	Meaning	Example
`kmeta.unknown-target-type`	error	A `has` or `knows` names a type that the metamodel does not define.	`has("bulb", "Bulb")` without a type `Bulb`
`kmeta.duplicate-type`	error	A type has the name of an earlier type. Reported at the later name.	two types `"Bulb"`
`kmeta.duplicate-key`	error	A type uses a key twice, for properties, relations or both. Reported at the later key.	`prop("watts", "number")` and `prop("watts", "string")` in one type

2.4.4 KModel

The examples are read against the lamp metamodel of 2 Modeling languages.

Code	Severity	Meaning	Example
`kmodel.unknown-class-type`	error	An object is of a type that the metamodel does not define.	`make object "lamp": "Lantern" { }`
`kmodel.unknown-property`	error	A `prop` names a property that the type of its object does not define.	`prop("color") -> "red"` in a bulb
`kmodel.unknown-relation`	error	A `has` or `knows` names a relation that the type of its object does not define.	`knows("room") -> "living room"` in a lamp
`kmodel.wrong-relation-kind`	error	A `has` names a `knows` relation, or a `knows` names a `has` relation.	`knows("bulb") -> "bulb"` in a lamp
`kmodel.invalid-number`	error	A number property gets a value that is not a number. The value is left out.	`prop("watts") -> "forty"`
`kmodel.invalid-boolean`	error	A boolean property gets a value other than `true` or `false` in any case. The value reads as false.	`prop("lit") -> "yes"`
`kmodel.several-values`	error	A property or relation that is not a list gets several values. Reported at each further statement.	`prop("watts") -> "20"` and `prop("watts") -> "40"` in one bulb
`kmodel.duplicate-id`	error	An object has the key of an earlier object, embedded objects included. Reported at the later key.	two objects `"lamp"`
`kmodel.unresolved-reference`	error	A `knows` names a key that no object has.	`knows("control") -> "kitchenSwitch"`
`kmodel.reference-type-mismatch`	error	A `knows` names an object of another type than the relation declares.	`knows("control") -> "bulb"` in the lamp

Code	Severity	Meaning	Example
`kmodel.embedded-type-mismatch`	warning	A `has` embeds an object of another type than the relation declares.	`has("bulb") -> make object "s":"Switch" { }`

2.4.5 KStates

The examples are parts of a state machine attached to `Lamp`.

Code	Severity	Meaning	Example
`kstates.initial-state-count`	error	No state or more than one state is marked `INITIAL`.	`STATES { STATE "off" { } STATE "on" { } }`
`kstates.action-arity`	error	An action rule has another number of operands than its operation takes.	`SET("mode")`
`kstates.tick-count`	error	A `DELAY` or `TIMEOUT` counts its ticks with a number that is not whole. The part is left out.	`DELAY: 2.5 TICKS`
`kstates.feature-off`	error	A construct of an optional feature that is switched off, see 2.3.10 Optional features.	`SHALLOW HISTORY STATE "on"` while histories are off
`kstates.duplicate-state`	error	A state has the name of an earlier state, at any depth. Reported at the later name.	`STATE "on" { STATE "on" { } }`
`kstates.unknown-state`	error	A transition, split, join or timeout names a state that does not exist.	`TRANSITION "off" -> "dimmed" { }` without a state `dimmed`
`kstates.repeated-split-target`	error	A split names the same target twice.	`SPLIT "off" -> "on", "on" { }`
`kstates.repeated-join-source`	error	A join names the same source twice.	`JOIN "on", "on" -> "off" { }`
`kstates.duplicate-macro`	error	A macro has the name of an earlier macro. The runtime only calls the earlier one.	two macros `"half"`
`kstates.duplicate-parameter`	error	A macro takes two parameters of the same name.	`TAKES("w", "number")` twice in one macro
`kstates.unknown-macro`	error	A `MACRO(...)` call, or an `@name(...)` call in `EVAL` code, names a macro that does not exist.	`MACRO("half", "bulb->maxWatts")` without a macro `half`
`kstates.macro-argument-count`	error	A `MACRO(...)` call passes more arguments than the macro takes, or an `@name(...)` call passes another number.	`@half(1, 2)` for a macro with one `TAKES`

Code	Severity	Meaning	Example
`kstates.missing-macro-arguments`	warning	A `MACRO(...)` call passes fewer arguments than the macro takes. The runtime reads each missing one from the path named like its parameter.	`MACRO("half")` for a macro with one `TAKES`
`kstates.several-event-clauses`	error	A condition has more than one `EVENT` clause. The runtime only matches the first.	`CONDITION { EVENT("switch", "press") EVENT("switch", "dim") }`
`kstates.event-clause-not-first`	warning	An `EVENT` clause is not the first clause of its condition. The runtime finds it in any place, but it reads better first.	`CONDITION { VALUE("bulb->lit") EVENT("switch", "press") }`
`kstates.history-without-inner-states`	warning	A state without inner states has a history.	`SHALLOW HISTORY STATE "off" { }`
`kstates.empty-breakpoint`	warning	A `BREAKPOINT` block marks nothing.	`BREAKPOINT { }`
`kstates.empty-tick-limit`	warning	A `TICK LIMIT` block has neither a `DELAY` nor a `TIMEOUT`.	`TICK LIMIT { }`
`kstates.delay-beyond-timeout`	warning	A `DELAY` is longer than the `TIMEOUT` of its block, so the timeout fires only after the delay.	`TICK LIMIT { DELAY: 5 TICKS TIMEOUT: AFTER 2 TICKS TRANSITION TO "off" }`
`kstates.comment-in-eval`	warning	`EVAL` code holds `//` outside a string literal, which starts a KStates comment, so a closing brace after it on its line does not count.	`return $(bulb->watts) / / 2` in an `EVAL`
`kstates.eval-without-return`	warning	`EVAL` code has no `return`, so it yields no value.	`EVAL { $(bulb->maxWatts) }`
`kstates.missing-state-machine-name`	error	An attachable state machine or a fragment has no name. Reported at `ATTACHABLE` or `FRAGMENT`, or at the empty string for an empty name.	`STATEMACHINE ATTACHABLE TO "Bulb"`
`kstates.attached-header`	error	A state machine `ATTACHED TO` a type has a name or a second type. Reported at each.	`STATEMACHINE "lamp" ATTACHED TO "Lamp"`
`kstates.repeated-attachable-type`	error	An attachable state machine lists a type twice. Reported at the later one.	`STATEMACHINE "glowing" ATTACHABLE TO "Bulb", "Bulb"`

Code	Severity	Meaning	Example
`kstates.container-use`	error	A `CONTAINER STATE` lacks `USE "fragment"`, reported at `CONTAINER`, or uses an empty name, reported at the empty string, or another state has a `USE`, reported at `USE`.	`CONTAINER STATE "self test" { }`
`kstates.container-state-content`	error	A container state has an `ENTRY` or `DO` block or inner states. Reported at each.	`CONTAINER STATE "self test" USE "bulb test" { ENTRY { SET("mode", "test") } }`
`kstates.final-state-content`	error	A final state is `INITIAL`, has a history, or holds more than a `BREAKPOINT` block. Reported at each.	`FINAL STATE "passed" { TICK LIMIT { DELAY: 2 TICKS } }`
`kstates.misplaced-final-state`	error	A final state stands in a state machine that is no fragment, or inside another state of a fragment. Reported at its name.	`FINAL STATE "done" { }` in the lamp
`kstates.fragment-without-final-state`	error	A fragment has no final state at its top level. Reported at `STATES`.	a fragment with only `INITIAL STATE "powered" { }`
`kstates.transition-from-final-state`	error	A transition, split or join leaves a final state. Reported at its name.	`TRANSITION "passed" -> "powered" { }`
`kstates.misplaced-done`	error	`DONE` stands on a `TRANSITION` or `SPLIT` whose source is no container state, or on a `JOIN` of which no source is a container state. Reported at `DONE`.	`TRANSITION "off" -> "on" { CONDITION { DONE } }`
`kstates.fragment-uses-itself`	error	A container state of a fragment uses the fragment itself. Reported at the name after `USE`.	`CONTAINER STATE "again" USE "bulb test" { }` in `bulb test`

A header without a type after `TO` is a syntax error, and so is `ATTACH` after any rule other than `SETOBJ` and `APPENDOBJ`.

The checks of names and calls are left out for a text with a syntax error: `kstates.unknown-state`, `kstates.unknown-macro`, `kstates.macro-argument-count` and `kstates.missing-macro-arguments`. So is the count of initial states when the `STATES` block has a syntax error.

A text that is a fragment or has a container state may call a macro that another file of its composition defines, so the reader reports no `kstates.unknown-macro` there and does not count the arguments of such a call; the verifier checks these calls against the whole composition. A container state with a history gets no `kstates.history-without-inner-states`, since its inner states come from its fragment. `kstates.misplaced-done` is left out while a source of the transition names no state, and `kstates.fragment-without-final-state` while the `STATES` block has a syntax error.

2.4.6 Project

The verifier reports these problems. The examples are parts of a state machine attached to `Lamp` in a project with the lamp metamodel and the lamp model of 2 Modeling languages, and with a macro `half` that takes a number.

Code	Severity	Meaning	Example
`project.unknown-attached-type`	error	The domain metamodel defines no type of the name after `ATTACHED TO`, or of a name after `ATTACHABLE TO`, which is reported at that name.	`STATEMACHINE ATTACHED TO "Lantern"`
`project.attached-object-type`	error	The model has no object of the key the state machine runs on, or that object is of another type than the state machine is attached to.	the lamp state machine run on `"wallSwitch"`
`project.unknown-path`	error	The path of an action rule or of a `VALUE` condition leads nowhere.	`SET("bulb->color", "red")`
`project.operation-mismatch`	error	An operation does not fit what its path leads to, see 2.4.6.2 Operations.	`APPEND("mode", "on")`
`project.literal-mismatch`	error	The string value of a `SET`, `APPEND` or `SETLIST` cannot be read as the type of its property.	`SET("bulb->watts", "bright")`
`project.unresolved-code-reference`	error	A `$(path)` in `EVAL` code leads nowhere.	`EVAL { return $(bulb->power) }`
`project.unknown-macro-type`	error	A `TAKES` or `RETURNS` names no primitive type, no domain type and no event type, or uses a function other than `list` and `reference`.	`TAKES("w", "Watt")`
`project.unresolved-macro-argument`	error	An argument of a `MACRO(...)` call leads nowhere.	`MACRO("half", "bulb->power")`
`project.macro-argument-type`	error	An argument of a `MACRO(...)` call leads to a value of another type than its parameter takes.	`MACRO("half", "mode")`
`project.unknown-event-field`	error	`event->...` reads a field that no event that can be in scope there carries, or the event metamodel does not declare the event.	`VALUE("event->level")` after `EVENT("switch", "dim")`
`project.unreachable-state`	warning	A state can never be active. Only the outermost of such states is reported.	a state that no transition leads into

Code	Severity	Meaning	Example
`project.shadowed-transition`	warning	An earlier transition from the same state has no condition and no `NOT LOOPING`, so it always fires first and this one never fires.	`TRANSITION "off" -> "on" { }` before another transition from `off`
`project.join-never-fires`	warning	No split leads to branches in which each source of the join is active in a branch of its own.	`JOIN "full", "dimmed" -> "off" { }` without a split into them
`project.split-inside-parallel`	warning	The source of a split is active only inside a branch of a split, where no split fires.	a split from a target of another split
`project.event-not-in-scope`	warning	Code or a condition reads `event` where no event can be in scope.	`$(event->bright)` in the initial state
`project.unset-property`	warning	The model gives the object the state machine runs on no value on the way of a read, and no action of the project writes any key of the path.	`$(bulb->lit)` when the model gives the bulb no `lit` and no action sets it
`project.duplicate-state-machine-name`	error	Two named state machines of the project, attachable ones or fragments, have the same name. Reported at the name in each.	two files with `STATEMACHINE "glowing" ATTACHABLE TO ...`
`project.unknown-state-machine`	error	A `behaves after`, an `ATTACH` or an `EXCHANGE` names, as a string, no attachable state machine of the project.	`EXCHANGE("bulb", "flashing")` without a state machine `flashing`
`project.attachable-type-mismatch`	error	A `behaves after`, an `ATTACH` or an `EXCHANGE` names a state machine that does not list the type of the object in question.	`EXCHANGE("bulb", "keeper")` when `keeper` is attachable to `Room` only
`project.attach-on-knows`	error	`ATTACH` follows a `SETOBJ` or `APPENDOBJ` on a `knows` relation, which links an existing object instead of creating one. Reported at the clause.	`SETOBJ("self", "control", "switch") ATTACH "pressing"`
`project.behaves-after-in-events`	error	A type of the event metamodel has `behaves after`. Reported at the clause, in the event metamodel.	`type "press" "A press" behaves after "glowing" { }`
`project.attached-fragment`	error	A `behaves after`, an `ATTACH` or an `EXCHANGE` names a fragment, which only a container state runs. Reported at the name.	`EXCHANGE("bulb", "bulb test")`

Code	Severity	Meaning	Example
`project.unknown-fragment`	error	A container state uses a name that no fragment of the project has. Reported at the name after `USE`.	`CONTAINER STATE "self test" USE "bulb tset" { }`
`project.not-a-fragment`	error	A container state uses a state machine that is no fragment. Reported at the name after `USE`.	`CONTAINER STATE "self test" USE "glowing" { }`
`project.fragment-cycle`	error	A fragment contains itself through other fragments. Reported once, at the name after `USE` of the container state that closes the cycle.	`bulb test` uses `wiring test`, which uses `bulb test`
`project.qualified-name-clash`	error	A flattened name equals another state name of the composition. Reported at the later state.	a state `"self test/ powered"` next to the container state `self test`
`project. conflicting-macros`	error	Two files of a composition define macros of the same name with different definitions. Reported at the later macro.	`half` in the lamp and another `half` in the bulb test
`project.unknown-macro`	error	A call in a fragment, or in a file with a container state, names a macro that no file of the composition defines. Reported at the call, and only if every state machine of the project could be read.	`MACRO("third", "bulb->maxWatts")` in the bulb test
`project.macro-from-other-file`	warning	A call names a macro that only another file of the composition defines. Reported at the call.	`MACRO("half", "bulb->maxWatts")` in a fragment without a macro `half`
`kstates.macro-argument-count`	error	A call of a macro that only another file of the composition defines passes a number of arguments the macro does not take, by the rule of the reader. The composition does not flatten. Reported at the call.	`MACRO("half", "bulb->maxWatts", "bulb->power")` in a fragment without a macro `half`

2.4.6.1 Paths

The verifier follows a path key by key, the way the runtime follows it. Each key names a relation or a property of the type reached so far. A path goes on from the first object of a list relation, but not from a property. In the code of a macro, a path cannot go on from a list. A path starts at a binding of a `WITH` block if it starts with its name, at the event in scope if it starts with `event`, in the code of a macro at a parameter if it starts with its name, and otherwise at the object the state machine runs on.

The paths of `IF IN SCOPE` blocks are never reported. A path inside such a block that starts with one of its paths, or with `event` if one of its paths does, is not reported as leading nowhere.

2.4.6.2 Operations

`SET` needs a property that is not a list. `APPEND`, `SETLIST` and `DROPLIST` need a list property. On a relation, the message names the relation operation to use instead. `SETOBJ` needs a relation that holds one object and `APPENDOBJ` a list relation. A string target of either must be a path to one object of the type of the relation, and only a `knows` relation may take it, since a `has` relation takes a new object. `DROPOBJ` and `EXCHANGE` need a path to one object. A `VALUE` condition needs a boolean or a string property that is not a list.

2.4.6.3 Events and states

An event comes into scope at the target of the transition or split whose `EVENT` clause matched it, and stays while that state is active. The code of a state sees the events of the transitions into the state, into a state around it and into a state inside it. The conditions of a transition see the event of their own `EVENT` clause, or else the events in scope in the state the transition starts in. The conditions of a join see none.

A state is reachable if a transition, split, join or timeout leads into it, or into a state inside it, from the initial state. Every condition counts as possibly true. A transition into a state enters none of its inner states.

2.4.6.4 Attachment

An attachable state machine runs on objects of each type it lists, so the verifier checks the paths and operations of such a state machine once for each type. A problem found for every type is reported once, as it is. A problem found for some types only names the type in front of its message, as in `For type 'Dimmer': Type 'Dimmer' has no property or relation 'lit'`.

The names that `behaves after`, `ATTACH` and `EXCHANGE` use are checked only where they are written as strings; a name that an `EVAL` or a `MACRO` computes is checked by the runtime when the action runs. The type an `ATTACH` is checked against is the type of its relation, and the type an `EXCHANGE` is checked against is the type of the object its path leads to; `EXCHANGE` needs a path to one object, like `DROPOBJ`. A name that no attachable state machine has is not reported while a state machine of the project has a syntax error, since that state machine might carry the name. The `behaves after` clauses are checked in a domain metamodel without syntax errors.

2.4.6.5 Pages

The verifier flattens each state machine of the project that is no fragment with the fragments of the project, the same way the runtime does, see 2.3.12 Pages. A problem that prevents the flattening is reported in the text where it stands, a fragment included. A problem the reader of that text reports already, such as a fragment that uses itself, is not repeated. `project.unknown-fragment` is not reported while a state machine of the project has a syntax error, since that state machine might be the fragment.

The other checks of the project then run on the flattened state machine, so the content of a fragment is checked where it runs, with the type of the state machine that uses it. A problem inside a fragment is reported in the text of the fragment, once for each place however often the fragment is used. A state of a fragment is reachable when the container state that uses it is, through the start path of the fragment and its transitions.

A composition that does not flatten is checked as the single state machine it starts from, as written. A fragment that no state machine uses is checked on its own, as far as it can be without the type of a host: its states and transitions, the fragments it uses and its macro calls. A fragment that a state machine uses, but whose compositions do not flatten, is checked on its own as well, without its macro calls, which may rely on the macros of the state machine that uses it.

2.5 Using the library

karpfen-dsl-tools is the Kotlin library that reads, checks and writes the three languages. The runtime and the IDE use it, and any JVM program can use it too. The examples use the lamp files of 2 Modeling languages: `lamp.kmeta`, `lamp.kmodel`, `lamp.kstates`, and `lamp-events.kmeta` from 2.1.7 Event definitions.

2.5.1 Adding the library to a build

Item	Value
Group and artifact	`io.karpfen:karpfen-dsl-tools`
Version	`2.0.0-SNAPSHOT`
Repository	`https://maven.pkg.github.com/karpfenproject/karpfen-dsl-tools` (GitHub Packages)
JDK	21
Dependencies a consumer gets	ANTLR runtime 4.13.1, `org.json`, SLF4J API

GitHub Packages asks for credentials also for reading: a GitHub user and a token that may read packages. The library itself names them as the Gradle properties `GitHubPackagesUsername` and `GitHubPackagesPassword` when it publishes.

A build may also take the library from a git submodule. The runtime includes the checkout as a subproject with `include 'karpfen-dsl-tools'` in its `settings.gradle`, and the IDE includes it as an included build. To build the library on its own, run `./gradlew build` in its folder with JDK 21.

2.5.2 Reading a text

A reader reads one text and never throws. `KmetaReader`, `KmodelReader` and `KstatesReader` each have the method `read(text)`, which returns a `ParseResult`. It holds the model the reader built, the problems it found and a source map, which tells where each element of the model is written. A byte order mark at the start of the text is ignored, while positions still count it as the first char of the first line, as the text holds it.

Member of `ParseResult`	Content
`model`	The metamodel, model or state machine. With errors it is partial: the reader leaves out what it cannot read. It is null only if nothing could be built, such as for a KMeta text without a type.
`diagnostics`	Every problem, ordered by its position in the text.
`errors`	The problems of severity `ERROR`.
`hasErrors`	True if there is at least one error.
`sourceMap`	The ranges of the elements of `model` in the text.

`KmodelReader` takes the metamodel in its constructor, and `KstatesReader` the optional features that are switched on, see 2.5.6 Optional features. A reader keeps no state between reads, so one reader may serve several threads.

```
import io.karpfen.dsl.diagnostics.Diagnostic
import io.karpfen.dsl.reader.kmeta.KmetaReader
import io.karpfen.dsl.reader.kmodel.KmodelReader
import java.io.File

fun report(fileName: String, problems: List<Diagnostic>) {
    for (problem in problems) {
        val start = problem.range.start
        println("$fileName:${start.line}:${start.column}: ${problem.severity} ${problem.
    ↪ code.id}: ${problem.message}")
    }
}

fun readLamp(folder: File) {
    val metamodelRead = KmetaReader().read(File(folder, "lamp.kmeta").readText())
    report("lamp.kmeta", metamodelRead.diagnostics)
    val metamodel = metamodelRead.model ?: return
    val modelRead = KmodelReader(metamodel).read(File(folder, "lamp.kmodel").readText())
    report("lamp.kmodel", modelRead.diagnostics)
    val room = modelRead.model?.objects?.singleOrNull() ?: return
    val bulb = room.getRel("lamp").single().getRel("bulb").single()
    println(bulb.getProp("maxWatts"))
    println(modelRead.sourceMap.nameRangeOf(bulb))
}
```

For the lamp files, `readLamp` reports no problem and prints the maximum watts of the bulb and where its key is written:

```
[40.0]
SourceRange(start=SourcePosition(offset=148, line=4, column=37), end=SourcePosition(offset
    ↪ =152, line=4, column=41))
```

A diagnostic has a `range` with a `start` and an `end`, each with an `offset`, a `line` and a `column`; lines and columns count from 1. Its `code` is a `DiagnosticCode`, whose `id` is the code that 2.4 Checks lists, and its `severity` is `ERROR` or `WARNING`. The code is stable, the `message` is meant for people. A metamodel with `prop("watts", "Number")` in line 2 gives:

```
lamp.kmeta:2:20: ERROR kmeta.unknown-property-type: Unknown type 'Number' of prop 'watts';
    ↪ a prop is of type 'string', 'number', 'boolean'
```

A metamodel holds its types in `types`, each a `ClassType` with `simpleProperties` and `objectProperties`, and its root type in `rootClass`. A model holds its root object in `objects`. A `DataObject` has its `id`, its type `ofType`, the values of a property by `getProp(key)` and the objects of a relation by `getRel(key)`, both as lists. A `ClassType` whose KMeta type has `behaves after` holds it in `behavesAfter`, a `BehavesAfter` with the `stateMachineName`. A state machine read from KStates is a `StateMachine` with `attachedToClass`, `states`, `transitions` and `macros`:

```
import io.karpfen.dsl.reader.kstates.KstatesReader
import java.io.File

fun readLampMachine(folder: File) {
    val read = KstatesReader().read(File(folder, "lamp.kstates").readText())
    report("lamp.kstates", read.diagnostics)
    val machine = read.model ?: return
    println(machine.attachedToClass)
    for (state in machine.states) {
        println("${state.name} ${read.sourceMap.rangeOf(state)?.start?.line}")
    }
}
```

```
Lamp
off 3
on 10
```

An attachable state machine, see 2.3.1 The state machine, has `isAttachable` true, its `name` and its types in `attachableTo`, and an empty `attachedToClass`. An attached one has the `name` null. `types` holds the types of either form. A fragment, see 2.3.12 Pages, has `isFragment` true, its `name`, no types and `isAttachable` false. A `State` tells its kind: `isContainer` with the name of its fragment in `fragment`, and `isFinal`. `DONE` is a `DoneCondition`. An action rule with an `ATTACH` clause holds it in `attach`, an `AttachClause` whose `stateMachineName` is the value, and `EXCHANGE` is the operation `ActionOperationType.EXCHANGE`.

The source map has `rangeOf(element)` for the whole element, `nameRangeOf(element)` for its name or key, `codeRangeOf(element)` for the code of an `EVAL` and `operandRangesOf(element)` for the operands of an action rule. It knows the elements of the model by identity. The name of an attached state machine is its type. The name of an attachable state machine is its name, and its operands are its types. An `AttachClause` and a `BehavesAfter` range over the whole clause, keyword included, and the name of a `BehavesAfter` is the name of the state machine.

2.5.3 Throwing converters

The converters are the throwing form of the readers, for a text that is expected to be right. They are `KmetaDSLConverter`, `KmodelDSLConverter` and `KstatesDSLConverter` in the package `io.karpfen.dsl.dsl.textual`, each with a function for a string and one for a file. Each reads through its reader, so every rule of the language holds for it as for the reader. It returns the model of a text without errors and throws for a text with at least one error. Warnings do not make it throw. `KstatesDSLConverter` reads with every optional feature switched on.

```
import io.karpfen.dsl.dsl.textual.JsonToModelConverter
import io.karpfen.dsl.dsl.textual.KmetaDSLConverter
import io.karpfen.dsl.dsl.textual.KmodelDSLConverter
import io.karpfen.dsl.dsl.textual.KstatesDSLConverter
import java.io.File

fun convertLamp(folder: File) {
    val metamodel = KmetaDSLConverter.parseKmetaFile(File(folder, "lamp.kmeta").path)
    val events = KmetaDSLConverter.parseKmetaString(File(folder, "lamp-events.kmeta").
    ↪ readText(), metamodel.types)
    val model = KmodelDSLConverter.parseKmodelFile(File(folder, "lamp.kmodel").path,
    ↪ metamodel)
    val machine = KstatesDSLConverter.parseKstatesFile(File(folder, "lamp.kstates").path)
    println("${metamodel.rootClass.name} ${model.objects.single().id} ${machine.states.size
    ↪ }")
    val payload = JsonToModelConverter(events).toDataObject("""{"bright": true}""", "dim")
    println(payload.getProp("bright"))
}
```

```
Room living room 2
[true]
```

`parseKmetaString` takes the types of the domain metamodel as its second argument when it reads an event metamodel. `KmodelDSLConverter.parseSingleObject` reads a KModel text such as an event payload and returns its root object. A KModel text is always one object, so it reads the text exactly as `parseKmodelString` does, with the same errors. `JsonToModelConverter` turns a JSON object into an object of a type, here the payload of the event `dim`.

A converter throws an `InvalidTextException` from the package `io.karpfen.dsl.reader`, the same type for all three languages. It is an `IllegalArgumentException`. Its `errors` hold every error the reader found, as the diagnostics of 2.5.2 Reading a text, and its `source` holds the path of the file, or null for a string. The message names each error on a line of its own, with the position, the code and the message. If the state `off` of `lamp.kstates` lacks its `INITIAL`, `parseKstatesFile("lamp.kstates")` throws with the message:

```
lamp.kstates has 1 error:
lamp.kstates:2:5: kstates.initial-state-count: Exactly one state must be marked as INITIAL,
    ↪  but found 0.
```

For a string, the first line reads `The text has 1 error:` and the lines start with the position. The converters take this step with `modelOrThrow()` of `ParseResult`, which a program can also call on the result of a reader: it returns the model of a text without errors and throws the same exception otherwise.

Use a converter where a wrong text is a failure, such as a runtime that loads a deployed project or a test. Use a reader where a wrong text is normal and its problems are shown, such as an editor, or where a partial model is needed.

`Model.collectAllObjects(objects)` in the package `io.karpfen.dsl.instance` returns the given objects and every object embedded in them, each once, each object before the objects it embeds. For a model read from text, `Model.collectAllObjects(model.objects)` returns all objects of the model.

2.5.4 Printing back to text

The printers write a metamodel, a model or a state machine as text: `KmetaPrinter`, `KmodelPrinter` and `KstatesPrinter` in the packages below `io.karpfen.dsl.printer`. `print` writes a whole text. The printers also write a single element as a fragment that can replace that element in an existing text: `printType` and `printRule` for KMeta, `printObject` for KModel, `printState`, `printTransition` and `printMacro` for KStates. The `convertTo... String` functions of the converters call the same printers.

```
import io.karpfen.dsl.instance.Model
import io.karpfen.dsl.printer.kmodel.KmodelPrinter
import io.karpfen.dsl.printer.kstates.KstatesPrinter
import io.karpfen.dsl.states.StateMachine

fun printLamp(model: Model, machine: StateMachine) {
    print(KmodelPrinter().print(model))
    val bulb = model.objects.single().getRel("lamp").single().getRel("bulb").single()
    println(KmodelPrinter().printObject(bulb))
    println(KstatesPrinter().printState(machine.states.first()))
}
```

For the lamp, the bulb and the state `off` come out as:

```
make object "bulb":"Bulb" {
    prop("maxWatts") -> "40.0"
    prop("watts") -> "0.0"
    prop("lit") -> "false"
}
INITIAL STATE "off" {
    ENTRY {
        SET("mode", "off")
        SET("bulb->lit", "false")
        SET("bulb->watts", "0")
    }
}
```

The printers use one layout: four spaces per level, one statement per line, and an empty block as `{ }`. A text that is read and printed again loses its comments, except comments inside `EVAL` code. A number of a model is written as the library holds it, so `"40"` comes back as `"40.0"`. The KMeta printer writes the properties of a type before its relations and the root type last. `KmetaPrinter().print(events, metamodel.types)` writes an event metamodel without the domain types it was read with. A printer throws an `IllegalArgumentException` for an element that its language cannot write, such as a name with a line break.

2.5.5 Building models in code

`MetamodelBuilder` and `ModelBuilder` in the package `io.karpfen.dsl.dsl.functional` build a metamodel and a model without text. `makeClass` defines a type, `makeNumber`, `makeString` and `makeBool` and their `...List` forms make properties, and `makeClassProp` and `makeClassPropList` make relations, `EMBEDDED` for `has` and `LINK` for `knows`. A relation can only lead to a type that is already defined.

```
import io.karpfen.dsl.dsl.functional.MetamodelBuilder
import io.karpfen.dsl.meta.Metamodel

fun buildLampMetamodel(): Metamodel {
    val m = MetamodelBuilder()
    m.makeClass("Bulb", "A light bulb", props = listOf(m.makeNumber("maxWatts"), m.
    ↪ makeNumber("watts"), m.makeBool("lit")))
    m.makeClass("Switch", "A wall switch", props = listOf(m.makeString("position"), m.
    ↪ makeStringList("presses")))
    m.makeClass(
        "Lamp", "A lamp with a bulb, wired to a switch",
        props = listOf(m.makeString("mode")),
        rels = listOf(m.makeClassProp("bulb", "Bulb", m.EMBEDDED), m.makeClassProp("control
    ↪ ", "Switch", m.LINK))
    )
    m.makeClass(
        "Room", "The room the lamp stands in",
        rels = listOf(m.makeClassProp("lamp", "Lamp", m.EMBEDDED), m.makeClassProp("switch",
    ↪  "Switch", m.EMBEDDED))
    )
    m.setRoot("Room")
    return m.build()
}
```

`ModelBuilder.makeObject(type, id)` makes an object. It gives every property a default value, `false`, the empty string, `Double.MIN_VALUE` or an empty list, and every relation to one object a new default object without id. `assignProps` and `assignRels` replace these values; a key that the type does not have is ignored.

```
import io.karpfen.dsl.dsl.functional.ModelBuilder
import io.karpfen.dsl.instance.Model
import io.karpfen.dsl.meta.Metamodel

fun buildLampModel(metamodel: Metamodel): Model {
    val b = ModelBuilder(metamodel)
    val wallSwitch = b.makeObject("Switch", "wallSwitch")
    wallSwitch.assignProps(mapOf("position" to "down"))
    val bulb = b.makeObject("Bulb", "bulb")
    bulb.assignProps(mapOf("maxWatts" to 40.0, "watts" to 0.0, "lit" to false))
    val lamp = b.makeObject("Lamp", "lamp")
    lamp.assignProps(mapOf("mode" to "off"))
    lamp.assignRels(mapOf("bulb" to bulb, "control" to wallSwitch))
    val room = b.makeObject("Room", "living room")
    room.assignRels(mapOf("lamp" to lamp, "switch" to wallSwitch))
    return b.build()
}
```

The model of a builder holds every object it made, the default objects included.

`KmodelPrinter().print` starts at the object of the root type with an id that no other object embeds, and for this model writes the same text as the lamp model of 2 Modeling languages, with numbers such as `"40.0"`.

2.5.6 Optional features

`KstatesReader` takes the optional features that a text may use, see 2.3.10 Optional features. The default is `KstatesFeature.ALL`. Each feature has a stable `id`, and `KstatesFeature.byId` finds a feature by its id, for example from a settings file.

```
import io.karpfen.dsl.reader.ParseResult
import io.karpfen.dsl.reader.kstates.KstatesReader
import io.karpfen.dsl.states.KstatesFeature
import io.karpfen.dsl.states.StateMachine

fun readWithFeatures(text: String, featureIds: List<String>): ParseResult<StateMachine> {
    val switchedOn = featureIds.mapNotNull { KstatesFeature.byId(it) }.toSet()
    return KstatesReader(switchedOn).read(text)
}
```

The converter `KstatesDSLConverter` always accepts every feature.

2.5.7 Verifying a project

`ProjectVerifier` in the package `io.karpfen.dsl.verification` checks the state machines of a project against the metamodels and the model, see 2.4.6 Project. It takes the results of the readers in a `ProjectInput`: the domain metamodel, the event metamodel or null, the model or null, and the state machines. Each state machine is a `StateMachineInput` with the key of the object it runs on, or null if it is not known yet. The list holds every state machine of the project, the attached and the attachable ones and the fragments alike; an attachable state machine and a fragment have no key. The verifier needs all of them to check the names that `behaves after`, `ATTACH` and `EXCHANGE` use. The event metamodel is read with the domain types as base types.

```
import io.karpfen.dsl.reader.kmeta.KmetaReader
import io.karpfen.dsl.reader.kmodel.KmodelReader
import io.karpfen.dsl.reader.kstates.KstatesReader
import io.karpfen.dsl.verification.ProjectInput
import io.karpfen.dsl.verification.ProjectVerifier
import io.karpfen.dsl.verification.StateMachineInput
import java.io.File

fun verifyLamp(folder: File) {
    val metamodel = KmetaReader().read(File(folder, "lamp.kmeta").readText())
    val domainTypes = metamodel.model?.types.orEmpty()
    val events = KmetaReader(domainTypes).read(File(folder, "lamp-events.kmeta").readText()
    ↪ )
    val model = metamodel.model?.let { KmodelReader(it).read(File(folder, "lamp.kmodel").
    ↪ readText()) }
    val lamp = StateMachineInput(KstatesReader().read(File(folder, "lamp.kstates").readText
    ↪ ()), "lamp")
    val findings = ProjectVerifier().verify(ProjectInput(metamodel, events, model, listOf(
    ↪ lamp)))
    report("lamp.kstates", findings.of(lamp))
}
```

`findings.of(lamp)` holds the problems in the text of that state machine, ordered by position. `findings.metamodel` and `findings.events` hold the problems in the text of the domain metamodel and of the event metamodel, which concern `behaves after`. `findings.all`, `findings.errors` and `findings.warnings` hold those of all texts. The verifier adds to the diagnostics of the readers and never repeats them, so a tool shows both.

2.5.8 Flattening a composition

`StateMachineFlattener.flatten(machine, available)` in the package `io.karpfen.dsl.composition` flattens the composition that starts from `machine`, see 2.3.12 Pages. `available` holds the state machines it may use, usually every state machine of the project; it uses only the fragments among them, found by name. The function is pure: it changes neither its input nor anything else, and it throws for no problem of the texts.

It returns a `FlatteningResult`. A `Flattened` holds the flat state machine in `machine`, with the header of `machine` and no container state left to resolve: each container state holds copies of the states of its fragment, named with slashes. `sources` lists the state machines of the composition, the starting one first. `originOf(name)` and `origins` tell where each state was written, as a `StateOrigin` with the `fragment` it was written in, or null for the starting state machine, its `localName` in that text and the flattened names of the `containers` above it. A `NotFlattened` holds the `problems` that prevent the flattening, each a `FlatteningProblem` with the state machine it stands in, the element it concerns and a `message`: `UnknownFragment`, `NotAFragment`, `FragmentCycle`, `NameClash`, `ConflictingMacros`, `UnknownMacro` and `MacroArgumentMismatch`, a call whose number of arguments does not fit the macro it names in another file of the composition.

```
import io.karpfen.dsl.composition.FlatteningResult
import io.karpfen.dsl.composition.StateMachineFlattener
import io.karpfen.dsl.reader.kstates.KstatesReader
import java.io.File

fun flattenLamp(folder: File) {
    val lamp = KstatesReader().read(File(folder, "lamp.kstates").readText()).model ?:
    ↪ return
    val bulbTest = KstatesReader().read(File(folder, "bulb test.frag.kstates").readText()).
    ↪ model ?: return
    when (val result = StateMachineFlattener.flatten(lamp, listOf(bulbTest))) {
        is FlatteningResult.Flattened -> {
            for ((name, origin) in result.origins) {
                println("$name ${origin.fragment} ${origin.localName}")
            }
        }
        is FlatteningResult.NotFlattened -> result.problems.forEach { println(it.message) }
    }
}
```

For the lamp with the container state `self test` and the bulb test of 2.3.12.2 Container states and DONE, this prints:

```
off null off
self test null self test
self test/powered bulb test powered
self test/passed bulb test passed
self test/failed bulb test failed
```

```
on null on
```

In the flat state machine, `isContainer` and `isFinal` stay as written. `InitialStates.startPath(machine)` in the package `io.karpfen.dsl.states` returns the states a state machine starts in, and `InitialStates.enteredBelow(state)` the states entered below a container state when it is entered, both the outermost first, by the rule of 2.3.2.2 The initial state. `initialInnerState` of a state is the first state of that path below it. The verifier flattens each state machine of a project in the same way, see 2.4.6.5 Pages, and the runtime flattens a state machine before it runs.

2.5.9 Logging

The library logs through the SLF4J API and ships no logging backend. The program that uses it picks one, or gets no log output.

Logger	Level	What
Below `io.karpfen.dsl.reader`	error	A reader that failed through a defect, with its stack trace.
Below `io.karpfen.dsl.reader`	warn	A reader that failed on nesting deeper than its parser can follow.

A reader reports the problems of a text only as diagnostics, never in the log, and a converter only in its exception. When a reader fails, it reports the diagnostic `reader.failure` as well, and a converter throws with it.

3 Execution

The karpfen runtime is a server program that runs models. It reads a metamodel, a model, event definitions and state machines in the languages of 2 Modeling languages, runs state machines on the objects of the model and runs all of them in steps called ticks. A state machine is attached to one object, or it is attachable and objects get instances of it at the start and while the model runs. A state machine reads and changes its object and the objects it reaches from there. Events come from clients over a WebSocket and from the state machines themselves. Clients watch the run over the same WebSocket.

The runtime holds everything in memory. One running runtime holds any number of environments. An environment is one run of one model: it has its own metamodel, model, event definitions, state machines, settings, features, events and connected clients. Environments do not see each other.

The IDE starts a runtime and drives it for you, as 4.5 Running describes. Chapter 3 describes the runtime itself, for every client.

The life of an environment has these steps. Each step is one call of the HTTP API in 3.4 HTTP API.

1. A client creates the environment and gets its key, for example `env-1763035112345`. The environment is in the phase `configuring`.
2. The client uploads the metamodel, then the model, the event definitions, one attached state machine per object that gets one and the attachable state machines. The runtime reads every upload at once and refuses a text with errors, line by line. An accepted upload replaces the one before it.
3. The client sets the tick delay and the time to live of events, registers the WebSocket clients that observe objects or listen to event domains, and switches features on.
4. The client runs the environment. From now on its uploads, settings and observers are fixed. The runtime checks that the environment has a metamodel and a model, that every attached state machine belongs to an object of the model, and that every type that behaves after a state machine finds an attachable one that lists it. The environment is in the phase `activated`.
5. The client starts the environment. The state machines start in their initial states and tick until the environment is stopped. Objects that rules create or remove gain and lose state machine instances on the way. The phase is `running`. Events that clients send go to the state machines, and clients get the changes of the model and of the active states pushed over the WebSocket.
6. The client stops the environment. The run is frozen: the model, the state machines with their active states and the waiting events stay as they are, the phase is `stopped`, and the connections of the clients stay open.
7. The client may start the environment again. The run continues with the next tick as if there had been no stop.
8. The client drops the environment when it no longer needs it. Its engine, its Python processes and its connections end, and its key is gone. A client may drop an environment in any phase. An environment that is not dropped stays in the runtime, frozen when it is stopped, until the runtime ends.

The phase `halted` means that the run ended on its own after an error inside the runtime, without a stop call. The phase of an environment is part of every model snapshot of 3.4 HTTP API.

The sections of chapter 3:

Section	What it covers
3.1 Execution semantics	What happens in a tick: active states, transitions, actions, events, splits and joins, histories, tick limits and the life of state machine instances
3.2 Runtime features	The features that change a run: histories, tick limits, stepping, breakpoints, recording and replaying events, branches on threads of their own
3.3 Running the runtime	Starting the runtime on your machine or in Docker, its configuration file and its trace log
3.4 HTTP API	Every route, its parameters, its answers and its errors, with one whole session
3.5 WebSocket	Connecting, sending events and every message the runtime pushes
3.6 Examples	The example models that come with the runtime and how to run them

3.1 Execution semantics

The rules in this section say what the runtime does with a state machine while an environment runs: which states are active, when ENTRY and DO blocks run, which transition fires, what actions do to the model and how events travel. The syntax of the constructs is in 2.3 KStates. The tables of ticks in this section come from runs of the runtime with the state machines shown here.

3.1.1 Example state machine

Most examples use the lamp. The metamodel has a `Lamp` with a property `mode`, a `Bulb` in the relation `bulb` and a `Switch` in the relation `control`. The model has the lamp `lamp`, its bulb `bulb` and the wall switch `wallSwitch`, as in the IDE example `examples/lamp`. The event definitions have the events `press` and `dim`, and `dim` carries the property `bright`:

```
type "press" "Presses the wall switch" {
}
type "dim" "Turns the dimmer one step" {
    prop("bright", "boolean")
}
```

The state machine of the lamp in this section has the inner states `full` and `dimmed` with fixed watts:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            ENTRY {
                SET("mode", "off")
                SET("bulb->lit", "false")
```

```
                SET("bulb->watts", "0")
            }
        }
        STATE "on" {
            ENTRY {
                SET("mode", "on")
                SET("bulb->lit", "true")
            }
            STATE "full" {
                ENTRY { SET("bulb->watts", "40") }
            }
            STATE "dimmed" {
                ENTRY { SET("bulb->watts", "20") }
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "full" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "full" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "dimmed" -> "full" {
            CONDITION {
                EVENT("switch", "dim")
                VALUE("event->bright")
            }
        }
    }
}
```

The wall switch has its own state machine, attached to `wallSwitch`. It reacts to the same `press` events:

```
STATEMACHINE ATTACHED TO "Switch" {
    STATES {
        INITIAL STATE "down" { }
        STATE "up" {
            ENTRY {
                SET("position", "up")
                APPEND("presses", "up")
            }
        }
        STATE "back down" {
            ENTRY {
                SET("position", "down")
                APPEND("presses", "down")
            }
        }
    }
    TRANSITIONS {
        TRANSITION "down" -> "up" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "up" -> "back down" {
```

```
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "back down" -> "up" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

3.1.2 Ticks

An environment runs in ticks. Ticks are numbered from 1 on the first start; a start after a stop goes on with the next number, as 3.1.13.9 Stop and start, forcing and clients describes. In one tick the runtime does these steps in this order:

1. It applies the states that clients forced with `setActiveState` since the last tick, as 3.1.11 Forcing a state describes.
2. With the feature EventRecording recording, it hands the events that clients sent since the last tick to the state machines and records them under the number of this tick. With the feature EventInjection injecting, it hands over the injected events that are due. Both are in 3.2 Runtime features.
3. With the feature TickLimit on, it counts this tick for every active state.
4. It removes events whose time to live is over.
5. It runs one step of every state machine instance, one after the other, in the order of 3.1.13.2 The order of the steps: the attached state machines in the order of their first upload, then the instances of attachable state machines. An instance that ended earlier in the tick does no step. With the feature TrueParallel on, the branches of a split state machine do their steps at the same time, and this step ends when all of them are done.
6. It reports the active states of every state machine whose active states changed in this tick. In the first tick after the first start it reports every state machine.
7. It lets the instances that ended in this tick leave and the instances that rules asked for join, as 3.1.13.3 When instances join and leave describes.

The tick delay is the least time from the start of one tick to the start of the next one, in milliseconds. A tick that takes 30 ms with a tick delay of 500 ms is followed by a wait of 470 ms. A tick that takes longer than the tick delay is followed by the next tick at once. The environment takes the tick delay from `setTickDelay` or, without it, from the option `engine.defaultTickDelayMs` of 3.3 Running the runtime.

Events that clients send without EventRecording reach the state machines as soon as they arrive. The runtime looks for them every 10 ms, also in the middle of a tick, so a state machine later in the order can see an event that an earlier one did not see in the same tick.

The state machines of one environment run on one thread, one after the other. A state machine that waits for Python in an EVAL holds up all others of its environment. The feature TrueParallel gives each branch of a split a thread of its own, as 3.2.9 TrueParallel describes.

3.1.3 Active states

The active states of a state machine form a stack: an outer state, the state inside it, and so on down to the innermost active state. With the lamp in `dimmed`, the stack is `on`, `dimmed`. Outside a split, a state machine has one stack. In a split it has one stack per branch, as 3.1.8 Splits and joins describes. Clients see the stack in the message `stateChange` of 3.5 WebSocket.

3.1.3.1 Initial state

At the start, the runtime builds the stack level by level, by the rule of 2.3.2.2 The initial state, which the checks of dsl-tools and the IDE use as well. On each level it takes the state that is marked `INITIAL` or that holds the marked state somewhere inside it, and the first state of the level when there is none. It goes on below the chosen state while that state has inner states, and stops at a state without inner states.

The reader of KStates allows exactly one `INITIAL` in a state machine (check `kstates.initial-state-count`). So the state machine starts in the marked state and in every state around it, wherever these stand among their siblings. In the following text, the stack at the start is `on`, `dimmed`, although `off` is the first top level state, because the mark sits inside `on`:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        STATE "off" { }
        STATE "on" {
            STATE "full" { }
            INITIAL STATE "dimmed" { }
        }
    }
    TRANSITIONS { }
}
```

With `INITIAL STATE "on"` and no mark inside it, the stack at the start is `on`, `full`. All states of the start stack are entered in tick 1, from the outside in. A container state on the way counts as a state with inner states, so the start goes on into its fragment, as 3.1.14.3 Entering a container state describes.

3.1.3.2 Entering states

A state that becomes active is entered: its ENTRY block runs once. The ENTRY blocks of the states that a transition enters run in the tick after the transition, from the outermost to the innermost entered state. The ENTRY blocks of the start stack run in tick 1.

The DO block of the innermost active state runs once in every tick in which no transition fired before it, also in the tick in which the state was entered. The DO blocks of the outer states never run while an inner state is active.

States have no exit blocks. Nothing runs when a state is left.

Where the runtime goes for each way of entering a state and each kind of target, and in which tick, is gathered in 3.1.15 Where the engine goes.

3.1.3.3 Steps of a tick

In its step of a tick, a state machine does these things in this order. The first transition that fires ends the step.

1. With the feature History on, it notes the active inner states of every state that has a history.
2. It enters the states not yet entered, from the outside in. Before each ENTRY block a breakpoint on the entry of the state can halt the run. After each ENTRY block that has at least one action, it looks for a transition that can fire, over the whole stack. If one fires, the ENTRY blocks of the states further inside do not run.
3. With the feature TickLimit on, it fires the timeout of a state whose time is over.
4. With the feature History on, it leads on into the remembered inner state when the innermost active state has a history and something to remember.
5. It runs the DO block of the innermost active state.
6. It looks for a transition that can fire and fires it.
7. When no transition fired, it consumes the events that a transition of an active state waited for but could not use, as 3.1.7.3 Consumption describes.

The lamp and the switch, with a press sent before tick 3, a `dim` with `bright` false before tick 5 and a press before tick 7:

Tick	Event	lamp	wallSwitch
1		ENTRY of `off` runs. Stack `off`	Stack `down`
2		Stack `off`	Stack `down`
3	press	`"off" -> "full"` fires. Stack `on`, `full`	`"down" -> "up"` fires. Stack `up`
4		ENTRY of `on` and of `full` run. `watts` is 40	ENTRY of `up` runs
5	dim	`"full" -> "dimmed"` fires. Stack `on`, `dimmed`	Stack `up`
6		ENTRY of `dimmed` runs. `watts` is 20	Stack `up`
7	press	`"on" -> "off"` fires. Stack `off`	`"up" -> "back down"` fires
8		ENTRY of `off` runs. `watts` is 0	ENTRY of `back down` runs

After tick 8 the list `presses` of the switch is `["up", "down"]`.

A transition can fire right after an ENTRY block. In the next run the `dim` arrives while the lamp is off, waits there and fires right after the ENTRY of `on`. The ENTRY of `full` never runs:

Tick	Event	lamp
2	dim	Stack `off`. No transition of `off` waits for `dim`, so the event stays
4	press	`"off" -> "full"` fires. Stack `on`, `full`
5		ENTRY of `on` runs, then `"full" -> "dimmed"` fires on the waiting `dim`. Stack `on`, `dimmed`
6		ENTRY of `dimmed` runs. `watts` is 20

3.1.4 Transitions

3.1.4.1 Which transition fires

The runtime looks for a transition from the innermost active state outwards. For each state it tries the transitions that start at this state, `TRANSITION` and `SPLIT`, in the order of the file. The first one whose condition holds fires. At most one transition fires per stack in one step. A transition that starts at an outer state applies while any of its inner states is active: `"on" -> "off"` switches the lamp off from `full` and from `dimmed`. A transition of an inner state wins over one of an outer state when both could fire.

A `JOIN` is tried before all other transitions, as 3.1.8 Splits and joins describes.

3.1.4.2 Conditions

A transition without `CONDITION` block fires whenever it is tried. A condition holds when all of its clauses hold.

`EVENT("switch", "press")` holds when an event named `press` in the domain `switch` is there for this state machine. When several such events are there, the runtime tries them from the oldest to the newest and takes the first one for which the other clauses hold. The other clauses see this event as `event`. A condition has at most one `EVENT` clause. It may stand anywhere among the clauses; the runtime treats it the same as a first clause.

`VALUE("path")` holds when the path leads to the boolean `true` or to the string `true` in any case. `VALUE("true")` always holds and `VALUE("false")` never. A path starts at the object of the state machine, `VALUE("bulb->lit")`, or at the event, `VALUE("event->bright")`. A path that leads nowhere makes the clause false.

`EVAL { ... }` holds when its Python code returns `True`. Any other result and any error make it false; the error goes to the console of the runtime. 3.1.6 EVAL and macros has the details.

A condition without `EVENT` clause sees as `event` the event that led into the current state, as 3.1.7.5 Scoped events describes.

`DONE` holds when the container state the transition leaves rests on a final state of its fragment, as 3.1.14.5 Final states and DONE describes.

In the lamp, `"dimmed" -> "full"` needs a `dim` with `bright` true. A `dim` with `bright` false in `dimmed` fires nothing and is consumed:

Tick	Event	lamp
2	press	`"off" -> "full"` fires
3		ENTRY of `on` and `full` run
4	dim, bright false	`"full" -> "dimmed"` fires
5		ENTRY of `dimmed` runs
6	dim, bright false	Nothing fires. The event is consumed
7	dim, bright true	`"dimmed" -> "full"` fires
8		ENTRY of `full` runs. Stack `on`, `full`

3.1.4.3 Where a transition leads

After a transition, the stack is the path from the top level down to the target state. The runtime does not descend into the inner states of the target: `"full" -> "on"` leaves the stack `on`, and from then on the DO block of `on` runs. A container state is the one exception: the runtime goes down into its fragment, as 3.1.14.3 Entering a container state describes.

The runtime enters the states of the new stack that were not on the old stack. A transition from a state to itself, and a transition from a state to one of its outer states, enter the target again together with the state around it. With the stack `on`, `full`, the transition `"full" -> "full"` enters `on` and `full` again, and `"full" -> "on"` enters `on` again. A transition to itself on the top level, `"on" -> "on"`, enters `on` again.

3.1.4.4 NOT LOOPING

A transition marked `NOT LOOPING` does not fire when it was the last transition that fired in its stack. Then the runtime tries the next one. With this state machine

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            ENTRY { SET("mode", "on") }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" { }
        TRANSITION "on" -> "on" NOT LOOPING { }
        TRANSITION "on" -> "off" { }
    }
}
```

the lamp runs in a circle of three ticks:

Tick	lamp
1	`"off" -> "on"` fires
2	ENTRY of `on` runs, then `"on" -> "on"` fires
3	ENTRY of `on` runs. `"on" -> "on"` was the last transition, so `"on" -> "off"` fires
4	`"off" -> "on"` fires

3.1.5 Actions

3.1.5.1 Blocks and errors

An ENTRY or DO block runs its actions in order. The changes of one block reach the clients together, after the block, as the messages `objectChanged` and `objectDeleted` of 3.5 WebSocket.

An action that fails stops the rest of its block. The changes of the actions before it stay. The runtime writes the error to its console and to the trace log, and the run goes on. In this ENTRY block the second action fails because `Bulb` has no property `colour`. `mode` becomes `broken` and `lit` stays as it was:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            ENTRY {
                SET("mode", "broken")
                SET("bulb->colour", "red")
                SET("bulb->lit", "true")
            }
        }
    }
    TRANSITIONS { }
}
```

The trace log has the line:

```
ACTION_ERROR: Error in onEntry for state 'off': 'colour' on 'Bulb' is not a scalar property
    ↪ (use SETOBJ for relations)
```

3.1.5.2 Values on the right side

The value of an action is a string, an `EVAL` block or a `MACRO` call.

A string is read as the type of the property it goes to. For a property of type `number`, `"40"` is the number 40, and a string that is no number is an error of the action. For a property of type `boolean`, the string must be `true` or `false`. For a property of type `string`, the string is taken as it is.

An `EVAL` block or a `MACRO` call computes the value, as 3.1.6 EVAL and macros describes. When it returns nothing, the action does nothing.

3.1.5.3 Properties and lists

A path names a property of the object of the state machine, `"mode"`, or of an object reached over relations, `"bulb->watts"`.

Action	What it does
`SET("path", value)`	Sets a property that holds one value
`APPEND("path", value)`	Adds the value at the end of a list property. A list that has no value yet is created
`SETLIST("path", value)`	Replaces the whole list. A string gives a list of one element, an `EVAL` or `MACRO` gives the list it returns
`DROPLIST("path")`	Removes the list property

```
STATEMACHINE ATTACHED TO "Switch" {
    STATES {
        INITIAL STATE "down" {
            ENTRY {
                SET("position", "down")
                SETLIST("presses", "reset")
                APPEND("presses", "down")
            }
        }
    }
    TRANSITIONS { }
}
```

After the ENTRY block, `presses` is `["reset", "down"]`.

3.1.5.4 Objects and relations

Action	What it does
`SETOBJ("source", "relation", value)`	Sets a relation that holds one object
`APPENDOBJ("source", "relation", value)`	Adds an object to a relation that holds a list
`DROPOBJ("path")`	Removes an object from the model
`SETOBJ(...) ATTACH name`, `APPENDOBJ(...) ATTACH name`	Creates the object with an instance of the attachable state machine `name`
`EXCHANGE("path", name)`	Gives the object at the path an instance of the attachable state machine `name`

The source is the path to the object that owns the relation. `""` and `"self"` stand for the object of the state machine.

A `has` relation needs a new object. The value is an `EVAL` or `MACRO` that returns the object as a dictionary with the properties of its type. The runtime builds the object; its key is the entry `__id__` of the dictionary or, without it, `gen_<Type>_<n>`. A dictionary whose `__id__` is the key of an object that already exists changes that object in place. `SETOBJ` on a `has` relation removes the object that was there before, with everything it holds.

A `knows` relation needs an object that exists. The value is a string with the path from the object of the state machine to that object, or an `EVAL` or `MACRO` that returns a dictionary with its `__id__` or the key as a string.

The type of the object must be the type of the relation. A `has` relation given an existing object, a `knows` relation given a new one, and an object of the wrong type are errors of the action.

A dictionary with the `__id__` of an existing object changes that object in place, wherever an `EVAL` or a `MACRO` returns it, also nested inside another dictionary. Its properties take the values of the dictionary. A `has` relation in the dictionary holds afterwards the objects the dictionary lists: an object it held before and the dictionary does not list leaves the model as with `DROPOBJ`, with everything it holds, and a dictionary without `__id__`, or with an `__id__` no object has, is a new object. An object that another object holds cannot be listed; that is an error of the action. A `knows` relation in the dictionary must list objects that exist. So the lamp is given a new bulb, and the old bulb is removed, by:

```
SETOBJ("self", "light", EVAL { return {"__id__": "lamp", "bulb": {"maxWatts": 60, "watts":
    ↪ 0}} })
```

where `light` is a `knows` relation of the object of the state machine to the lamp.

`DROPOBJ` removes the object at the path with everything it holds. The runtime removes every `knows` reference to a removed object. The object of the state machine itself cannot be removed by its own path. The state machine instances of all removed objects end, also those of the objects that a replacing `SETOBJ` removes. New objects get the instances their types behave after, and `ATTACH` and `EXCHANGE` give objects other instances, as 3.1.13 State machine

management and lifecycle describes. The lamp of the IDE example drops its spare bulb when it breaks:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "working" { }
        STATE "broken" {
            ENTRY {
                DROPOBJ("spare")
                SET("mode", "broken")
            }
        }
    }
    TRANSITIONS {
        TRANSITION "working" -> "broken" {
            CONDITION { EVENT("service", "smash") }
        }
    }
}
```

3.1.5.5 Raising events

`EVENT("domain", "name")` as an action raises an event without payload. It is on the bus at once with the time to live of the environment. State machines later in the order see it in the same tick, the state machine that raised it sees it when it next looks for a transition. In this run the switch, which comes first, raises `press` when it enters `up`, and the lamp reacts in the same tick:

```
STATEMACHINE ATTACHED TO "Switch" {
    STATES {
        INITIAL STATE "down" { }
        STATE "up" {
            ENTRY {
                SET("position", "up")
                EVENT("switch", "press")
            }
        }
    }
    TRANSITIONS {
        TRANSITION "down" -> "up" {
            CONDITION { EVENT("button", "push") }
        }
    }
}
```

Tick	Event	wallSwitch	lamp
2	push in domain `button`	`"down" -> "up"` fires	Stack `off`
3		ENTRY of `up` runs and raises `press`	`"off" -> "full"` fires
4		Stack `up`	ENTRY of `on` and `full` run

3.1.5.6 IF IN SCOPE and WITH

`IF IN SCOPE("path", ...) { ... }` runs its actions only when every path leads to a value. A path into `event` needs a scoped event that has this property. A property with the value `false` counts as a value.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            ENTRY {
                IF IN SCOPE("event->bright") {
                    SET("mode", "dimmed by event")
                }
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "dim") }
        }
    }
}
```

A `dim` switches the lamp on and sets `mode` to `dimmed by event`.

`WITH MACRO("name", ...) AS "binding" { ... }` computes the macro once and runs the actions in the block. The actions read the result as `$(binding)` and `$(binding->property)` in their `EVAL` blocks and paths.

3.1.6 EVAL and macros

An `EVAL` block holds Python code that ends with `return`. Before the code runs, the runtime replaces every `$(path)` by the value at the path as a Python value. A number becomes a Python number, a string a Python string, a boolean `True` or `False`, a list a Python list and an object a dictionary with the entries `__id__`, `__type__`, its properties and its relations, with the objects of the relations as dictionaries again. `$(event->bright)` reads the event in scope. A path that cannot be followed, such as `event->bright` with no event in scope, is an error of the `EVAL`.

The value that the code returns goes to the action or the condition. In a condition only `True` makes the clause hold.

An `EVAL` whose code is one `return` of numbers, `True`, `False`, arithmetic, comparisons, `and`, `or`, `not` and parentheses, once the paths are filled in, is computed by the runtime itself. Every other `EVAL` runs in Python. The runtime calls `python3`, `python` or `py`, whichever it finds first on the path. Code with `import` or `def` runs in a Python process that stays open for the run of the environment. With the feature TrueParallel, branches that run such code at the same time each get a process of their own.

The lamp of the IDE example computes its watts this way:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
```

```
            ENTRY {
                SET("bulb->watts", EVAL { return $(bulb->maxWatts) })
            }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION {
                EVENT("switch", "press")
                EVAL { return $(bulb->watts) + 10 <= $(bulb->maxWatts) }
            }
        }
    }
}
```

A macro is a named piece of Python code in the `MACROS` block of the file. `MACRO("name", "path", ...)` calls it. Each argument is a path from the object of the state machine, from `event` or from a binding of `WITH`, and its value is handed over for the parameter of the same place in `TAKES`. In the code of the macro, `$(parameter)` is the value of the parameter and `$(parameter->property)` reads into it. The result must fit the type in `RETURNS`. Inside Python code, `@name(arguments)` calls another macro of the file as a Python function.

3.1.7 Events

3.1.7.1 Domains and names

Every event has a domain and a name, `switch` and `press`. A transition waits for an event with `EVENT("switch", "press")`. Events come from clients over the WebSocket, from the feature EventInjection and from the action `EVENT`. The domain is free text. Clients that listen to a domain get every event of that domain pushed, as 3.5 WebSocket describes.

3.1.7.2 Time to live

An event expires when it is older than its time to live. The time to live of an environment comes from `setEventTtl` or, without it, from the option `engine.defaultEventTtlMs`. A time to live of 0 means that events never expire. The runtime removes expired events at the start of each tick and every 10 ms in between.

3.1.7.3 Consumption

Each state machine sees every event once. An event that the lamp consumed is still there for the switch. Within one state machine an event is consumed in one of these ways:

1. It fires a transition. The option `engine.eventConsumptionOnFire` is `true` by default. With `false`, the event that fired a transition stays and can fire further transitions in later ticks.
2. No transition fired in the step, and a transition of an active state waits for an event with this domain and name. Then the event is consumed, also when the other clauses of the transition did not hold.
3. It expires.

An event that no transition of the active states waits for stays until it expires, and fires a transition when a state that waits for it becomes active within its time to live. The `dim` in the second table of 3.1.3.3 Steps of a tick waits in `off` this way.

3.1.7.4 Payloads

An event from a client carries a payload as JSON or as KModel text. The runtime reads it against the type of the event definitions that has the name of the event. A `dim` with the payload `{"bright": true}` gives the event the property `bright` with the value `true`. A payload that cannot be read leaves the event without properties: JSON that does not fit the event type, and KModel text with any error of the KModel reader, such as a syntax error, an unknown property or a value of the wrong type. The runtime writes the errors to its console. The event still fires transitions by its name, paths into `event` lead nowhere, and the run goes on. The forms of the message are in 3.5 WebSocket.

Events raised with the action `EVENT` have no payload. Events replayed by EventInjection are not read against the event definitions, so paths into them lead nowhere.

3.1.7.5 Scoped events

The event that fired a transition becomes the scoped event of the target state. While this state is active, its ENTRY and DO blocks, its `IF IN SCOPE` paths and the conditions without `EVENT` clause read it as `event`. An inner state that was entered without an event of its own reads the scoped event of the nearest outer state. A state loses its scoped event when it is left.

3.1.7.6 Several state machines

All state machines of an environment share one bus of events. Each takes the events it waits for and consumes them for itself only. The lamp and the switch both react to the same `press` in the same tick in the first table of 3.1.3.3 Steps of a tick. This holds for every instance: two objects with instances of the same attachable state machine consume their events each for itself, and an instance that joins during the run sees the events on the bus whose time to live is not over.

3.1.8 Splits and joins

A `SPLIT` with its condition holding replaces the one stack of the state machine with one branch per target state. Each branch has the stack from the top level down to its target and enters its states in the next tick. The event that fired the split is the scoped event of every target.

While the state machine has branches, it first tries every `JOIN`. A join fires when its condition holds and each of its source states is active in a different branch, with exactly as many branches as the join has sources. Then the state machine has one stack again, the path to the target of the join, and the other branches do no more step in this tick. When no join fires, each branch does its step as in 3.1.3.3 Steps of a tick, one branch after the other in the order of the targets of the split, on the one thread of the environment. With the feature TrueParallel on, the branches do their steps at the same time on threads of their own, as 3.2.9 TrueParallel describes. The stack that clients see is the stacks of all branches one after the other.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "test bulb" {
            ENTRY { SET("bulb->lit", "true") }
        }
        STATE "bulb ok" {
            ENTRY { SET("bulb->lit", "false") }
        }
        STATE "test switch" { }
```

```
        STATE "check position" { }
        STATE "switch ok" {
            ENTRY { SET("mode", "tested") }
        }
        STATE "on" {
            ENTRY { SET("mode", "on") }
        }
    }
    TRANSITIONS {
        SPLIT "off" -> "test bulb", "test switch" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "test bulb" -> "bulb ok" { }
        TRANSITION "test switch" -> "check position" { }
        TRANSITION "check position" -> "switch ok" { }
        JOIN "bulb ok", "switch ok" -> "on" { }
    }
}
```

Tick	Event	Branch 1	Branch 2
2	press	The split fires. Stack `test bulb`	Stack `test switch`
3		ENTRY of `test bulb` runs, then `"test bulb" -> "bulb ok"` fires	`"test switch" -> "check position"` fires
4		ENTRY of `bulb ok` runs	`"check position" -> "switch ok"` fires
5		The join fires. Stack `on`	
6		ENTRY of `on` runs	

The ENTRY of `switch ok` never runs, because the join fires before the branch enters it, so `mode` never becomes `tested`.

A join sees the events of all branches. It consumes its event for the new stack only.

A split into container states goes down into each of them, so every branch starts inside its fragment. `DONE` on a join waits for all container states among its sources, as 3.1.14.5 Final states and DONE describes.

The trace log gives every branch a path number of its own, as 3.3.5 Trace log describes. In the example the lamp runs on path 0 up to the split, branch 1 on path 1, branch 2 on path 2, and the stack `on` after the join on path 3. A second split would start paths 4 and 5. Every line of a branch names its path, so the lines of the two branches can be told apart, also when they interleave with TrueParallel on.

3.1.9 Histories

With the feature History on, a state marked `SHALLOW HISTORY` or `DEEP HISTORY` remembers where it was. At the start of each step the runtime notes, for every active state with a history that has an active inner state, the inner state: the state right inside it for `SHALLOW HISTORY`, the innermost active state for `DEEP HISTORY`. When a transition later leads into the state itself, the state is entered, and in the same tick the runtime leads on into the remembered state. That ends the step; the remembered state is entered in the next tick. Each state machine instance has its own histories. A history stays when the environment is stopped and started again, and is forgotten when its instance leaves. Without the feature History, the marks have

no effect. A container state with a history goes to the remembered state at once, as 3.1.14.6 Histories, tick limits and breakpoints describes.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        STATE "off" {
            ENTRY { SET("mode", "off") }
        }
        INITIAL SHALLOW HISTORY STATE "on" {
            ENTRY { SET("mode", "on") }
            STATE "full" {
                ENTRY { SET("bulb->watts", "40") }
            }
            STATE "dimmed" {
                ENTRY { SET("bulb->watts", "20") }
            }
        }
    }
    TRANSITIONS {
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "full" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
    }
}
```

Tick	Event	lamp
1		ENTRY of `on` and `full` run
2	dim	`"full" -> "dimmed"` fires
3		ENTRY of `dimmed` runs
4	press	`"on" -> "off"` fires
5		ENTRY of `off` runs
6	press	`"off" -> "on"` fires. Stack `on`
7		ENTRY of `on` runs, then the history leads on to `dimmed`
8		ENTRY of `dimmed` runs. `watts` is 20

3.1.10 Tick limits

With the feature TickLimit on, a `TICK LIMIT` block counts the ticks of its state. The tick in which the ENTRY block of the state runs is its first tick. Each state machine instance counts its own ticks. The counts stay when the environment is stopped and started again, and are forgotten when the instance leaves. Without the feature TickLimit, `TICK LIMIT` blocks have no effect.

`DELAY: n TICKS` holds the state for its first n ticks. A transition that would leave the state in that time fires nothing. Its event stays, and no other transition fires in this step. `TIMEOUT: AFTER n TICKS TRANSITION TO "state"` leads to the named state in the tick after the first n ticks, after the ENTRY blocks and before the DO block. When several active states have a

timeout that is due, the innermost one wins. A delay holds back a timeout as well.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            TICK LIMIT { DELAY: 2 TICKS }
            ENTRY { SET("mode", "off") }
        }
        STATE "on" {
            TICK LIMIT { TIMEOUT: AFTER 3 TICKS TRANSITION TO "off" }
            ENTRY { SET("mode", "on") }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

Tick	Event	lamp
1		ENTRY of `off` runs. First tick of `off`
2	press	Second tick of `off`. The delay holds the transition, the press stays
3		`"off" -> "on"` fires on the waiting press
4		ENTRY of `on` runs. First tick of `on`
5		Second tick of `on`
6		Third tick of `on`
7		The timeout leads to `off`
8		ENTRY of `off` runs
9	press	Second tick of `off`. The delay holds the transition
10		`"off" -> "on"` fires

3.1.11 Forcing a state

The route `setActiveState` of 3.4 HTTP API puts a state machine into a named state while the environment runs. The runtime applies it at the start of the next tick. The stack becomes the path to the named state, with one branch. A state inside a container state is named with its qualified name, such as `self test/lit`, and a container state is gone down into as by a transition, see 3.1.14.3 Entering a container state. No ENTRY block runs. The runtime drops every event of the domain whose name is the key of the object of the state machine and every client event that has not reached the state machines yet. The state machine is the instance the object has in the latest tick, also one that joined during the run. In the trace log the forced state ends the paths the state machine ran and starts the next path; the line `Forced state` names both.

3.1.12 Errors in a tick

An error in an action stops its block, as 3.1.5.1 Blocks and errors describes. An error in an `EVAL` or `VALUE` clause makes the clause false. In all these cases the run goes on. The runtime writes the error to its console; errors of actions also go to the trace log as `ACTION_ERROR`.

3.1.13 State machine management and lifecycle

An environment runs state machine instances. An instance is one state machine running on one object, with its own stack, branches, events, histories, tick counts and breakpoint overrides. One object has at most one instance at a time, so the key of the object names the instance. This section tells when instances begin, in which order they step and when they end. How to write attachable state machines, `behaves after`, `ATTACH` and `EXCHANGE` is in 2.3.11 State machine management and lifecycle.

3.1.13.1 Where instances come from

An environment holds two kinds of state machines. An attached state machine, `STATEMACHINE ATTACHED TO "Type"`, is uploaded with `attachedTo` and belongs to the object with that key. An attachable state machine, `STATEMACHINE "name" ATTACHABLE TO "Type", ...`, is uploaded without `attachedTo` and kept as a definition under its name. 3.4.3 Uploads has both forms.

An object gets an instance of a definition in four ways:

- its type `behaves after` the definition and the object is in the model when the environment starts;
- its type `behaves after` the definition and a rule creates the object;
- an `ATTACH` clause names the definition for the object its rule creates;
- `EXCHANGE` gives the object the definition.

An attached state machine wins over the `behaves after` of the type of its object. An `ATTACH` clause wins over the `behaves after` of the type of the object the rule creates. Every new instance starts in its initial state, as 3.1.3.1 Initial state describes.

`runEnvironment` refuses an environment in which a type of the metamodel behaves after a name that no attachable state machine has, or after one that does not list the type, as 3.4.7 Running shows. So every object of such a type can get its instance, also one that a rule creates later.

3.1.13.2 The order of the steps

The instances step one after the other in a fixed order:

1. the attached state machines, in the order of their first upload;
2. the instances for `behaves after` of the objects in the model at the start, in the order of the objects in the model: an object first, then the objects it holds, relation by relation, as the KModel text writes them;
3. the instances that joined during the run, in the order in which they joined.

An instance that `EXCHANGE` gives an object takes the place of the instance it replaces. An object without an instance that gets one through `EXCHANGE` joins at the end.

3.1.13.3 When instances join and leave

The order changes only at the end of a tick, after every instance has done its step. A rule that creates an object asks for the instances of the new objects, and `EXCHANGE` asks for the new instance. At the end of the tick these instances join. Each does its first step in the next tick, so its ENTRY blocks run there.

An instance ends at the moment its object leaves the model or `EXCHANGE` replaces it. It does no further step from that moment: an instance later in the order skips its step in this tick. An instance that ends during its own action block, because the block removed an object that holds its own object or exchanged its own state machine, finishes the block, and then its step ends without looking for a transition. At the end of the tick the ended instances leave the order. Its histories, tick counts and the overrides of breakpoints for its object leave with it.

The same holds with the feature TrueParallel on, where the branches of a split create, remove and exchange objects at the same time on threads of their own: the requests of all branches are collected and carried out together at the end of the tick. A request for an object that is no longer in the model at that point is dropped. With TrueParallel on, an instance that ends during a step stops each of its branches after the block that branch is in.

3.1.13.4 New objects

`SETOBJ` and `APPENDOBJ` on a `has` relation create an object and every object it holds, as 3.1.5.4 Objects and relations describes. A dictionary with the `__id__` of an existing object creates the new objects it lists in a `has` relation in the same way. Every new object whose type behaves after a definition gets an instance of it. With `ATTACH name` after the rule, the object the rule creates gets an instance of `name` instead; the objects it holds still follow their types.

The runtime reads the name of `ATTACH` before it computes the new object and checks it against the type of the relation. A name that fails is an error of the action, and the rule creates nothing.

3.1.13.5 EXCHANGE

`EXCHANGE("path", name)` ends the instance of the object at the path and gives the object an instance of `name`. The path is read like the path of `DROPOBJ`: it starts at the object of the state machine and `self` is no keyword in it. The object at the path may be the object of the state machine itself, reached over relations; then the instance finishes its block and leaves, as 3.1.13.3 describes. `EXCHANGE` with the name of the running definition starts the object again in the initial state of a new instance. Two `EXCHANGE` on one object in one tick leave the object with the instance of the second.

The trace log shows an exchange as one line `STATE_MACHINE_EXCHANGED`, which names the object, the state machine that ended and the one that started, the paths that ended and the path of the new instance. Two `EXCHANGE` on one object in one tick give one such line, from the state machine before the tick to the one named last. `EXCHANGE` on an object without an instance gives the line `STATE_MACHINE_ATTACHED`. An instance that joins after another one left the same object, for example after the object was removed and created again with the same key, goes on with the next path number of the object. The lines are described in 3.3.5 Trace log.

3.1.13.6 Removed objects

An object leaves the model through `DROPOBJ`, through a `SETOBJ` that replaces the object of a `has` relation, and through a dictionary with the `__id__` of the object that holds it, when the dictionary no longer lists it, as 3.1.5.4 Objects and relations describes. In all cases every object it holds leaves with it, and the instances of all these objects end. An instance that a rule asked for in the same tick for one of these objects never joins. `DROPOBJ` still cannot remove the object of its own state machine by its own path, but it can remove an object that holds it.

3.1.13.7 Errors

A name is a string, an `EVAL` or a `MACRO`, as 3.1.5.2 Values on the right side describes. A string is the name itself; an `EVAL` and a `MACRO` must return a string. These are errors of the action, handled like the others of 3.1.5.1 Blocks and errors:

Error	Text in the trace log
The value is not a string	`The name of a state machine must be a string, but the value is <value>`
No definition has the name	`No attachable state machine is named '<name>'`
The definition does not list the type of the object	`The state machine '<name>' is attachable to '<A>', '<B>', not to '<type>'`
`ATTACH` after a `knows` relation	`ATTACH needs a has relation, but '<relation>' is a knows relation; use EXCHANGE for an existing object`
The path of `EXCHANGE` leads to no object	`EXCHANGE path '<path>' does not resolve to an object`

The type of `ATTACH` is the type of the relation, the type of `EXCHANGE` the type of the object at the path. A failed `ATTACH` fails its whole rule, and a failed `EXCHANGE` leaves the instance of the object as it was.

3.1.13.8 Example

The example uses the metamodel of 2.3.11: `Bulb` behaves after `glowing`, and `Room` has spare bulbs. The model has the room `living room` with the lamp `lamp`, its bulb `bulb` with `maxWatts` 40 and `watts` 0, and the switch `wallSwitch`. The environment holds the attachable state machines `glowing` and `blinking` of 2.3.11, and the state machine of the room, also from 2.3.11, is attached to `living room`:

```
STATEMACHINE ATTACHED TO "Room" {
    STATES {
        INITIAL STATE "waiting" { }
        STATE "stocked" {
            ENTRY {
                APPENDOBJ("self", "spares", MACRO("new bulb", "lamp->bulb->maxWatts"))
    ↪ ATTACH "blinking"
            }
        }
        STATE "party" {
            ENTRY {
                EXCHANGE("lamp->bulb", "blinking")
            }
        }
    }
```

```
    TRANSITIONS {
        TRANSITION "waiting" -> "stocked" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "stocked" -> "party" {
            CONDITION { EVENT("switch", "press") }
        }
    }
    MACROS {
        MACRO "new bulb" {
            TAKES("watts", "number")
            RETURNS("Bulb")
            DEFINITION {
                EVAL { return {"maxWatts": $(watts), "watts": $(watts)} }
            }
        }
    }
}
```

At the start the room gets its attached state machine and `bulb` an instance of `glowing`, in this order. The new spare bulb gets the key `gen_Bulb_0`. A press comes before tick 3 and one before tick 6:

Tick	Event	living room	bulb	gen_Bulb_0
1		Stack `waiting`	ENTRY of `dark` of `glowing` runs. `watts` is 0, so it stays	
3	press	`"waiting" -> "stocked"` fires	Stack `dark`	
4		ENTRY of `stocked` creates `gen_Bulb_0` with `ATTACH "blinking"`	Stack `dark`	Joins at the end of the tick, after `bulb`
5		Stack `stocked`	Stack `dark`	ENTRY of `off` runs, then `"off" -> "on"` fires
6	press	`"stocked" -> "party"` fires	Stack `dark`	ENTRY of `on` runs, then `"on" -> "off"` fires
7		ENTRY of `party` exchanges the state machine of `bulb`	No step. Leaves at the end of the tick, an instance of `blinking` takes its place	ENTRY of `off` runs
8		Stack `party`	ENTRY of `off` of `blinking` runs, then `"off" -> "on"` fires	ENTRY of `on` runs

In tick 8 the order is `living room`, `bulb`, `gen_Bulb_0`. The trace log shows the change of tick 7 as one line, `STATE_MACHINE_EXCHANGED: State machine 'glowing' exchanged for 'blinking'` for `bulb`, with `ends=0, path=1`. A `DROPOBJ("lamp")` in the room would end the instance of `bulb` in the same way, because the lamp holds the bulb, and nothing would take its place.

3.1.13.9 Stop and start, forcing and clients

Only the first start builds instances over the model. A stop freezes the environment, and a start after it continues the run with the next tick, as if there had been no stop. Everything stays as the stop found it:

- every instance, also those that `ATTACH` and `EXCHANGE` gave objects, with its place in the order;
- the stacks and branches of every instance, the events it consumed and its scoped events;
- the histories, the counts of tick limits, and the state and overrides of breakpoints;
- the number of the tick, and with it the lists of EventRecording and EventInjection;
- the path numbers of the trace log, so the branches go on with their paths and new ones count on;
- the pause of TickByTick and a halt at a breakpoint;
- the events on the bus, which keep the time to live they had left at the stop, because the time while the environment is stopped does not count.

A stopped environment stays frozen in the runtime until it is started, dropped with `POST /dropEnvironment` of 3.4.2 Environments, or the runtime ends. A drop ends it for good, also when it is stopped while a breakpoint halts it.

Nothing runs again that ran before the stop: no ENTRY block runs twice, and no instance starts anew. A stop lets the running tick end first. When the run waits at a breakpoint inside a tick, the halted tick stays as it is until the start and a `resume`. Events that clients send while the environment is stopped wait and reach the state machines after the start.

`setActiveState` of 3.1.11 Forcing a state works on the instance an object has in the latest tick, also on one that joined during the run.

Clients learn about instances through the messages `stateMachineAttached` and `stateMachineDetached` of 3.5 WebSocket, which name the object and the definition, and through the member `stateMachines` of the model snapshot. At the first start every instance is reported before the first tick; a start after a stop reports nothing anew. An instance that joins is reported at the end of the tick it joined in, its first stack after its first step. An instance that leaves is reported at the end of the tick it ended in, and its stack disappears from the snapshot.

3.1.14 Pages

Container states, fragments, final states and the clause `DONE` split a state machine into pages. How to write them is in 2.3.12 Pages. They belong to the core language and need no feature.

3.1.14.1 Flattening at activation

`runEnvironment` flattens every attached state machine and every attachable state machine of the environment with the fragments of the environment, before anything runs. A container state then holds a copy of the states of its fragment as its inner states, and a fragment used by two container states runs as two copies that share nothing. The engine runs only these flat state machines. A state machine that does not flatten, for example because its container state uses a fragment the environment does not hold, refuses the activation, as 3.4.7 Running shows. The environment then stays configurable for the missing upload and another `runEnvironment`.

An attachable state machine is flattened once. Every instance of it runs the same flat state machine, whether it comes from `behaves after`, `ATTACH` or `EXCHANGE`. The texts the environment keeps are the uploaded ones, and `observatory/statemachine` answers them, not the flat state machines.

3.1.14.2 Qualified names

A state of a fragment runs under the name of its container state, a slash and its name in the fragment. The state `lit` of the fragment `bulb test` below runs as `self test/lit`. When a fragment uses a fragment again, the names nest, as in `self test/wiring/lit`. A state that stands inside another state of the fragment keeps its name in the fragment behind the container state: `high` inside `second` runs as `self test/high`. The states of the state machine itself keep their names.

The trace log, the message `stateChange`, the member `activeStates` of the snapshot and `setActiveState` use these names. A state machine without container states looks exactly as before.

3.1.14.3 Entering a container state

Whenever a container state is entered, the runtime goes down into it at once: into the states its fragment starts in, by the rule of 3.1.3.1 Initial state, and further down while one of them is a container state again. This holds for every way a state is entered: the start of an instance, a transition, a split, a join, the timeout of a tick limit, a history and `setActiveState`, and for the instances that `behaves after`, `ATTACH` and `EXCHANGE` start. A stack therefore never ends on a container state. A state with inner states written in its braces is different: a transition enters it as itself, as 3.1.4.3 Where a transition leads describes.

The states entered below the container state are entered in the next tick, from the outside in, like the other states a transition enters. A container state has no ENTRY and no DO block.

The examples use this fragment:

```
STATEMACHINE "bulb test" FRAGMENT {
    STATES {
        INITIAL STATE "lit" {
            ENTRY { SET("bulb->lit", "true") }
        }
        STATE "dark" {
            ENTRY { SET("bulb->lit", "false") }
        }
        FINAL STATE "passed" { }
    }
    TRANSITIONS {
        TRANSITION "lit" -> "dark" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "dark" -> "passed" {
            CONDITION { EVENT("switch", "dim") }
        }
    }
}
```

The lamp runs it in its container state `self test`:

```
STATEMACHINE ATTACHED TO "Lamp" {
```

```
    STATES {
        INITIAL STATE "off" {
            ENTRY { SET("mode", "off") }
        }
        CONTAINER STATE "self test" USE "bulb test" { }
        STATE "on" {
            ENTRY { SET("mode", "on") }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "self test" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "self test" -> "on" {
            CONDITION { DONE }
        }
        TRANSITION "self test" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

With a press before tick 2 and a `dim` before tick 4 and before tick 6:

Tick	Event	lamp
1		ENTRY of `off` runs. Stack `off`
2	press	`"off" -> "self test"` fires. Stack `self test`, `self test/lit`
3		ENTRY of `self test/lit` runs. `lit` is true
4	dim	`"self test/lit" -> "self test/dark"` fires
5		ENTRY of `self test/dark` runs. `lit` is false
6	dim	`"self test/dark" -> "self test/passed"` fires. Stack `self test`, `self test/passed`
7		`DONE` holds and `"self test" -> "on"` fires. Stack `on`
8		ENTRY of `on` runs

3.1.14.4 Leaving a container state

The container state is active around the states of its fragment, so its transitions apply in every one of them, as for any outer state. A transition that leaves the container state drops all its inner states. A press during the self test above leads back to `off` from `self test/lit` and from `self test/dark`. On `self test/passed` the transition with `DONE` is written before the one on the press and fires first, so the lamp goes on to `on` in the next step. The next press after `off` starts the fragment again in `self test/lit`. The runtime looks for transitions from the innermost state outwards, so a transition of a state of the fragment wins over one of the container state when both could fire.

3.1.14.5 Final states and DONE

A final state has no actions and no transitions of its own. The stack rests on it, and the runtime still tries the transitions of the states around it, as for every state. Without a transition that leads out of the container state, the stack stays on the final state.

`DONE` on a `TRANSITION` or `SPLIT` holds exactly while the state right inside the container state the transition leaves is a final state of its fragment. With other clauses in the same condition all of them must hold. An event that such a condition waits for is treated as for every condition whose other clauses do not hold: when nothing fires in the step, the event is consumed, as 3.1.7.3 Consumption describes. With `CONDITION { EVENT("switch", "abort") DONE }`, an `abort` that arrives before the fragment rests on a final state is lost, and only an `abort` after that leads out. On a `JOIN` the event waits until `DONE` holds. `DONE` holds again whenever the fragment rests on a final state, also after the container state was left and entered again and its fragment ran to its end once more.

Leaving through a final state takes one step more than a transition that leads straight out. The transition into the final state ends its step, and `DONE` fires in the next tick at the earliest, as ticks 6 and 7 of the example in 3.1.14.3 Entering a container state show.

`DONE` on a `JOIN` holds when every source of the join that is a container state rests on a final state in the branch that stands in it. Sources that are no container states do not count for it. The join still needs one branch per source, as 3.1.8 Splits and joins describes. In the second lamp of 2.3.12.2 Container states and DONE, the split starts one branch in `bulb check/powered` and one in `switch check/waiting`. A branch that reaches a final state rests there until the other one has reached one as well, and then the join fires. With the feature TrueParallel on, the branches run on threads of their own, and the join works the same way.

3.1.14.6 Histories, tick limits and breakpoints

With the feature History on, a container state with a history remembers like any other state: `SHALLOW HISTORY` the state right inside it, `DEEP HISTORY` the innermost active state. When the container state is entered again, the runtime goes to the remembered state at once, in the same step, instead of starting the fragment. This is different from a state with inner states written in its braces, which first rests on itself for one tick, as 3.1.9 Histories describes. A remembered state that has inner states of its own is entered as itself, as by a transition, and a remembered container state is gone down into. Without something to remember, or without the feature History, the fragment starts as usual.

A `TICK LIMIT` of a container state counts the ticks of the container state while any state of its fragment is active. Its delay holds back the transitions that leave the container state, not those inside the fragment, and its timeout leads out of the container state. A timeout that leads into a container state goes down into it.

A `BREAKPOINT` on the entry of a container state halts the run before the container state is entered, and the states of its fragment are entered after it. Breakpoints of the states of a fragment work as on every state; each copy of a fragment has its own states, so a breakpoint of one copy halts only that copy.

3.1.14.7 The trace log

The lines of the trace log name the states with their qualified names. In the example of 3.1.14.3 Entering a container state, tick 4 writes the line `TRANSITION_FIRED: self test/lit -> self test/dark` with `from=self test/lit, to=self test/dark` and `stack=self test, self test/dark`. An instance that starts in a container state writes `Initial state stack: self test > self test/lit`, and `setActiveState` with `self test` writes `Forced state: self test > self test/lit`. The lines are described in 3.3.5 Trace log.

3.1.14.8 Compound states and container states

A compound state holds inner states written in its braces. A container state holds the copy of a fragment. What each can be written with, their names and their macros are compared in 2.3.13 Compound states and container states. At run time they differ in these points:

Point	Compound state	Container state
A transition into the state itself	Enters the state alone and rests on it. Its inner states are not entered	Enters the state and the start of its fragment in the same step. The stack never ends on the container state
A transition into an inner state	Enters the compound state too, when it was not active	Not possible: no file names a state of a fragment it uses
ENTRY and DO	The compound state has its own blocks; its DO runs while it is the innermost state	The container state has none
Names in the trace log and in `stateChange`	The names as written: `self test`, `lit`	Qualified: `self test`, `self test/lit`
Re-entry with a history	Rests on itself for one tick, then leads on to the remembered state	Goes to the remembered state at once
End of the inner behaviour	A transition of an inner state leads out	The fragment rests on a final state, and `DONE` on a transition of the container state leads out
A transition of the outer state	Leaves every inner state	Leaves every state of the fragment

The same self test written both ways. With a compound state, the transitions lead into its inner states:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "self test" {
            STATE "lit" {
                ENTRY { SET("bulb->lit", "true") }
            }
            STATE "dark" {
                ENTRY { SET("bulb->lit", "false") }
            }
            STATE "passed" { }
        }
        STATE "on" {
            ENTRY { SET("mode", "on") }
        }
```

```
    }
    TRANSITIONS {
        TRANSITION "off" -> "lit" { }
        TRANSITION "lit" -> "dark" { }
        TRANSITION "dark" -> "passed" { }
        TRANSITION "passed" -> "on" { }
    }
}
```

With a container state, the fragment holds the steps and `DONE` leads out:

```
STATEMACHINE "bulb test" FRAGMENT {
    STATES {
        INITIAL STATE "lit" {
            ENTRY { SET("bulb->lit", "true") }
        }
        STATE "dark" {
            ENTRY { SET("bulb->lit", "false") }
        }
        FINAL STATE "passed" { }
    }
    TRANSITIONS {
        TRANSITION "lit" -> "dark" { }
        TRANSITION "dark" -> "passed" { }
    }
}
```

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        CONTAINER STATE "self test" USE "bulb test" { }
        STATE "on" {
            ENTRY { SET("mode", "on") }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "self test" { }
        TRANSITION "self test" -> "on" {
            CONDITION { DONE }
        }
    }
}
```

Both run in the same ticks; only the names differ. An ENTRY block with actions is followed at once by the search for a transition, so `lit` and `dark` are left in the tick they are entered:

Tick	Compound state	Container state
1	`"off" -> "lit"` fires. Stack `self test`, `lit`	`"off" -> "self test"` fires. Stack `self test`, `self test/lit`
2	ENTRY of `lit` runs, then `"lit" -> "dark"` fires	ENTRY of `self test/lit` runs, then `"self test/lit" -> "self test/dark"` fires

Tick	Compound state	Container state
3	ENTRY of `dark` runs, then `"dark" -> "passed"` fires. Stack `self test`, `passed`	ENTRY of `self test/dark` runs, then `"self test/dark" -> "self test/passed"` fires. Stack `self test`, `self test/passed`
4	`"passed" -> "on"` fires. Stack `on`	`DONE` holds and `"self test" -> "on"` fires. Stack `on`
5	ENTRY of `on` runs	ENTRY of `on` runs

With `TRANSITION "off" -> "self test" { }` in the first text, the compound state is entered alone. The stack stays `self test`, no inner state is ever entered, and `bulb->lit` never changes. The container state cannot be entered that way: it always starts its fragment.

3.1.15 Where the engine goes

This section gathers in one place where a stack goes when states are entered, which states are entered and in which tick their ENTRY blocks run. The rules behind it are in 3.1.3 Active states, 3.1.4 Transitions and 3.1.14 Pages.

3.1.15.1 Ways of entering

A step of a state machine ends with the first transition that fires. The states a transition enters run their ENTRY blocks in the next step of the same branch, which is the next tick, from the outside in. An ENTRY block with at least one action is followed by a search for a transition; when one fires, the ENTRY blocks further inside do not run.

Way	When the stack changes	ENTRY of the entered states
Start of the environment	Before tick 1. The stack is the start path of 3.1.3.1 Initial state	Tick 1, every state of the start path
Instance that joins during the run (`behaves after` of a created object, `ATTACH`, `EXCHANGE`)	At the end of the tick that asked for it. The stack is the start path of its state machine	The next tick, its first step
Transition	In the step that fires it	The next tick
Split	In the step that fires it; one stack per target	The next tick, in each branch
Join	At the start of the step, before every other transition; one stack	The next tick
Timeout of a tick limit	In the step, after the ENTRY blocks and before the DO block, which then does not run	The next tick
History of a compound state	In the step after the compound state was entered, right after its ENTRY block; its DO block does not run	The tick after that
History of a container state	In the same step as the transition into the container state	The next tick

Way	When the stack changes	ENTRY of the entered states
`setActiveState`	At the start of the next tick, before any step	Never. The forced states count as entered without running their ENTRY blocks, and the step of that tick goes on with the DO block of the innermost one

The first tick of a newly joined instance and every other step look for transitions as usual, so a transition can follow right after the ENTRY blocks.

3.1.15.2 Kinds of target

Target	Stack afterwards	Entered	The branch rests on
A state without inner states	The path from the top level to the state	The states of the path that were not active	The state
A compound state	The path to the compound state, nothing below it	The states of the path that were not active	The compound state; its DO block runs from the next tick. With a history that remembers a state, it leads on one tick later
An inner state of a compound state	The path to the inner state	The compound state too, when it was not active	The inner state
A container state	The path to the container state and the start path of its fragment, or the remembered state of its history	The container state, when it was not active, and every state below it	The innermost state of the fragment, never the container state alone
A state of a fragment, by a transition of the fragment	The path to the state, inside the container state	The states of the path that were not active; the container state stays active and is not entered again	The state
A final state	The path to the final state	The final state	The final state, until a transition of a state around it fires

A transition from a state to itself, or to one of the states around it, enters the target again together with the state around it, as 3.1.4.3 Where a transition leads describes. For a container state this means: `"self test" -> "self test"` leaves the fragment, enters `self test` again and starts its fragment anew.

3.1.15.3 The start and the INITIAL mark

The start follows the `INITIAL` mark through every compound state, wherever the mark stands, by the rule of 2.3.2.2 The initial state. In the lamp

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        STATE "off" { }
        STATE "on" {
            STATE "full" { }
            INITIAL STATE "dimmed" { }
        }
    }
    TRANSITIONS { }
}
```

the stack at the start is `on`, `dimmed`, although `off` is the first state of the top level. Both states are entered in tick 1.

The mark of a fragment is not seen from the state machine that uses it. A container state never counts as holding a mark, so the start does not choose it for the mark inside its fragment. In this lamp the fragment `levels` marks its state `dimmed`, and the start is `spare`, `ready`:

```
STATEMACHINE "levels" FRAGMENT {
    STATES {
        STATE "full" { }
        INITIAL STATE "dimmed" { }
        FINAL STATE "out" { }
    }
    TRANSITIONS { }
}
```

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        STATE "off" { }
        CONTAINER STATE "on" USE "levels" { }
        STATE "spare" {
            INITIAL STATE "ready" { }
        }
    }
    TRANSITIONS { }
}
```

A state machine must mark one of its own states, so `on` is the start only when it is marked itself, `INITIAL CONTAINER STATE "on" USE "levels"`. Then the start is `on`, `on/dimmed`: inside the container state the mark of its fragment counts. Without any mark of its own the text is refused at the upload with `kstates.initial-state-count`.

3.1.15.4 Entering and leaving, tick by tick

A compound state entered by a transition. With

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            ENTRY { SET("mode", "on") }
            STATE "full" { }
            STATE "dimmed" { }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "off" -> "dimmed" {
            CONDITION { EVENT("switch", "dim") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

and a press before tick 2, a press before tick 4 and a `dim` before tick 6:

Tick	Event	lamp
2	press	`"off" -> "on"` fires. Stack `on`
3		ENTRY of `on` runs. Stack `on`; `full` and `dimmed` are not entered
4	press	`"on" -> "off"` fires. Stack `off`
6	dim	`"off" -> "dimmed"` fires. Stack `on`, `dimmed`
7		ENTRY of `on` runs, then the stack rests on `dimmed`

A compound state with a history. With `SHALLOW HISTORY STATE "on"` in the text above, the feature History on, the stack `on`, `dimmed` left by a press, and a press again:

Tick	Event	lamp
k	press	`"off" -> "on"` fires. Stack `on`
k+1		ENTRY of `on` runs, then the history leads on. Stack `on`, `dimmed`
k+2		ENTRY of `dimmed` runs

A container state with a history goes there in tick k already. With `SHALLOW HISTORY CONTAINER STATE "self test" USE "bulb test"` in the lamp of 3.1.14.3 Entering a container state, left by a press in `self test/dark` and entered again by a press:

Tick	Event	lamp
k	press	`"off" -> "self test"` fires. Stack `self test`, `self test/dark`
k+1		ENTRY of `self test/dark` runs

Without the feature History, or while the history remembers nothing yet, the stack in tick k is `self test`, `self test/lit`.

Leaving. A transition from a state leaves every state inside it, whatever is active there, and no block runs for the states left. The stack becomes the path to the target, entered as the tables above describe. A transition of a state inside wins over one of the state around it when both could fire, because the runtime looks from the innermost state outwards.

3.2 Runtime features

A feature is a part of the runtime that a client switches on for one environment. Two features give language constructs their effect at run time: History for `SHALLOW HISTORY` and `DEEP HISTORY`, TickLimit for `TICK LIMIT` blocks. TrueParallel runs the branches of a split on threads of their own. The other features let a client step through a run, halt it at breakpoints, and record and replay the events of a run. All features are off in a new environment.

Feature	What it does	Needs
`History`	Gives the history marks of states their effect	
`TickLimit`	Gives `TICK LIMIT` blocks their effect	
`TickByTick`	Pauses the run and advances it tick by tick	
`Breakpoint`	Halts the run at the entry and exit of states and at the start and end of transitions	`TickByTick`
`EventRecording`	Records the events that clients send, by tick	
`EventInjection`	Hands a list of events to the state machines at given ticks	
`EventReplayController`	Turns a recording into the list of EventInjection	`EventRecording`, `EventInjection`
`TrueParallel`	Lets the branches of a split do their steps at the same time, each on a thread of its own	

3.2.1 Switching features

The routes for features are in 3.4.10 Features. A client switches a feature on with `activateFeature` and off with `deactivateFeature`, in any phase of the environment. A switch takes effect at once, also in the middle of a run.

Switching on a feature also switches on the features it needs. Switching off a feature also switches off the features that need it, and the features that were only switched on because it needed them. Switching off `TickByTick` switches off `Breakpoint`.

A client talks to a feature with `sendMessageToFeature`. The body of the request is the message as text, and the body of the answer is the answer of the feature. History, TickLimit and TrueParallel take no messages; they answer every message with an empty body. The messages of every other feature are listed below.

```
curl -X POST "http://127.0.0.1:8080/activateFeature?envKey=env-1763035112345&featureName=
    ↪ TickByTick"
curl -X POST "http://127.0.0.1:8080/sendMessageToFeature?envKey=env-1763035112345&
    ↪ featureName=TickByTick" -d "status"
```

```
running
```

3.2.2 History

With History on, a state marked `SHALLOW HISTORY` or `DEEP HISTORY` leads on into the inner state it was in when it is entered again. The rules and an example are in 3.1.9 Histories, and for a container state in 3.1.14.6 Histories, tick limits and breakpoints. What a state remembers stays when the environment is stopped and started again, and is forgotten when History is switched off.

3.2.3 TickLimit

With TickLimit on, `DELAY` holds a state for its first ticks and `TIMEOUT` leaves a state after a number of ticks. The rules and an example are in 3.1.10 Tick limits. The counts stay when the environment is stopped and started again.

3.2.4 TickByTick

TickByTick pauses the run between two ticks and lets it advance by a number of ticks. While the run is paused, the environment still takes events from clients and answers snapshots of its model.

Message	Answer	What happens
`pause`	`execution paused`	The run waits before its next tick
`resume`	`execution resumed`	The run goes on
`tick:N`	`execution is advancing N tick(s)`	The paused run does N more ticks and waits again. The answer says `tick` for 1 and `ticks` for more
`status`	`paused` or `running`	

`tick:N` works only while the run is paused and no earlier `tick:N` is still under way. Otherwise the answer is `execution is currently not paused, ticks will not be applied`. With N less than 1 the answer is `amount of ticks must not be less than one`. Any other message is refused with status 400 and the text `Invalid message: <message>`.

The first start of an environment resumes a paused run. To step from the first ticks on, send `pause` after the start. Switching TickByTick off also resumes the run. A stop keeps the pause: after the start the run still waits, and a `resume` or `tick:N` sent while the environment is stopped takes effect only after the start.

While TickByTick is on, every client connected to the environment over the WebSocket gets the message `engineStatus` each time the run starts to wait and each time it goes on, as 3.5 WebSocket describes.

```
curl -X POST "http://127.0.0.1:8080/sendMessageToFeature?envKey=env-1763035112345&
    ↪ featureName=TickByTick" -d "pause"
curl -X POST "http://127.0.0.1:8080/sendMessageToFeature?envKey=env-1763035112345&
    ↪ featureName=TickByTick" -d "tick:3"
```

```
execution paused
execution is advancing 3 ticks
```

3.2.5 Breakpoint

Breakpoint halts the run at a place inside a tick. The run then waits as if paused with TickByTick. `resume` lets the run go on, and `tick:1` lets it go on to the next breakpoint or to the start of the next tick, whichever comes first. Both are messages to TickByTick.

3.2.5.1 Places

A run halts at these places:

Place	When
Entry of a state	Before the ENTRY block of the state runs, also when the block is empty
Exit of a state	When a transition leaves the state, before the transition starts. The states left are taken from the innermost outwards. A transition from a state to itself leaves no state
Start of a transition	Before the transition changes the active states
End of a transition	After the transition changed the active states

A transition that a `DELAY` holds back halts nowhere. The timeout of a `TICK LIMIT` and the step into the state that a history remembers count as transitions. They have no text, so only an override halts at their start or end.

3.2.5.2 Breakpoints in the text and overrides

A state machine marks breakpoints in its text with `BREAKPOINT { ENTRY EXIT }` in a state and `BREAKPOINT { START END }` in a transition, as 2.3 KStates shows. An override switches a breakpoint on or off at run time, whatever the text says. Overrides belong to the environment and stay when it starts again. The overrides of an object are dropped when its state machine instance leaves, as 3.1.13 State machine management and lifecycle describes, so an instance that later runs on the object starts with the breakpoints of its text.

All messages to Breakpoint are JSON objects with the entry `action`:

Action	Further entries	Answer
`addStateOverride`	`modelElementId`, `stateIdentifier`, `timing` (`entry` or `exit`), `breakpointStatus` (`true` or `false`)	`Breakpoint override has added a breakpoint on state <state> entry in statemachine attached to <object>`, with `removed` for `false` and `exit` for the exit

Action	Further entries	Answer
`removeStateOverride`	`modelElementId`, `stateIdentifier`, `timing`	`Breakpoint override on state <state> entry has been removed in statemachine attached to <object>`
`addTransitionOverride`	`modelElementId`, `transitionIdentifier`, `timing` (`start` or `end`), `breakpointStatus`	`Breakpoint override has added a breakpoint on transition <transition> start in statemachine attached to <object>`
`removeTransitionOverride`	`modelElementId`, `transitionIdentifier`, `timing`	`Breakpoint override on transition <transition> start has been removed in statemachine attached to <object>`
`getOverrides`		The overrides in four groups, as shown below
`getStatus`		The status of the run as JSON, as shown below

`modelElementId` is the key of the object the state machine instance runs on. `stateIdentifier` is the name of the state. A wrong timing gives an answer that starts with `Breakpoint override could not be added` or `Breakpoint override could not be removed`, an unknown action the answer `Invalid action: <action>`, and a message that is no JSON or lacks an entry an answer that starts with `Error when parsing the message:`. All these answers have status 200.

`transitionIdentifier` names the transition as the runtime writes it. For `TRANSITION "off" -> "full"` with an `EVENT` clause it is:

```
Transition(fromState=off, toState=full, allowLoops=true, startBreakpoint=false,
    ↪ endBreakpoint=false, condition=EVENT)
```

`allowLoops` is `false` for a transition marked `NOT LOOPING`. `startBreakpoint` and `endBreakpoint` say whether the text marks the breakpoints. `condition` is `VALUE` for a transition without `CONDITION`, `EVENT`, `EVAL` or `VALUE` for a condition of one clause, and `COMPOSITE` for a condition of more clauses. A split starts with `SplitTransition(fromState=off, toStates=[test bulb, test switch], ...` and a join with `JoinTransition(fromStates=[bulb ok, switch ok], toState=on, ...`. The entry `halt` of the status below names the transition in the same form. Transitions with the same identifier share their overrides.

```
{
  "action": "addStateOverride",
  "modelElementId": "lamp",
  "stateIdentifier": "on",
  "timing": "entry",
  "breakpointStatus": true
}
```

```
Breakpoint override has added a breakpoint on state on entry in statemachine attached to
    ↪ lamp
```

The answer of `getOverrides`:

```
{Entry Breakpoints={lamp={on=true}}, Exit Breakpoints={}, Start Breakpoints={}, End
    ↪ Breakpoints={}}
```

3.2.5.3 Status

`getStatus` answers a JSON object. `paused` is true once a pause was asked for and until the run is resumed. `waiting` is true while the run waits. `halt` is there only while the run waits at a breakpoint and names the place. The same object is the payload of the WebSocket message `engineStatus`.

For a state:

```
{"paused":true,"waiting":true,"halt":{"modelElementId":"lamp","timing":"entry","state":"on
    ↪ "}}
```

For a transition, `kind` is `TRANSITION`, `SPLIT` or `JOIN`, `from` and `to` are the states, and `index` is the place of the transition in the `TRANSITIONS` block of its file, counted from 0. A timeout and the step into a remembered state have the index -1:

```
{"paused":true,"waiting":true,"halt":{"modelElementId":"lamp","timing":"start","transition
    ↪ ":"Transition(fromState=off, toState=full, allowLoops=true, startBreakpoint=false,
    ↪ endBreakpoint=false, condition=EVENT)","kind":"TRANSITION","from":["off"],"to":["
    ↪ full"],"index":0}}
```

The order of the entries in these objects is not fixed.

3.2.6 EventRecording

EventRecording records the events that clients send over the WebSocket, under the number of the tick in which they reach the state machines. While it records, an event from a client waits for the start of the next tick. Events that the state machines raise and injected events are not recorded.

Message	Answer	What happens
`start`	`recording started`	Recording starts
`stop`	`recording stopped`	Recording stops. Events that wait for the next tick go to the state machines at once
`get`	The recording as JSON	
`clear`	`cleared recorded events`	The recording is emptied

Any other message gets the answer `invalid message: <message>`. A recording stays when the environment is stopped and started again, and the ticks go on with the next number, so the recording goes on where it stopped.

The recording is a JSON object. Its keys are tick numbers, and each value is a list of events, each written as a string with a JSON object:

```
{"5": ["{\"environmentKey\":\"switch\",\"messageType\":\"dim\",\"payload\":\"{\\\"bright
    ↪ \\\": false}\",\"payloadFormat\":\"JSON\"}"]}
```

`environmentKey` is the domain of the event, `messageType` its name, `payload` its payload as text and `payloadFormat` one of `NONE`, `JSON` and `KMODEL`. The order of the entries is not fixed.

3.2.7 EventInjection

EventInjection holds a list of events by tick and hands each event to the state machines at the start of its tick.

Message	Answer	What happens
`upload:<list>`	`successfully uploaded events`	Replaces the list
`start`	`injection started`	Injection starts
`stop`	`injection stopped`	Injection stops
`get`	The list as JSON	
`clear`	`cleared event queue`	The list is emptied

Any other message gets the answer `invalid message: <message>`. The list has the form of a recording. Each event can be a string with a JSON object, as in a recording, or a JSON object. Each event needs `environmentKey`, `messageType` and `payload`; `payloadFormat` is optional. A list that cannot be read is refused with an error status and the reason as text.

While injection runs, the runtime hands over at the start of each tick every event of the list whose tick is not later than the current tick, and removes it from the list. When injection starts in tick 10, the events of ticks 1 to 10 come at once.

The time to live of an injected event counts from the upload of the list. An event whose tick comes later than its time to live after the upload has expired when it is handed over, and the state machines never see it. For long lists, set the time to live of the environment to 0. Injected events carry their payload as text only, so paths into `event` lead nowhere for them.

```
curl -X POST "http://127.0.0.1:8080/sendMessageToFeature?envKey=env-1763035112345&
    ↪ featureName=EventInjection" \
  -d 'upload:{"3": [{"environmentKey": "switch", "messageType": "press", "payload": ""}]}'
curl -X POST "http://127.0.0.1:8080/sendMessageToFeature?envKey=env-1763035112345&
    ↪ featureName=EventInjection" -d "start"
```

```
successfully uploaded events
injection started
```

3.2.8 EventReplayController

EventReplayController replays a recording. It needs EventRecording and EventInjection and switches them on.

Message	Answer	What happens
`initialize`	`replay successfully initialized`	Stops recording and injection and uploads the recording as the list of EventInjection

Any other message gets the answer `invalid message: <message>`. When the upload fails, the answer starts with `replay could not be initialized, following error occurred:`.

A start after a stop continues the run with the next tick, as 3.1.13.9 Stop and start, forcing and clients describes. In the same environment the ticks of a recording have passed by then, so after `initialize` and a start all recorded events come in the next tick. A replay in the same ticks as before runs in a new environment:

1. Switch on EventRecording and send it `start`.
2. Start the environment and send events from a client.
3. Send `get` to EventRecording and keep the answer.
4. Create a new environment with the same uploads, switch on EventInjection, send it `upload:` with the recording and `start`.
5. Run and start the new environment. The events come in the same ticks as before, on the model as it was uploaded.

With TrueParallel off, the trace log of the replay is the trace log of the recorded run line by line, apart from the times and the key of the environment, also when state machines are attached, exchanged and detached and when the state machines split and join, since the path numbers count the same way.

3.2.9 TrueParallel

Without TrueParallel, the branches of a split do their steps one after the other on the one thread of the environment, in the order of the targets of the split, as 3.1.8 Splits and joins describes. Every run of the same model with the same events takes the same course.

With TrueParallel on, each branch gets a thread of its own when the split has fired, and keeps it for as long as the branch lives. In every tick all branches of a state machine do their steps at the same time on the shared model, and the tick ends when every branch has done its step. Ticks keep their numbers, so TickLimit, TickByTick, Breakpoint, EventRecording and EventInjection count them as before, and the active states are reported once per tick after all steps. A join is still tried before the branches step. A branch that waits for Python in an EVAL holds up only itself until the end of the tick, and each branch that calls Python at the same time as another one gets a Python process of its own.

A switch of TrueParallel takes effect at the start of the next tick. The threads of branches end when a join merges the branches, when the environment stops and at the end of the tick in which TrueParallel was switched off. A start after a stop gives the branches new threads at their next step. A stop while a branch waits at a breakpoint keeps the threads of the halted tick until it ends. State machines without a split run as without the feature, and so do the several state machines of an environment, which still do their steps one after the other.

3.2.9.1 What a modeler can rely on

- Each single action is atomic: one `SET`, `APPEND`, `SETLIST`, `DROPLIST`, `SETOBJ`, `APPENDOBJ`, `DROPOBJ` or `EVENT` changes the model or the bus in one go, and no other branch sees it half done.
- An action that computes its value first reads and then writes. Between the two, another branch can write. Two branches that increment the same number with `SET("count", EVAL { return $(count) + 1 })` can lose an increment.
- Two branches that write the same property race. Which value stays is not defined.
- The order of actions across branches is not defined. Within one branch the actions run in their order, as always.
- The changes of one ENTRY or DO block still reach the clients together after the block, as 3.1.5.1 Blocks and errors describes. Blocks of different branches reach them in the order in which they ended. The revisions of the pushes still increase one by one.
- Events that one branch raises can be seen by another branch in the same tick or only in the next one.
- Branches that create, remove or exchange objects with state machines change the instances of the environment only at the end of the tick, all together, as 3.1.13.3 When instances join and leave describes. An instance that a branch ends stops its other branches after the block they are in.
- With Breakpoint on, branches that reach breakpoints in the same tick halt one after the other. `resume` lets the waiting branch go on, and the next one halts. While one branch waits, the others finish their steps.

For a deterministic run, as in tests and in replays with EventInjection, leave TrueParallel off.

The trace log names the path of the branch in every line of a branch, as 3.3.5 Trace log describes. With TrueParallel on, the lines of branches that step at the same time interleave in the order in which they were written, and the path tells them apart. The path numbers are the same as with TrueParallel off, because splits, joins and exchanges are carried out by the thread of the environment in the same order either way.

```
curl -X POST "http://127.0.0.1:8080/activateFeature?envKey=env-1763035112345&featureName=
    ↪ TrueParallel"
```

3.3 Running the runtime

The runtime is one program that listens for HTTP requests and WebSocket connections on one port, by default `http://127.0.0.1:8080` and `ws://127.0.0.1:8080/ws`. It runs on your machine with Java or in a Docker container. The IDE builds and starts the container itself, as 4.5 Running describes.

Way to run	Needs

3.3.1 Requirements

Way to run	Needs
On your machine	Java 21. Python 3 on the path as `python3`, `python` or `py` for state machines with Python code, with the packages that this code imports
In Docker	Docker. The image brings Java 21 and Python 3 with the packages `numpy`, `scipy`, `sympy`, `pandas`, `regex`, `python-dateutil` and `chardet`

State machines whose `EVAL` blocks and macros the runtime computes itself need no Python, as 3.1.6 EVAL and macros describes.

The runtime is built with the KStates, KMeta and KModel readers of karpfen-dsl-tools, which is a git submodule in the folder `karpfen-dsl-tools`. Clone the repository with its submodule:

```
git clone --recurse-submodules https://github.com/karpfenproject/karpfen-runtime.git
cd karpfen-runtime
```

3.3.2 Running on your machine

The script `run_local.sh` builds the runtime and starts it from the folder of the repository:

```
./run_local.sh
```

The build runs without tests and ends in `build/install/karpfen-runtime/bin/karpfen-runtime`. The runtime reads `application.conf` from the folder it starts in. To start it without the script, from the folder of the repository:

```
./gradlew installDist -x test
./build/install/karpfen-runtime/bin/karpfen-runtime
```

`./gradlew run` builds and starts the runtime in one step, also with `application.conf` from the folder of the repository.

At the start the runtime prints the configuration it uses:

```
[Main] Configuration loaded:
[Main]   Server: 127.0.0.1:8080
[Main]   WebSocket enabled: true
[Main]   Log level: INFO
[Main]   Engine tracing: false
[Main]   Default tick delay: 50ms
[Main]   Default event TTL: 3000ms
[Main]   Event consumption on fire: true
```

3.3.3 Running in Docker

The script `run_docker.sh` builds the image `karpfen-runtime` from the `Dockerfile` of the repository and runs it:

```
./run_docker.sh
./run_docker.sh /var/log/karpfen
```

The script hands the container a copy of `application.conf` in which `server.host` is `0.0.0.0`, so the runtime can be reached from outside the container. It maps port 8080 of the container to port 8080 of your machine. When you change `server.port`, change the port in the script too. The folder in the first argument, by default `logs` in the folder of the repository, is mounted as `/app/logs` in the container. For trace logs on your machine, set `engineTracing.tracingLogDirectory` to `/app/logs`. The container is removed when it ends.

The image has no `application.conf` of its own. A container started without one runs with the defaults of the table below, and with `server.host` at `127.0.0.1` it cannot be reached from outside the container. To run the image without the script, mount a configuration file at `/app/application.conf`:

```
docker build -t karpfen-runtime .
docker run --rm -p 8080:8080 -v "$PWD/docker.conf:/app/application.conf:ro" karpfen-runtime
```

3.3.4 Configuration

The runtime reads `application.conf` from the folder it starts in. Without the file it uses the defaults. The file has one option per line in the form `name = value`, grouped in blocks:

```
server {
  host = "127.0.0.1"
  port = 8080
}
engine {
  defaultTickDelayMs = 50
  defaultEventTtlMs  = 3000
}
```

The runtime finds an option by its name inside the block and does not look at the name of the block. A line that starts with `#` is a comment, and so is text after `#` when two blanks or a tab stand before it. A value may stand in quotes. A number that cannot be read gives the default. A boolean is `true` for `true` in any case and `false` for anything else.

Option	Default	In the shipped file	Meaning
`server.host`	`127.0.0.1`	`127.0.0.1`	The address the runtime listens on. `0.0.0.0` listens on all addresses
`server.port`	`8080`	`8080`	The port for HTTP and the WebSocket
`websocket.enabled`	`true`	`true`	With `false`, the runtime pushes no messages to WebSocket clients
`websocket.queueTimeoutMs`	`1000`	`1000`	Read and not used
`logging.level`	`INFO`	`INFO`	Printed at the start and not used
`logging.consoleOutput`	`true`	`false`	Read and not used

Option	Default	In the shipped file	Meaning
`engine.defaultTickDelayMs`	`1000`	`50`	The tick delay of an environment without `setTickDelay`, in milliseconds
`engine.defaultEventTtlMs`	`1000`	`3000`	The time to live of events in an environment without `setEventTtl`, in milliseconds. 0 means events never expire
`engine.eventConsumptionOnFire`	`true`	`true`	With `false`, an event that fires a transition is not consumed by it, as 3.1.7.3 Consumption describes
`engineTracing.tracingEnabled`	`false`	`false`	With `true`, the runtime writes a trace log per environment
`engineTracing.tracingLogDirectory`	none	`logs`	The folder of the trace logs, relative to the folder the runtime starts in or absolute. Without it no trace file is written
`engineTracing.tracingConsoleOutput`	`false`	`false`	With `true` and tracing on, the runtime also prints every trace line
`engineTracing.simpleTrace`	`true`	`true`	Read and not used

A tick delay of 0 lets an environment tick without any pause and keeps one processor core busy.

3.3.5 Trace log

With `engineTracing.tracingEnabled` set to `true`, the runtime writes one trace log per environment to `<tracingLogDirectory>/engine-trace-<envKey>.log`. The file is opened when the environment is run and closed when the runtime ends. A stop and a start of the environment go on writing to it.

Every line has the time, the key of the environment, the key of the object of the state machine or `*` for the whole environment, the type of the entry, a message and details:

```
[HH:mm:ss.SSS] [<envKey>] [<object>] <TYPE>: <message> | <name>=<value>, <name>=<value>
```

The beginning of the trace of the lamp:

```
# Engine Trace Log -- Engine: env-1763035112345 -- Started: 2026-11-13T09:12:01.120Z
[10:12:01.124] [env-1763035112345] [lamp] STATE_MACHINE_ATTACHED: State machine of the run
    ↪ configuration attached | stateMachine=, path=0
[10:12:01.125] [env-1763035112345] [lamp] INITIAL_STATE: Initial state stack: off | path=0,
    ↪  stack=off
[10:12:01.126] [env-1763035112345] [wallSwitch] STATE_MACHINE_ATTACHED: State machine of
    ↪ the run configuration attached | stateMachine=, path=0
[10:12:01.126] [env-1763035112345] [wallSwitch] INITIAL_STATE: Initial state stack: down |
    ↪ path=0, stack=down
[10:12:01.127] [env-1763035112345] [*] ENGINE_START: Engine started with 2 state machine(s),
    ↪  tickDelay=500ms
[10:12:01.128] [env-1763035112345] [lamp] TICK_START: Tick #1 | path=0, stack=off,
    ↪ currentStates=off, parallel=false
```

```
[10:12:01.129] [env-1763035112345] [lamp] STATE_ENTRY_EXEC: Executing onEntry for state '
    ↪ off' | path=0, state=off, stack=off
[10:12:01.131] [env-1763035112345] [lamp] STATE_DO_SKIP: onDo for state 'off' is empty --
    ↪ skipped | path=0, state=off, stack=off
[10:12:01.131] [env-1763035112345] [lamp] TICK_END: Tick #1 completed | path=0, stack=off,
    ↪ currentStates=off, parallel=false
```

The types of entries:

Type	Written when	Message
`INITIAL_STATE`	A state machine gets its start stack, right after it is attached or exchanged, or a client forces a state	`Initial state stack:  on > full` or `Forced state:  on > full`
`ENGINE_START`	The environment starts, or continues after a stop	`Engine started with 2 state machine(s), tickDelay=500ms` or `Engine continued after tick 42 with 2 state machine(s)`
`ENGINE_STOP`	The environment stops	`Engine stopped after 42 ticks`
`STATE_MACHINE_ ATTACHED`	A state machine instance joins on an object without one	`State machine 'glowing' attached`, or `State machine of the run configuration attached` for an attached state machine
`STATE_MACHINE_ DETACHED`	A state machine instance leaves and no other takes its place	`State machine 'glowing' detached`
`STATE_MACHINE_ EXCHANGED`	`EXCHANGE` ended the instance on an object and started another one in its place	`State machine 'glowing' exchanged for 'blinking'`, or `State machine of the run configuration exchanged for 'blinking'`
`TICK_START`	A state machine begins its step	`Tick #3`
`TICK_END`	A state machine ends its step	`Tick #3 completed`
`STATE_ENTRY_ EXEC`	An ENTRY block runs	`Executing onEntry for state 'on'`
`STATE_ENTRY_ SKIP`	A state with an empty ENTRY block is entered	`onEntry for state 'on' is empty - skipped`
`STATE_DO_EXEC`	A DO block runs	`Executing onDo for state 'on'`
`STATE_DO_SKIP`	The innermost state has an empty DO block	`onDo for state 'on' is empty - skipped`
`TRANSITION_ START`	A transition begins	`off -> full`
`TRANSITION_ FIRED`	A transition changed the active states	`off -> full`
`TRANSITION_ SKIPPED_LOOP`	A `NOT LOOPING` transition was skipped	`Skipped NOT LOOPING transition`
`SPLIT_START`, `SPLIT_FIRED`	A split begins and is done	`off -> test bulb,test switch`
`JOIN_START`, `JOIN_FIRED`	A join begins and is done	`bulb ok,switch ok -> on`
`ACTION_ERROR`	An action failed	`Error in onEntry for state 'off':  <error>`

Every entry of a state machine names a path. A path is one stretch of the state machine on an object that a single branch runs from its beginning to its end. The numbers count per object from 0 and are never given twice while the environment lives, so the object and the path number together name one stretch, and grouping the lines by object and path separates everything, also the lines of branches that interleave while the feature TrueParallel is on. The first state machine on an object runs on path 0. Each target of a split starts a new path, a join ends the paths of the split and starts one more, a forced state ends the current paths and starts one more, and a state machine that `EXCHANGE` or a later attachment brings to the same object starts with the next number. A stop and a start keep the paths.

The details, in the order in which they appear:

Type	**Details**
`STATE_MACHINE_ATTACHED`	`stateMachine` (the name, empty for a state machine of the run configuration), `path`
`STATE_MACHINE_DETACHED`	`stateMachine`, then `path`, or `paths` while the state machine is split
`STATE_MACHINE_EXCHANGED`	`ended` (the name of the state machine that ended, empty for one of the run configuration), `started`, `ends` (the paths that ended), `path` (the path of the new state machine)
`INITIAL_STATE`	`path`, `stack`; after a forced state first `ends`, the paths the force ended
`TICK_START`, `TICK_END`	`path`, or `paths` while the state machine is split; `stack` (the active states of all branches), `currentStates` (the innermost state of each branch, in the order of `paths`), `parallel`
entries of states	`path`, `state`, `stack` (the stack of the branch)
`TRANSITION_START`, `TRANSITION_FIRED`	`path`, `from`, `to`, `stack`
`TRANSITION_SKIPPED_LOOP`	`path`, `stack`
`SPLIT_START`	`path`, `from`, `to`
`SPLIT_FIRED`	`path` (the path that splits), `starts` (the new paths, in the order of the targets), `from`, `to`
`JOIN_START`	`paths` (the paths about to be joined), `from`, `to`
`JOIN_FIRED`	`ends` (the joined paths), `path` (the new path), `from`, `to`
`ACTION_ERROR`	`path`, `state`, `stack`

A detail that lists several paths separates them with commas. The entries of the whole environment, under `*`, name no path. A line about a state machine as it stands names `path`, or `paths` while it is split; a line that changes the paths names those it ends in `ends` and those it starts in `starts` or `path`.

Some lines of a lamp that tests its bulb and its switch in two branches and joins them in `on`, shortened to object, type, message and details:

```
[lamp] TRANSITION_FIRED: idle -> off | path=0, from=idle, to=off, stack=off
[lamp] SPLIT_FIRED: off -> test bulb,test switch | path=0, starts=1,2, from=off, to=test
    ↪ bulb,test switch
[lamp] TICK_END: Tick #2 completed | paths=1,2, stack=test bulb,test switch, currentStates=
    ↪ test bulb,test switch, parallel=true
[lamp] STATE_ENTRY_EXEC: Executing onEntry for state 'test bulb' | path=1, state=test bulb,
```

```
    ↪  stack=test bulb
[lamp] TRANSITION_FIRED: test switch -> check position | path=2, from=test switch, to=check
    ↪  position, stack=check position
[lamp] STATE_ENTRY_EXEC: Executing onEntry for state 'bulb ok' | path=1, state=bulb ok,
    ↪ stack=bulb ok
[lamp] JOIN_FIRED: bulb ok,switch ok -> on | ends=1,2, path=3, from=bulb ok,switch ok, to=
    ↪ on
[lamp] STATE_ENTRY_EXEC: Executing onEntry for state 'on' | path=3, state=on, stack=on
```

An exchange of the state machine of the bulb, which ran on path 0, for `blinking` writes one line, followed by the start stack of the new state machine:

```
[bulb] STATE_MACHINE_EXCHANGED: State machine 'glowing' exchanged for 'blinking' | ended=
    ↪ glowing, started=blinking, ends=0, path=1
[bulb] INITIAL_STATE: Initial state stack: off | path=1, stack=off
```

Two exchanges of one object in the same tick write one line that names the state machine that ran before the tick and the one named last. `EXCHANGE` on an object without a state machine writes `STATE_MACHINE_ATTACHED`. A client that observes the state machines still gets `stateMachineDetached` and then `stateMachineAttached` for an exchange (see 3.5.3.6).

3.3.6 Stopping the runtime

Ctrl+C in the terminal of the runtime, or the end of the Docker container, stops it. The runtime closes the WebSocket connections of all environments with code 1001 and the reason `Runtime stopped`, ends their engines with their Python processes and trace logs, closes its server and ends. It keeps nothing: all environments, models and recordings are gone.

A single environment ends for good with `POST /dropEnvironment` of 3.4.2 Environments while the runtime goes on. Its connections close with code 1000 and the reason `Environment dropped`, and its trace file stays in the trace directory. A runtime that runs for a long time and serves many runs should have its clients drop the environments they no longer need, since a stopped environment keeps its model, its engine and its Python processes in memory until it is dropped.

The runtime prints what it does on its console, also when tracing is off: environments that start and stop, clients that register, payloads it cannot read, failed actions and failed `EVAL` clauses.

3.4 HTTP API

A client drives the runtime over HTTP: it creates environments, uploads models, runs them, registers WebSocket clients and talks to features. The examples use the lamp of the IDE example `examples/lamp` and the runtime at `http://127.0.0.1:8080`.

3.4.1 Conventions

Parameters go in the query string. Uploads and messages to features go in the body as plain text. The runtime accepts requests from any origin.

A successful call answers status 200. Its body is empty unless the route below names an answer. A failed call answers one of these statuses with the reason as plain text in the body:

Status	When
400	A parameter is missing or wrong, the environment does not exist, a name is unknown, or an upload has errors
409	The call does not fit the phase of the environment or what it holds
500	Anything else went wrong

A missing parameter gives the text `Missing required parameter: <name>`, an unknown environment `Environment with key <envKey> does not exist`.

Group	Routes
3.4.2 Environments	`POST /createEnvironment`, `POST /dropEnvironment`
3.4.3 Uploads	`PUT /setMetamodel`, `PUT /setModel`, `PUT /setEventDefinitions`, `PUT /setStateMachine`
3.4.5 Settings	`POST /setTickDelay`, `POST /setEventTtl`
3.4.6 Clients	`POST /registerClientForWebSocket`, `POST /registerObjectObserver`, `POST /registerDomainListener`
3.4.7 Running	`POST /runEnvironment`, `POST /startEnvironment`, `POST /stopEnvironment`, `POST /setActiveState`
3.4.8 Observation	`GET /observatory/environments`, `GET /observatory/statemachine`, `GET /observatory/stateMachines`, `POST /observatory/registerClient`, `GET /observatory/model`, `POST /observatory/subscribeModel`, `POST /observatory/unsubscribeModel`
3.4.9 Health	`GET /health`
3.4.10 Features	`GET /getRegisteredFeatures`, `GET /getActiveFeatures`, `POST /activateFeature`, `POST /deactivateFeature`, `POST /sendMessageToFeature`

3.4.2 Environments

`POST /createEnvironment` creates an environment in the phase `configuring` and answers its key as text. The key has the form `env-<milliseconds since 1970>` and is never handed out twice by one runtime.

```
curl -X POST "http://127.0.0.1:8080/createEnvironment"
```

```
env-1763035112345
```

`POST /dropEnvironment` with `envKey` drops the environment for good, in every phase: `configuring`, `activated`, `running`, `halted` and `stopped`. The runtime ends everything the environment owns: the engine with its thread and the threads of branches, a pause of TickByTick or a halt at a breakpoint, the Python sessions, the features with what they recorded, the events that wait, the observers and listeners, the access keys that no client used yet, and the WebSocket connections of its clients, which close with code 1000 and the reason `Environment dropped`. The trace log is closed and stays on disk. Pushes that the runtime has not sent yet when the drop comes are not sent any more, so a client that needs the last model reads the snapshot of 3.4.8 Observation before it drops.

The call answers 200 with an empty body once the engine has ended, normally within the rest of the running tick. A tick that does not end within 5 s, because an `EVAL` waits for Python, has its Python processes ended and its thread interrupted; the call answers after that. A drop that comes while a stop or a start is under way waits for it. From the moment the drop begins, every route answers for the key as for an environment that never existed, the key no longer appears in `observatory/environments`, and the runtime never hands it out again. So of two drops of one environment at the same time, one answers 200 and the other 400 with `Environment with key <envKey> does not exist`, which is also the answer for a key that never existed.

```
curl -X POST "http://127.0.0.1:8080/dropEnvironment?envKey=env-1763035112345"
```

3.4.3 Uploads

The four upload routes take the text of a file as body and the key of the environment as `envKey`. They work only in the phase `configuring`; after `runEnvironment` they answer 409 with `Environment with key <envKey> is already active and cannot be modified in the intended way.` The runtime reads every upload at once. A text with errors is refused, as 3.4.4 Rejected uploads describes, and the environment stays as it was. A byte order mark at the very start of a text is ignored, so such a text is read like the same text without it. An accepted upload replaces the one before it. Warnings of the readers do not refuse an upload; the runtime prints them on its console.

Route	Parameters	Body
`PUT /setMetamodel`	`envKey`	A KMeta text
`PUT /setModel`	`envKey`	A KModel text
`PUT /setEventDefinitions`	`envKey`	A KMeta text with the event definitions
`PUT /setStateMachine`	`envKey`, `attachedTo` for an attached state machine	A KStates text, also a fragment

`setModel` reads the model against the metamodel of the environment and answers 409 with `Metamodel must be set before receiving a model for environment <envKey>` when there is none. A new metamodel after the model does not read the model again.

`setEventDefinitions` reads the event definitions with the types of the metamodel at hand, so an event type may use the types of the domain. Upload the metamodel first when it does. The event definitions are 2.1 KMeta types, one per event, named like the event.

`setStateMachine` takes the two forms of 2.3.11 State machine management and lifecycle. With `attachedTo`, the text must be an attached state machine, `STATEMACHINE ATTACHED TO "Type"`, and the runtime attaches it to the object of the model whose key is `attachedTo`. One object has at most one attached state machine; a second upload for the same key replaces it and keeps its place in the order of the state machines. The runtime does not compare the type in `ATTACHED TO` with the type of the object.

Without `attachedTo`, the text must be an attachable state machine, `STATEMACHINE "name" ATTACHABLE TO "Type", ...`, and the runtime keeps it as a definition under its name. Objects get instances of it at run time, as 3.1.13 State machine management and lifecycle describes. A second upload with the same name replaces it and keeps its place. An environment holds any

number of definitions besides its attached state machines.

Without `attachedTo`, the text may also be a fragment, `STATEMACHINE "name" FRAGMENT`, of 2.3.12 Pages. The runtime keeps it under its name, and the container states of the other state machines use it, as 3.1.14 Pages describes. A fragment is never attached to an object. Upload every fragment that a state machine of the environment uses, directly or through other fragments, before `runEnvironment`. Attachable state machines and fragments share one name space: an upload replaces the one of the same name, whichever form it had. Of the same form it keeps the place of the old one; of the other form the old one leaves and the new one comes at the end of its kind. `observatory/statemachine` answers the text of a fragment by its name; the route `observatory/stateMachines` does not list fragments. An empty name in the header of a fragment or an attachable state machine, `STATEMACHINE "" FRAGMENT`, and an empty name after `USE` are errors of the reader, so such a text is refused as 3.4.4 Rejected uploads describes.

The runtime reads the text with all optional features of the language switched on. A text in the wrong form answers 400 with one of these texts, and the environment stays as it was:

- `The state machine for '<attachedTo>' must have the attached form STATEMACHINE ATTACHED TO "Type", but it is the attachable state machine '<name>'; upload an attachable state machine without attachedTo`
- `The state machine for '<attachedTo>' must have the attached form STATEMACHINE ATTACHED TO "Type", but it is the fragment '<name>'; upload a fragment without attachedTo`
- `A state machine without attachedTo must have the attachable form STATEMACHINE "name" ATTACHABLE TO "Type" or the fragment form STATEMACHINE "name" FRAGMENT; give attachedTo for a state machine ATTACHED TO a type`

```
curl -X PUT "http://127.0.0.1:8080/setMetamodel?envKey=env-1763035112345" --data-binary
    ↪ @lamp.kmeta
curl -X PUT "http://127.0.0.1:8080/setModel?envKey=env-1763035112345" --data-binary @lamp.
    ↪ kmodel
curl -X PUT "http://127.0.0.1:8080/setEventDefinitions?envKey=env-1763035112345" --data-
    ↪ binary @lamp-events.kmeta
curl -X PUT "http://127.0.0.1:8080/setStateMachine?envKey=env-1763035112345&attachedTo=lamp
    ↪ " --data-binary @lamp.kstates
curl -X PUT "http://127.0.0.1:8080/setStateMachine?envKey=env-1763035112345&attachedTo=
    ↪ wallSwitch" --data-binary @switch.kstates
curl -X PUT "http://127.0.0.1:8080/setStateMachine?envKey=env-1763035112345" --data-binary
    ↪ @glowing.kstates
curl -X PUT "http://127.0.0.1:8080/setStateMachine?envKey=env-1763035112345" --data-binary
    ↪ @bulb-test.frag.kstates
```

Use `--data-binary`; `-d` drops the line breaks, and the lines of errors would no longer fit the file.

3.4.4 Rejected uploads

An upload with errors answers 400. The body has a first line that names the upload and the number of errors, and one line per error with line, column, message and code, separated by line feeds, without a line feed at the end:

```
Upload of the <upload> rejected: <n> errors
<line>:<column>: <message> (<code>)
```

`<upload>` is `metamodel`, `model`, `event definitions`, `state machine for '<attachedTo>'` or, for an upload without `attachedTo`, `fragment` when the text reads as a fragment and `attachable state machine` otherwise. For one error the first line ends in `1 error`. The codes and messages are the checks of 2.4 Checks.

A metamodel with an unknown property type and a relation to a missing type:

```
type "Bulb" "A light bulb" {
    prop("watts", "decimal")
}
type "Lamp" "A lamp" {
    has("bulb", "Bulb")
    knows("control", "Switch")
}
```

```
Upload of the metamodel rejected: 2 errors
2:20: Unknown type 'decimal' of prop 'watts'; a prop is of type 'string', 'number', '
    ↪ boolean' (kmeta.unknown-property-type)
6:23: Unknown class type 'Switch' of knows 'control' in type 'Lamp' (kmeta.unknown-target-
    ↪ type)
```

A model against the lamp metamodel with a wrong number and an unknown property:

```
make object "lamp":"Lamp" {
    prop("mode") -> "off"
    has("bulb") -> make object "bulb":"Bulb" {
        prop("watts") -> "bright"
        prop("colour") -> "white"
    }
}
```

```
Upload of the model rejected: 2 errors
4:27: 'bright' is not a number, but prop 'watts' is of type 'number' (kmodel.invalid-number
    ↪ )
5:15: Class type 'Bulb' has no property 'colour' (object 'bulb') (kmodel.unknown-property)
```

A state machine with a transition to a state that does not exist:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" { }
        STATE "on" { }
    }
    TRANSITIONS {
```

```
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "dark" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

```
Upload of the state machine for 'lamp' rejected: 1 error
10:29: No state is named 'dark'; check the spelling or define the state (kstates.unknown-
    ↪ state)
```

3.4.5 Settings

Both settings work only in the phase `configuring` and answer 409 like the uploads after it.

Route	Parameters	What it sets
`POST /setTickDelay`	`envKey`, `milliseconds`	The tick delay, the least time from the start of one tick to the start of the next
`POST /setEventTtl`	`envKey`, `ttlMs`	The time to live of events in milliseconds; 0 means events never expire

A value that is no whole number gives 400 with `Missing or invalid required parameter: milliseconds` or `Missing or invalid required parameter: ttlMs`. Without these calls the environment uses `engine.defaultTickDelayMs` and `engine.defaultEventTtlMs` of 3.3.4 Configuration.

```
curl -X POST "http://127.0.0.1:8080/setTickDelay?envKey=env-1763035112345&milliseconds=500"
curl -X POST "http://127.0.0.1:8080/setEventTtl?envKey=env-1763035112345&ttlMs=5000"
```

3.4.6 Clients

`POST /registerClientForWebSocket` with `clientId` and `envKey` registers a client for the WebSocket of the environment and answers an access key as text, in the form `ak-<clientId>-<milliseconds>-<number>`. The client sends it when it connects, as 3.5 WebSocket describes. An access key works for one connection. The route works in every phase. The client id is chosen by the client.

`POST /registerObjectObserver` with `envKey`, `clientId` and `objectId` lets the client get the message `objectChanged` when the object with this key changes and `objectDeleted` when it is removed. `POST /registerDomainListener` with `envKey`, `clientId` and `domain` lets the client get the message `domainEvent` for every event of this domain. Both work only in the phase `configuring` and answer 409 after it.

```
curl -X POST "http://127.0.0.1:8080/registerClientForWebSocket?clientId=panel&envKey=env
    ↪ -1763035112345"
curl -X POST "http://127.0.0.1:8080/registerObjectObserver?envKey=env-1763035112345&
    ↪ clientId=panel&objectId=bulb"
curl -X POST "http://127.0.0.1:8080/registerDomainListener?envKey=env-1763035112345&
    ↪ clientId=panel&domain=switch"
```

```
ak-panel-1763035113001-4821
```

3.4.7 Running

`POST /runEnvironment` with `envKey` fixes the configuration and makes the environment ready to start. The phase becomes `activated`. The call answers 409 with

- `Environment with key <envKey> must have both a metamodel and a model before it can be run` without metamodel or model,
- `Environment with key <envKey> has state machines attached to objects missing from its model: <keys>` with the keys separated by `, `, when a state machine is attached to a key that no object of the model has,
- `Environment with key <envKey> has types that behave after state machines it does not hold: <list>` when a type of the metamodel behaves after a name that no attachable state machine of the environment has; each entry of the list reads `'Bulb' behaves after 'glowing'`, separated by `, `,
- `Environment with key <envKey> has types that behave after state machines not attachable to them: <list>` when the attachable state machine does not list the type; each entry reads `'Bulb' behaves after 'glowing', which is attachable to 'Lamp'`, separated by `; `,
- `Environment with key <envKey> has state machines that cannot be flattened:` followed by one line per problem, when a state machine does not flatten with the fragments of the environment, as 3.1.14.1 Flattening at activation describes,
- `Environment with key <envKey> is already active and cannot be modified in the intended way.` when it ran before.

Each line of a failed flattening names the state machine of the environment, `State machine for '<attachedTo>'` or `Attachable state machine '<name>'`, then a colon and the problem as 2.3.12 Pages words it. The lines are separated by line feeds, without one at the end. A lamp whose container state uses a fragment that was not uploaded:

```
Environment with key env-1763035112345 has state machines that cannot be flattened:
State machine for 'lamp': Container state 'self test' of the state machine attached to '
    ↪ Lamp' uses 'bulb test', but no fragment is named 'bulb test'
```

The problems are a container state that uses a name no fragment has, or an attachable state machine instead of a fragment, a fragment that contains itself, a qualified name equal to another state name, two different macros of one name, a macro call that no file of the composition defines, and a macro call whose arguments do not fit the macro that another file of the composition defines. The last one reads, for a lamp that calls `half` of its fragment with two arguments:

```
State machine for 'lamp': In the state machine attached to 'Lamp': Macro 'half' takes 1
    ↪ argument, but the call passes 2; pass one argument per TAKES
```

The environment stays configurable: upload what is missing and call `runEnvironment` again.

An environment without state machines can run; it only ticks.

`POST /startEnvironment` with `envKey` starts the ticks. The phase becomes `running`. Starting a running environment changes nothing and answers 200. Starting a stopped environment continues the frozen run with the next tick, as 3.1.13.9 Stop and start, forcing and clients describes, and tells every connected client with `engineStatus` and the phase `running`.

`POST /stopEnvironment` with `envKey` freezes the environment. The phase becomes `stopped`. The runtime waits up to 5 s for the running tick to end, keeps everything for the next start and tells every connected client with `engineStatus` and the phase `stopped`. The WebSocket connections stay open, the snapshot and the other routes of 3.4.8 Observation answer, and events that clients send wait for the start. Uploads and settings stay refused as after `runEnvironment`. Stopping a stopped environment, or one that was never started, changes nothing and answers 200. The environment stays frozen until it is dropped or the runtime ends. The WebSocket connections close when the environment is dropped, with code 1000 and the reason `Environment dropped`, or when the runtime itself ends, with code 1001 and the reason `Runtime stopped`.

Before `runEnvironment`, `startEnvironment` and `stopEnvironment` answer 409 with `Environment with key '<envKey>' has no execution thread`.

`POST /setActiveState` with `envKey`, `modelElement` and `state` forces the state machine instance of `modelElement` into `state`, also an instance that joined during the run, as 3.1.11 Forcing a state describes. It answers 409 with `Environment with key <envKey> is not active` before `runEnvironment`, and 400 with `No state machine for model element '<modelElement>' in environment <envKey>` or `Unknown state '<state>' for model element '<modelElement>'`. While the environment is stopped, the state is applied at the first tick after the start. A state inside a container state is named with its qualified name, such as `self test/lit`, and a container state goes down into its fragment, as 3.1.14.2 Qualified names describes.

```
curl -X POST "http://127.0.0.1:8080/runEnvironment?envKey=env-1763035112345"
curl -X POST "http://127.0.0.1:8080/startEnvironment?envKey=env-1763035112345"
curl -X POST "http://127.0.0.1:8080/setActiveState?envKey=env-1763035112345&modelElement=
    ↪ lamp&state=dimmer"
curl -X POST "http://127.0.0.1:8080/stopEnvironment?envKey=env-1763035112345"
```

3.4.8 Observation

`GET /observatory/environments` lists every environment that was run, also stopped ones, with the keys of the objects that have state machine instances, in the order in which they step. Before the first start these are the instances the start will build, afterwards those of the latest tick:

```
[{"envKey":"env-1763035112345","modelElements":["lamp","wallSwitch"]}]
```

`GET /observatory/statemachine` with `envKey` and `modelElement` answers the KStates text of the state machine of the object: the text of the attachable state machine its instance runs, or else the text uploaded for it with `attachedTo`, or 400 with `No state machine source found for model element '<modelElement>' in environment <envKey>`. With `name` instead of `modelElement` it answers the text of the attachable state machine or the fragment of that name,

or 400 with `No attachable state machine named '<name>' in environment <envKey>`. Without both it answers 400 with `Missing required parameter: modelElement`.

`GET /observatory/stateMachines` with `envKey` answers, as JSON with the content type `application/json`, what the environment holds: the keys of the attached state machines and the attachable state machines with their types, both in upload order, and the instances in the order in which they step, each with the key of its object and the name of its definition, `null` for an attached state machine. Before `runEnvironment` the list of instances is empty. The route works in every phase.

```
{"attached":["living room"],"definitions":[{"name":"glowing","types":["Bulb"]},{"name":"
    ↪ blinking","types":["Bulb"]}],"instances":[{"modelElement":"living room","
    ↪ stateMachine":null},{"modelElement":"bulb","stateMachine":"glowing"}]}
```

`POST /observatory/registerClient` with `clientId`, `envKey` and `modelElement` registers a client for the WebSocket like `registerClientForWebSocket` and answers an access key. The client also gets the message `stateChange` of the state machine attached to `modelElement`, or of all state machines when `modelElement` is `*`. The route works in every phase.

`GET /observatory/model` with `envKey` answers the whole model of the environment as JSON with the content type `application/json; charset=UTF-8`:

```
{"environmentKey":"env-1763035112345","phase":"configuring","revision":0,"activeStates
    ↪ ":{},"stateMachines":{},"roots":[{"__id__": "living room", "__type__": "Room", "lamp
    ↪ ": {"__id__": "lamp", "__type__": "Lamp", "mode": "off", "bulb": {"__id__": "bulb",
    ↪ "__type__": "Bulb", "maxWatts": 40, "watts": 0, "lit": false}, "spare": {"__id__": "
    ↪ spare", "__type__": "Bulb", "maxWatts": 60, "watts": 0, "lit": false}, "control": {"
    ↪ __id__": "wallSwitch", "__type__": "Switch", "position": "down"}}, "switch": {"
    ↪ __id__": "wallSwitch", "__type__": "Switch", "position": "down"}}]}
```

Entry	Meaning
`environmentKey`	The key of the environment
`phase`	`configuring`, `activated`, `running`, `halted` or `stopped`, as 3 Execution describes
`revision`	The revision of the last change the snapshot contains, 0 before the first change
`activeStates`	For each object with a state machine instance, the last reported stack, with the qualified names of 3.1.14.2 inside container states. Empty before the first tick. An object leaves it when its instance leaves
`stateMachines`	For each object with a state machine instance, the name of its definition, `null` for an attached state machine. Empty before the first start
`roots`	The objects at the top of the model, written like the `value` of the message `objectChanged`

In the phase `configuring` the snapshot is the uploaded model at revision 0, and without a model the route answers 409 with `Environment with key <envKey> has no model`. In the other phases the runtime reads the model between two ticks, also while a breakpoint halts the run. When no tick ends within 10 s, the route answers 409 with `Environment with key <envKey> did not reach a tick boundary within 10000 ms`. A model in which an object reaches itself over its relations cannot be written as JSON; the route answers 409 with

`Reference loop through object '<key>' cannot be written as JSON.`

`POST /observatory/subscribeModel` with `envKey` and `clientId` lets the client get `objectChanged` and `objectDeleted` for every object of the model and `stateChange` for every state machine. The client must be connected to the WebSocket of the environment; otherwise the route answers 409 with `Client <clientId> has no open WebSocket session for environment <envKey>`. The subscription stays when the client connects again. `POST /observatory/unsubscribeModel` with the same parameters ends it; it answers 200 also when there was none. Both work in every phase. How a client keeps a copy of the whole model with a snapshot and these messages is in 3.5 WebSocket.

3.4.9 Health

`GET /health` answers `{"status":"ok"}` with the content type `application/json` while the runtime runs.

3.4.10 Features

Route	Parameters	Answer
`GET /getRegisteredFeatures`		The names of all features, separated by commas
`GET /getActiveFeatures`	`envKey`	The names of the features on in the environment, separated by commas
`POST /activateFeature`	`envKey`, `featureName`	Empty
`POST /deactivateFeature`	`envKey`, `featureName`	Empty
`POST /sendMessageToFeature`	`envKey`, `featureName`, the message as body	The answer of the feature

The features and their messages are in 3.2 Runtime features. The order of the names in the answers is not fixed. All routes work in every phase. The errors, all with status 400:

Text	When
`Feature '<name>' is not registered`	The name is no feature
`Feature '<name>' is already active in environment <envKey>`	`activateFeature` for a feature that is on
`Feature '<name>' is not active in environment <envKey>`	`deactivateFeature` for a feature that is off
`<name> has not been activated in environment <envKey>`	`sendMessageToFeature` to a feature that is off

```
curl "http://127.0.0.1:8080/getRegisteredFeatures"
```

```
History,TickLimit,TickByTick,Breakpoint,EventRecording,EventInjection,EventReplayController,
    ↪ TrueParallel
```

3.4.11 Whole session

A session that runs the lamp, steps it and watches the bulb, from the folder `examples/lamp` of the IDE:

```
R=http://127.0.0.1:8080
ENV=$(curl -s -X POST "$R/createEnvironment")
curl -X PUT "$R/setMetamodel?envKey=$ENV" --data-binary @lamp.kmeta
curl -X PUT "$R/setModel?envKey=$ENV" --data-binary @lamp.kmodel
curl -X PUT "$R/setEventDefinitions?envKey=$ENV" --data-binary @lamp-events.kmeta
curl -X PUT "$R/setStateMachine?envKey=$ENV&attachedTo=lamp" --data-binary @lamp.kstates
curl -X PUT "$R/setStateMachine?envKey=$ENV&attachedTo=wallSwitch" --data-binary @switch.
    ↪ kstates
curl -X POST "$R/setTickDelay?envKey=$ENV&milliseconds=500"
curl -X POST "$R/setEventTtl?envKey=$ENV&ttlMs=5000"
KEY=$(curl -s -X POST "$R/registerClientForWebSocket?clientId=panel&envKey=$ENV")
curl -X POST "$R/registerObjectObserver?envKey=$ENV&clientId=panel&objectId=bulb"
curl -X POST "$R/activateFeature?envKey=$ENV&featureName=TickByTick"
curl -X POST "$R/runEnvironment?envKey=$ENV"
```

The client `panel` now connects to `ws://127.0.0.1:8080/ws` and sends `panel:<ENV>:<KEY>` as its first frame. Then:

```
curl -X POST "$R/startEnvironment?envKey=$ENV"
curl -X POST "$R/sendMessageToFeature?envKey=$ENV&featureName=TickByTick" -d "pause"
```

The client sends a press over the WebSocket:

```
{"environmentKey": "switch", "messageType": "press", "payload": ""}
```

```
curl -X POST "$R/sendMessageToFeature?envKey=$ENV&featureName=TickByTick" -d "tick:2"
curl "$R/observatory/model?envKey=$ENV"
curl -X POST "$R/stopEnvironment?envKey=$ENV"
```

Within the two ticks the lamp switches on. The client gets `objectChanged` for `bulb` after the ENTRY block of `on`, with `lit` true, and after the ENTRY block of `full`, with `watts` 40. The snapshot shows `mode on` and the stack `on`, `full` of the lamp.

When the client no longer needs the run, it drops the environment, and its WebSocket connection closes with code 1000 and the reason `Environment dropped`:

```
curl -X POST "$R/dropEnvironment?envKey=$ENV"
```

3.5 WebSocket

Clients send events to an environment and get its changes pushed over the WebSocket at `ws://127.0.0.1:8080/ws`, on the host and port of 3.3.4 Configuration. All messages are text frames.

3.5.1 Connecting

A client first registers over HTTP with `registerClientForWebSocket` or `observatory/registerClient` of 3.4 HTTP API and gets an access key. Then it opens the WebSocket and sends as its first frame its client id, the key of the environment and the access key, separated by colons:

```
panel:env-1763035112345:ak-panel-1763035113001-4821
```

The runtime closes the connection with code 1003 and one of these reasons when the first frame is wrong:

Reason	When
`First message must be text`	The first frame is no text frame
`Invalid auth format`	The frame does not have exactly three parts
`Invalid credentials`	The access key is unknown, was used before, or belongs to another client or environment

The runtime sends no answer to a correct first frame. A client knows that it is connected when `observatory/subscribeModel` succeeds or when the first message arrives. Each access key works for one connection; a client that connects again registers again. The runtime sends a ping every 60 s and closes a connection that does not answer within 15 s.

The connections stay open when the environment is stopped and started again; the clients learn both through `engineStatus`. Two closes end them from the side of the runtime:

Code	Reason	When
1000	`Environment dropped`	`POST /dropEnvironment` dropped the environment of the connection
1001	`Runtime stopped`	The runtime itself ends

After either close the access key of the connection is gone. After a drop the environment is gone too, and a client that wants to go on needs a new environment.

3.5.2 Sending events

A client sends an event as a JSON object:

```
{"environmentKey": "switch", "messageType": "dim", "payload": "{\"bright\": true}", "
    ↪ payloadFormat": "json"}
```

Entry	Meaning
`environmentKey`	The domain of the event, as in `EVENT("switch", "dim")`. Despite its name it is not the key of the environment
`messageType`	The name of the event
`payload`	The payload as a string. Optional, empty by default
`payloadFormat`	`none`, `json` or `kmodel`, in any case. Optional

Without `payloadFormat` the runtime takes the format from the payload: empty is no payload, text that starts with `{` is JSON, text that starts with `make object` is KModel. The runtime reads the payload against the event type named like the event, as 3.1.7.4 Payloads describes. A payload in KModel is one `make object` of that type:

```
{"environmentKey": "switch", "messageType": "dim", "payload": "make object \"step\":\"dim\"
    ↪ { prop(\"bright\") -> \"true\" }"}
```

The event goes to the environment the connection belongs to. A frame that is no JSON object with `environmentKey` and `messageType` is ignored. Events sent before `runEnvironment` are dropped. Events sent after `runEnvironment` and before `startEnvironment` wait for the start. The runtime sends no answer to an event.

3.5.3 Pushed messages

Every message the runtime pushes has the same frame:

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"objectChanged","
    ↪ payload":{}}
```

Here `environmentKey` is the key of the environment. `payload` is a JSON object that depends on `messageType`. The runtime pushes nothing when `websocket.enabled` is `false`.

Message	Who gets it	When
`objectChanged`	Observers of the object, model subscribers	An action block changed a property or relation of the object
`objectDeleted`	Observers of the object, model subscribers	An action removed the object
`stateChange`	Clients of `observatory/registerClient` for this state machine or `*`, model subscribers	The stack of a state machine changed, and in the first tick after the first start
`stateMachineAttached`	The same clients as `stateChange` of the object	A state machine instance joined: at the first start for every instance, and at the end of a tick for each that joined in it
`stateMachineDetached`	The same clients as `stateChange` of the object	A state machine instance left, at the end of the tick in which it ended
`domainEvent`	Listeners of the domain	An event of the domain reached the bus
`engineStatus`	Every client connected to the environment	The run starts or ends a wait of TickByTick, the environment was stopped, or a stopped environment was started

3.5.3.1 objectChanged

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"objectChanged","
    ↪ payload":{"objectId":"bulb","revision":7,"value":{"__id__": "bulb", "__type__": "
    ↪ Bulb", "maxWatts": 40, "watts": 40, "lit": true}}}
```

`value` is the whole object after the action block: `__id__` with its key, `__type__` with its type, its properties in the order of the model, and its relations with the objects in them, again whole. Objects in a `knows` relation are written whole as well. A property without a value is left out. The message names the object whose own property or relation changed. A change of the bulb gives a message for `bulb` and none for `lamp`, which holds it.

A block that changes several objects gives one message per object, after the block.

3.5.3.2 objectDeleted

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"objectDeleted","
    ↪ payload":{"objectId":"spare","revision":9}}
```

`DROPOBJ("spare")` gives `objectDeleted` for `spare` and every object it holds, and `objectChanged` for the lamp that held it. The messages of the changes come first.

3.5.3.3 stateChange

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"stateChange","
    ↪ payload":{"modelElementId":"lamp","stack":["on","full"],"revision":8}}
```

`modelElementId` is the key of the object of the state machine. `stack` is its stack from the outside in; in a split it is the stacks of all branches one after the other. The order of the entries in `payload` is not fixed.

Inside a container state the stack names the states of the fragment with their qualified names, as 3.1.14.2 Qualified names describes. A lamp in the self test of 2.3.12 Pages reports `"stack": ["self test","self test/powered"]`. A stack never ends on a container state.

3.5.3.4 domainEvent

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"domainEvent","
    ↪ payload":{"domain":"switch","payload":{"event":"dim","payload":"{\"bright\": true
    ↪ }","payloadFormat":"json","source":"EXTERNAL_MESSAGE","timestamp":1763035114120,"
    ↪ ttlMs":5000}}}
```

Entry	Meaning
`event`	The name of the event
`payload`	The payload as the sender wrote it, as a string
`payloadFormat`	`none`, `json` or `kmodel`
`source`	`EXTERNAL_MESSAGE` for events from clients and from EventInjection, `INTERNAL` for events raised with the action `EVENT`
`timestamp`	When the runtime got or raised the event, in milliseconds since 1970
`ttlMs`	The time to live of the event

The order of the entries inside the inner `payload` is not fixed. A `domainEvent` has no revision.

3.5.3.5 engineStatus

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"engineStatus","
    ↪ payload":{"paused":true,"waiting":true}}
```

The payload is the status of 3.2.5.3 Status, with `halt` while a breakpoint holds the run. An `engineStatus` has no revision.

A stop and a start after it are told with the member `phase`, `stopped` or `running`, the phase of the snapshot, next to the status at that moment; without TickByTick the status is `{"paused":false,"waiting":false}`:

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"engineStatus","
    ↪ payload":{"phase":"stopped","paused":false,"waiting":false}}
```

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"engineStatus","
    ↪ payload":{"phase":"running","paused":false,"waiting":false}}
```

The `engineStatus` messages of TickByTick have no `phase`. A stop while the run is paused says `"paused":true` and `"waiting":true`, and so does the start after it, because the run still waits. A client that reads only `paused` and `waiting` keeps working; it does not learn of the stop. The first start of an environment sends no `engineStatus`.

3.5.3.6 stateMachineAttached and stateMachineDetached

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"
    ↪ stateMachineAttached","payload":{"modelElementId":"gen_Bulb_0","stateMachine":"
    ↪ blinking","revision":12}}
```

```
{"environmentKey":"env-1763035112345","clientId":"panel","messageType":"
    ↪ stateMachineDetached","payload":{"modelElementId":"bulb","stateMachine":"glowing","
    ↪ revision":15}}
```

`modelElementId` is the key of the object, the key that `stateChange` uses for the instance. `stateMachine` is the name of the attachable state machine the instance runs, and `null` for a state machine uploaded with `attachedTo`. So a client tells an instance, named by the object, from its definition, named by `stateMachine`. The order of the entries in `payload` is not fixed.

At the first start the runtime sends `stateMachineAttached` for every instance before the first tick, in the order in which they step. A start after a stop continues the run and announces nothing anew. During a run, an instance that joins is announced at the end of the tick it joined in, after the `stateChange` messages of that tick, and its first `stateChange` follows after its first step. An instance that leaves is announced at the end of the tick in which it ended, before the instances that join in the same tick; after it no `stateChange` comes for it. When `EXCHANGE` replaces an instance, `stateMachineDetached` with the old definition comes before `stateMachineAttached` with the new one, for the same object. `objectDeleted` of a removed object comes earlier, when the action removes it.

A client that does not know these messages can ignore them. It still gets `stateChange` for

every instance, also for those that join during the run, but it does not learn when one leaves.

3.5.4 Revisions

`objectChanged`, `objectDeleted`, `stateChange`, `stateMachineAttached` and `stateMachineDetached` carry a revision. Each environment counts its revisions from 1 on, one count for all five messages and all clients, in the order of the changes. The count goes on after a stop and a start; the two add no revision. A change of an object that cannot be written as JSON, because it reaches itself over its relations, uses up a revision and sends nothing.

3.5.5 Keeping a copy of the model

A client that shows the whole model keeps a copy this way:

1. Register with `registerClientForWebSocket`, connect, and call `observatory/subscribeModel`.
2. Read `observatory/model`. Its `revision` is the last change it contains.
3. Ignore pushed messages with a revision up to the one of the snapshot, and apply the later ones in order: replace the object for `objectChanged`, remove it for `objectDeleted`, replace the stack for `stateChange`, add the object with its definition to `stateMachines` for `stateMachineAttached`, and remove the object from `stateMachines` and `activeStates` for `stateMachineDetached`.
4. When a revision is missing, read `observatory/model` again.

3.5.6 Delivery

The runtime sends the messages of one client in the order they were made. A client that has more than one connection to the same environment gets each message on one of them. Messages for a client without a connection are dropped. After 10 s without a connection, the runtime also ends the registration of the client made with `observatory/registerClient`.

3.6 Examples

The folder `examples` of the runtime has four example models outside the lamp domain. Each folder has a metamodel, a model, one state machine, event definitions in `EVENTS.kmeta` and a `README.md` with the steps to run it. The lamp itself is in the folder `examples/lamp` of the IDE, and a room with spare bulbs whose state machines objects get at run time, as 3.1.13 State machine management and lifecycle describes, in the folder `examples/spares` of the IDE.

Folder	What it shows	Object of the state machine	Events
`single_robot_example`	A cleaning robot that drives around a room and steers away from obstacles and walls: nested states, `NOT LOOPING`, `EVAL` conditions, `SETOBJ` on a `knows` relation, `WITH ... AS`, macros with `numpy`, events raised by the state machine	`turtle`	`start` in domain `public`. `EVENTS.kmeta` also defines `setSpeed` and `teleport`, which the state machine does not use

Folder	What it shows	Object of the state machine	Events
`delivery_drone_example`	A delivery drone driven by events with payloads: conditions on payload fields, `$(event->...)` in ENTRY blocks, `IF IN SCOPE`, payload values copied into the model	`scout`	`assignDelivery`, `setCruiseSpeed`, `recall` in domain `control`
`autonomous_car_example`	A car that tests three subsystems in parallel before it drives: `SPLIT` into three branches and `JOIN` back	`av-1`	`powerOn` in domain `ignition`
`drone_swarm_example`	A swarm that grows, reshapes and shrinks its fleet: `APPENDOBJ`, `SETOBJ`, `DROPOBJ`, `SETLIST`, `DROPLIST`, `WITH ... AS`	`alpha-swarm`	`deploy`, `droneDown`, `recall` in domain `swarm`

Folder	Files
`single_robot_example`	`cleaning_robot.kmeta`, `cleaning_robot.kmodel`, `cleaning_robot.kstates`, `EVENTS.kmeta`
`delivery_drone_example`	`delivery_drone.kmeta`, `delivery_drone.kmodel`, `delivery_drone.kstates`, `EVENTS.kmeta`
`autonomous_car_example`	`autonomous_car.kmeta`, `autonomous_car.kmodel`, `autonomous_car.kstates`, `EVENTS.kmeta`
`drone_swarm_example`	`drone_swarm.kmeta`, `drone_swarm.kmodel`, `drone_swarm.kstates`, `EVENTS.kmeta`

All four use `EVAL` blocks or macros, so run them with Python at hand. The macros of the robot import `numpy`. The Docker image of 3.3.3 Running in Docker has both.

To run an example:

1. Start the runtime as 3.3 Running the runtime describes.
2. In the folder of the example, create an environment and upload the files. For the car:

```
ENV=$(curl -s -X POST http://127.0.0.1:8080/createEnvironment)
curl -X PUT "http://127.0.0.1:8080/setMetamodel?envKey=$ENV" --data-binary
    ↪ @autonomous_car.kmeta
curl -X PUT "http://127.0.0.1:8080/setModel?envKey=$ENV" --data-binary
    ↪ @autonomous_car.kmodel
curl -X PUT "http://127.0.0.1:8080/setEventDefinitions?envKey=$ENV" --data-binary
    ↪ @EVENTS.kmeta
curl -X PUT "http://127.0.0.1:8080/setStateMachine?envKey=$ENV&attachedTo=av-1" --
    ↪ data-binary @autonomous_car.kstates
curl -X POST "http://127.0.0.1:8080/setTickDelay?envKey=$ENV&milliseconds=300"
```

3. Register a WebSocket client and an observer of the object of the state machine:

```
KEY=$(curl -s -X POST "http://127.0.0.1:8080/registerClientForWebSocket?clientId=
    ↪ viewer&envKey=$ENV")
curl -X POST "http://127.0.0.1:8080/registerObjectObserver?envKey=$ENV&clientId=
```

```
    ↪ viewer&objectId=av-1"
```

4. Run and start the environment:

```
curl -X POST "http://127.0.0.1:8080/runEnvironment?envKey=$ENV"
curl -X POST "http://127.0.0.1:8080/startEnvironment?envKey=$ENV"
```

5. Connect the client to `ws://127.0.0.1:8080/ws`, send `viewer:<ENV>:<KEY>` as the first frame and then the event that starts the example:

```
{"environmentKey": "ignition", "messageType": "powerOn", "payload": ""}
```

The client gets `objectChanged` for the car each time an action block changes it. The README of each example names its events and payloads. The connection and the messages are in 3.5 WebSocket.

4 The IDE

karpfen IDE is a desktop program for the files of a karpfen system. It opens a folder as a project, edits the KMeta, KModel and KStates files and the environment descriptors, and checks them while you type. It draws each of the three kinds of model file as a diagram that can also be edited. It validates the whole project, runs an environment descriptor in a karpfen runtime inside a Docker container of its own, shows the model and the active states while the model runs, sends events into the run, halts the run at breakpoints and connects it to HTTP, MQTT and ROS through bridges. The same program validates and runs a project from a terminal, without a window.

The window has a menu bar, an editor area in the middle, tool windows around it and a status bar at the bottom. Each tool window can be shown and hidden in the View menu, and its border to the editor area can be dragged. The Runtime tool window can also be moved into a window of its own.

Part	Place	What it shows
Project	Left	The files of the folder in three sections: the project file and the descriptors, the folder, the bridge files.
Editor area	Middle	One tab per open file. Below the editor a bar holds Text, Split and Diagram for files with a diagram, the Validate button and the font size of the editor. Above the diagram of a fragment a bar names the files that use it.
Outline	Right	What the active file is made of: the types of a metamodel, the objects of a model, the states and transitions of a state machine, and the header of a fragment.
Problems	Bottom	The problems of the project in the groups Local and Global. It is hidden while there is no problem.
Language Tools Output	Bottom	The log of the language tools that read and check the files.
Runtime	Bottom	The runtime container, the runs, the debugger and three panes for the model, the events and the log of the selected run.
Status bar	Bottom edge	On the left the problems, `Analyzing...` while the files are checked, the state of the runtime and unseen messages of the language tools. On the right the line and column, the language, the line separator, the encoding, the counts of errors and warnings, and the zoom of the whole window.

The window has two zoom controls. `Zoom 100%` at the right end of the status bar, between a minus and a plus, scales everything in the window. `Font size 13` in the bar below the editor scales the text of the editors only. A click on either size sets it back to normal.

The menu bar has the menus File, Edit, View, Navigate and Run. The table lists every entry in the order of the menu bar. Open Recent is a submenu with one entry per folder, and Run > Language Features is a submenu whose entries read `Switch On` or `Switch Off` before the name of the feature, after what a click does.

Menu	Entry	What it does	Described in
File	Open Folder…	Opens a folder as the project.	4.1 Projects
File	Open Recent	Opens one of the folders opened before.	4.1 Projects
File	Save	Saves the file of the active tab.	4.2.8 Saving and changes outside the IDE
File	Save All	Saves every file with unsaved changes.	4.2.8 Saving and changes outside the IDE
File	Close Tab	Closes the active tab.	4.2.10 Keys
Edit	Undo	Takes back the last edit, also one made in a diagram.	4.2.10 Keys
Edit	Redo	Makes an edit that was taken back again.	4.2.10 Keys
Edit	Cut	Moves the selection to the clipboard.	4.2.10 Keys
Edit	Copy	Copies the selection to the clipboard.	4.2.10 Keys
Edit	Paste	Inserts the clipboard at the caret.	4.2.10 Keys
Edit	Select All	Selects the whole text of the editor.	4.2.10 Keys
Edit	Find…	Opens the search bar above the text.	4.2.7 Find and replace
Edit	Replace…	Opens the search bar with a field for the replacement.	4.2.7 Find and replace
Edit	Find Next	Selects the next match.	4.2.7 Find and replace
Edit	Find Previous	Selects the match before.	4.2.7 Find and replace
Edit	Toggle Comment	Comments the selected lines out or in.	4.2.10 Keys
Edit	Duplicate Line	Copies the line of the caret below itself.	4.2.10 Keys
Edit	Complete Code	Shows the completion at the caret.	4.2.2 Completion
View	Project	Shows or hides the Project tool window.	4.1.5 The project tree
View	Outline	Shows or hides the Outline tool window.	4.2.5 Outline
View	Problems	Shows or hides the Problems tool window.	4.2.6 Problems
View	Language Tools Output	Shows or hides the log of the language tools.	The table of the window above
View	Runtime	Shows or hides the Runtime tool window.	4.5.4 The Runtime tool window
View	Light Theme	Draws the window in light colours.	4.1.6 The project file
View	Dark Theme	Draws the window in dark colours.	4.1.6 The project file
Navigate	Go to Line…	Moves the caret to a line by its number.	4.2.7 Find and replace

Menu	Entry	What it does	Described in
Navigate	Go to Definition	Leads from a name to its declaration.	4.2.4 Go to definition
Run	Run	Validates, deploys and starts an environment descriptor.	4.5.2 Starting a run
Run	Stop	Stops the selected run.	4.5.5 Stopping a run
Run	Validate Project	Checks the whole project and shows the report.	4.4.3 Starting a validation
Run	Pause	Halts the selected run at the end of its tick.	4.6.3 The debugger line
Run	Resume	Lets a halted run go on up to the next breakpoint.	4.6.3 The debugger line
Run	Step	Lets a halted run go on to the next breakpoint or the end of the tick.	4.6.3 The debugger line
Run	Start Runtime	Starts the runtime container without a run.	4.5.1 The runtime container
Run	Stop Runtime	Stops and removes the runtime container.	4.5.1 The runtime container
Run	Configure Bridge	Opens the form of a bridge, or of a new one.	4.7.4 The bridge form
Run	Language Features > Histories	Switches histories on or off.	4.1.7 Language features
Run	Language Features > Tick limits	Switches tick limits on or off.	4.1.7 Language features
Run	Language Features > Breakpoints	Switches breakpoints and the debugger on or off.	4.1.7 Language features
Run	Language Features > Macros	Switches macros on or off.	4.1.7 Language features
Run	Language Features > Splits and joins	Switches splits and joins on or off.	4.1.7 Language features
Run	Language Features > Parallel branches on threads	Switches the runtime feature TrueParallel on or off for every run.	4.1.7 Language features

Light Theme and Dark Theme switch the colours of the whole window at once. The choice is stored in the project file as the key `theme` with the value `light` or `dark`. Without that line, or with `theme = system`, the window follows the colours of the operating system, and only an edit of the line brings that back.

The sections of this chapter follow the work with a project.

Section	Subject
4.1 Projects	The folder as a project, the kinds of files, the environment descriptor, the project file and the language features.
4.2 Text editor	Colours, completion, hover, navigation, the Outline, the Problems tool window and the keys of the editor.
4.3 Diagrams	The statechart, the class diagram and the object diagram: how to read them, edit them and arrange them by hand.

Section	Subject
4.4 Validating	What is checked while you type, what a validation of the whole project adds, and what blocks a run.
4.5 Running	The Docker container, the run dialog, the Runtime tool window, sending events, observation and captured logs.
4.6 Debugging	Breakpoints and the debugger bar with Pause, Resume and Step.
4.7 Bridges	Bridge files for HTTP, MQTT and ROS, their form and their routes.
4.8 Command line	`karpfen-ide validate` and `karpfen-ide run` without a window.

The languages themselves are the subject of 2 Modeling languages, and what a running model does is the subject of 3 Execution.

4.1 Projects

A project is a folder. File > Open Folder… (Cmd+O / Ctrl+O) opens one, and File > Open Recent lists the folders opened before. The IDE reads every file below the folder, decides from each file name what kind of file it is, and groups the model files into contexts that are checked and run together.

4.1.1 Kinds of files

The kind of a file follows from its name alone.

Name	Kind	Reference
`*.kmeta`	Domain metamodel	2.1 KMeta
`EVENTS.kmeta`, in any case	Event metamodel	2.1 KMeta
`*.kmodel`	Model	2.2 KModel
`*.kstates`	State machine	2.3 KStates
`*.frag.kstates`	Fragment, a page that container states of other state machines run	2.3.12 Pages
`*.kenv.json`	Environment descriptor	4.1.2 Environment descriptors
`*.kbridge.json`	Bridge	4.7 Bridges
`*.vis` next to a model file	Layout of a diagram, written by the IDE	4.3 Diagrams
`.karpfenproject`	Project file	4.1.6 The project file

An event metamodel can also have another name, such as `lamp-events.kmeta`, when a descriptor names it under `events`.

A fragment file holds one fragment, `STATEMACHINE "name" FRAGMENT`, and is named after it: the fragment `bulb_test` lives in `bulb_test.frag.kstates`. The language finds a fragment by its name alone, so the file name is a convention of the IDE that keeps the files easy to find. A fragment file whose name does not fit the name of its fragment, ignoring case, gets the warning `fragment-file-name` at that name, and so does a fragment file that holds a state machine that is no fragment. A fragment in a file named `*.kstates` is read and runs the same, without the warning.

The IDE leaves out these folders: `.git`, `.gradle`, `build`, `out`, `.idea`, `.kotlin`, `node_modules`, `karpfen-runs` and every folder whose name starts with a dot.

4.1.2 Environment descriptors

An environment descriptor is a JSON file named `*.kenv.json`. It says which metamodel, event metamodel, model and state machines form one context, which object of the model each state machine is attached to, and how fast the model runs. A descriptor is what Run starts. The paths in it are relative to the folder of the descriptor and must lie inside the project.

4.1.2.1 Keys

Key	Value	Meaning
`metamodel`	String, required	The domain metamodel, a `.kmeta` file.
`events`	String or `null`	The event metamodel, a `.kmeta` file. Without it the context has no events.
`model`	String, required	The model the runtime starts from, a `.kmodel` file.
`stateMachines`	List of objects	The state machines. Each entry has `file`, a `.kstates` file, and `attachedTo`, the key of the object it runs on. The type of that object should be the type after `ATTACHED TO` in the state machine. An entry without `attachedTo` names an attachable state machine, which the environment keeps as a definition, see 4.1.2.3 Definitions, or a fragment, see 4.1.2.5 Fragments. Without the key the list is empty.
`tickDelayMs`	Number	The time in milliseconds between two ticks. Without the key it is 50.
`eventTtlMs`	Number	How long in milliseconds an event waits for a state machine that takes it. 0 keeps events until a state machine takes them. Without the key it is 3000.
`observe`	Object	`objects`, required, with `"*"` for every object, and `domains`, a list of event domains. A run does not use it, because the IDE observes the whole model and listens to every domain the state machines use.
`runtime`	Object	`image` and `pythonPackages`. A run does not use it, because the IDE runs every model in its own container.
`bridges`	List of strings	Bridge files. A run does not use it. The run dialog chooses the bridges, see 4.5 Running.

The editor reports these problems in a descriptor.

Code	Severity	Cause
`descriptor-missing-file`	Error	A named file does not exist, lies in a left-out folder or lies outside the project.
`descriptor-invalid`	Error	A named file is of the wrong kind for its key, a value is of the wrong kind, a required key is missing, or a key is written twice.
`descriptor-unknown-key`	Warning	The key is not one of the table above.

Code	Severity	Cause
`descriptor-state-machine-form`	Error or warning	An entry with `attachedTo` names an attachable state machine or a fragment, or an entry without it names an attached one, an error. The same definition or fragment is listed twice, a warning.
`deploy-observe-superseded`	Info	The descriptor has `observe`.
`deploy-hosting-ignored`	Info	The descriptor has `runtime` or `bridges`.
`deploy-tick-delay-range`	Error	`tickDelayMs` is below 0 or above 2147483647.
`deploy-tick-delay-zero`	Warning	`tickDelayMs` is 0, so the runtime never waits between ticks.
`deploy-event-ttl-never-expires`	Info or warning	`eventTtlMs` is 0, an info, or negative, a warning. Events then never expire.

4.1.2.2 A minimal descriptor

This descriptor runs the lamp with only its own state machine.

```
{
  "metamodel": "lamp.kmeta",
  "events": "lamp-events.kmeta",
  "model": "lamp.kmodel",
  "stateMachines": [
    { "file": "lamp.kstates", "attachedTo": "lamp" }
  ]
}
```

The descriptor of the example in `examples/lamp` also attaches `switch.kstates` to `wallSwitch` and sets the timing:

```
{
  "metamodel": "lamp.kmeta",
  "events": "lamp-events.kmeta",
  "model": "lamp.kmodel",
  "stateMachines": [
    { "file": "lamp.kstates", "attachedTo": "lamp" },
    { "file": "switch.kstates", "attachedTo": "wallSwitch" }
  ],
  "tickDelayMs": 500,
  "eventTtlMs": 5000
}
```

4.1.2.3 Definitions

A state machine of the form `STATEMACHINE "name" ATTACHABLE TO "Type", ...` is attached to no object by the descriptor. Its entry has only `file`, and the environment keeps it under its name. The objects get instances of it through the `behaves after` of their type, an `ATTACH` clause or an `EXCHANGE`, see 2.3.11 State machine management and lifecycle.

The example in `examples/spares` attaches the room machine to the room and lists two definitions:

```
{
```

```
  "metamodel": "spares.kmeta",
  "events": "spares-events.kmeta",
  "model": "spares.kmodel",
  "stateMachines": [
    { "file": "room.kstates", "attachedTo": "room" },
    { "file": "glowing.kstates" },
    { "file": "flickering.kstates" }
  ],
  "tickDelayMs": 500,
  "eventTtlMs": 5000
}
```

The form of each file must fit its entry. The editor reports `descriptor-state-machine-form` at the file of an entry whose form does not fit.

A run uploads the definitions with the attached state machines, in the order of the descriptor, before it activates the environment. The runtime holds the definitions the descriptor lists and no others, so every name that a `behaves after`, an `ATTACH` or an `EXCHANGE` uses must be listed, see 4.5.4.7 State machine instances.

4.1.2.4 Creating a descriptor

Right-click a folder in the Project tool window and choose New Environment File. A dialog asks for the name, and `.kenv.json` is added. The new descriptor names the files of the folder's current context: the metamodel, the events, the first model and every state machine. Each state machine is attached to the first object of its `ATTACHED TO` type, otherwise to the first object of the model, and otherwise to the placeholder `"object"`. An attachable state machine and a fragment are listed without `attachedTo`. When the state machines use event domains, the descriptor also gets an `observe` key, which a run does not use. The timing keys are left out, so the defaults hold.

4.1.2.5 Fragments

A fragment is listed like a definition, with `file` and without `attachedTo`. A context holds the fragments its descriptor lists and no others, and the container states of its state machines find their fragments among them. A fragment that a container state uses but the descriptor does not list is the error `project.unknown-fragment` at the name after `USE`, see 4.4 Validating.

The example in `examples/pages` composes the lamp from three fragments. The fragment `bulb_test` uses the fragment `wiring_check` in a container state of its own, and the state `on` of the lamp holds a container state that runs `warm_up`:

```
{
  "metamodel": "pages.kmeta",
  "events": "pages-events.kmeta",
  "model": "pages.kmodel",
  "stateMachines": [
    { "file": "lamp.kstates", "attachedTo": "lamp" },
    { "file": "bulb_test.frag.kstates" },
    { "file": "wiring_check.frag.kstates" },
    { "file": "warm_up.frag.kstates" }
  ],
  "tickDelayMs": 500,
  "eventTtlMs": 5000
}
```

4.1.3 Folders without a descriptor

A folder without a descriptor forms a context of its own from the karpfen files directly inside it that no descriptor names. Its metamodel is the one `.kmeta` file that is not `EVENTS.kmeta`, its event metamodel is `EVENTS.kmeta`, and its models and state machines are all `.kmodel` and `.kstates` files of the folder, the `.frag.kstates` files included. An attachable state machine of the folder counts as a definition, a fragment of the folder is part of the context, and no state machine is attached to an object. Such a context is checked like any other, but it cannot be run, because Run needs a descriptor.

Situation	Finding
Several `.kmeta` files other than `EVENTS.kmeta`	The info `context-multiple-metamodels` on each of them. The first in alphabetical order, ignoring case, is the metamodel. The others belong to no context.
No `.kmeta` file other than `EVENTS.kmeta`	The info `context-no-metamodel` on each model and state machine.

A folder that holds a descriptor forms no context of its own. Its karpfen files that no descriptor names belong to no context.

4.1.4 Files that belong to no context

A file that belongs to no context is still checked. A state machine is read on its own, with the language features of the project, and its errors block a run like any other. A model or a metamodel is checked for its syntax only.

4.1.5 The project tree

The Project tool window shows the folder in three sections, divided by lines.

1. The project file `.karpfenproject` and the environment descriptors.
2. The folder with the model files and their subfolders.
3. The bridge files.

A double-click or Enter opens a file. A bridge file opens in its form. The `.vis` file of a diagram shows below its model file, which can then be expanded, and opens as text that cannot be edited.

The context menu of the tree has these entries.

Entry	Effect
New KMeta File, New KModel File, New KStates File, New Fragment File, New Environment File	Asks for a name and creates the file in the folder of the entry, with the suffix added. A new file reads without errors in its context. A new fragment file gets `.frag.kstates` and holds a fragment named after the file, with an initial state, a final state and a transition between them.
Rename…	Renames the file or folder. Not for the project folder itself.

Entry	Effect
Delete…	Deletes the file or folder. Not for the project folder itself.
Reveal in Finder, Show in Explorer or Show in File Manager	Shows the file in the file manager of the system.

Renaming a file does not change the descriptors or the other files that name it.

4.1.6 The project file

The file `.karpfenproject` in the folder holds the settings of the project. The IDE writes it when one of its values changes, so a folder that was never changed has none. It is plain text with one `key = value` per line. A line that starts with `#` is a comment. The file can be edited by hand and kept in version control.

```
# karpfen project settings. karpfen IDE reads and rewrites this file.
theme = dark
zoom = 110
editorFontSize = 14
pane.runtime = shown 400
open = lamp.kstates
open = lamp.kmeta
active = lamp.kstates
bridge = http.kbridge.json
capture = on
feature.parallel = on
feature.trueParallel = on
```

Key	Value
`theme`	`light`, `dark` or `system`.
`zoom`	The scale of the whole window in percent, from 70 to 200.
`editorFontSize`	The font size of the editors, from 8 to 40. The default is 13.
`pane.project`, `pane.outline`, `pane.problems`, `pane.output`, `pane.runtime`	`shown` or `hidden`, then the size of the tool window. The default sizes are 260, 240, 200, 180 and 400. `pane.output` is the Language Tools Output. `pane.runtime` may end in `detached` when the Runtime tool window has a window of its own, see 4.5.4 The Runtime tool window.
`open`	An open file, relative to the folder. One line per tab, in the order of the tabs.
`active`	The file of the active tab.
`bridge`	A bridge that was ticked in the last run dialog. One line per bridge.
`capture`	`on` when Capture run to file was ticked in the last run dialog. Without the line it is off.
`feature.<id>`	`on` for a language feature or a feature of the project that is switched on, see 4.1.7 Language features. Without the line the feature is off.

A value that makes no sense is dropped, and the default holds. A line with a key the IDE does not know is kept when the IDE writes the file.

4.1.7 Language features

Five parts of the state machine language are optional. Each project switches them on by hand, and all are off in a new project.

Feature	Id in `.karpfenproject`	Reference
Histories	`history`	2.3.7 Histories
Tick limits	`tickLimit`	2.3.8 Tick limits
Breakpoints	`breakpoints`	2.3.9 Breakpoints
Macros	`macros`	2.3.6 Macros
Splits and joins	`parallel`	2.3.4.3 Splits and joins

Run > Language Features has one entry per feature. It says what a click does, such as `Switch On Histories` or `Switch Off Histories`. The IDE writes the change into `.karpfenproject` as a line such as `feature.history = on`, and an edit of that line by hand has the same effect.

While a feature is off, every use of it in a state machine is the error `kstates.feature-off`, and a run is blocked. Completion and the menus of the diagrams do not propose the feature. A run switches on the runtime features that the project needs: History for histories, TickLimit for tick limits and Breakpoint for breakpoints, see 3.2 Runtime features. With breakpoints on, a run has a debugger, see 4.6 Debugging.

Below the language features, the submenu has the features of the project. They are no part of the language: they change nothing in the texts, in the checks or in completion, and only switch a feature of the runtime on in every run.

Feature	Id in `.karpfenproject`	Needs	Effect
Parallel branches on threads	`trueParallel`	Splits and joins	Every run switches on the runtime feature TrueParallel, which steps the branches of a split at the same time, each on a thread of its own, see 3.2.9 TrueParallel.

Its entry reads `Switch On Parallel branches on threads` or `Switch Off Parallel branches on threads`. While splits and joins are off, the entry is disabled. Switching splits and joins off switches it off as well, and a line `feature.trueParallel = on` in a project file without `feature.parallel = on` is dropped when the IDE reads the file. With the feature on, the order of the steps within one tick can differ from run to run, so leave it off when a run must repeat exactly.

4.1.8 Settings and logs outside the project

Only the recent folders, the place and size of the window and the place and size of the detached Runtime tool window are kept outside the project, in `~/.karpfen-ide/settings.json`. The log of the IDE is `~/.karpfen-ide/logs/ide.log`, and the log of the language tools is `~/.karpfen-ide/logs/dsl-tools.log`. The environment variable `KARPFEN_IDE_HOME` names another folder for both.

4.2 Text editor

Every karpfen file and every environment descriptor opens in the text editor, one tab per file. The editor checks the text while you type and knows the names of the project, so it completes them, explains them and leads to their declarations. Keys are written for macOS first and for Windows and Linux second, as in Cmd+S / Ctrl+S.

4.2.1 Colours and marks

The editor colours keywords, the action operations such as `SET` and `DROPOBJ`, strings, numbers, operators and comments. The names of attachable state machines, after `STATEMACHINE`, `behaves after`, `ATTACH` and as the second argument of `EXCHANGE`, have a colour of their own. Inside an `EVAL` block it colours the Python code on its own, with its keywords, strings, numbers and `#` comments, and marks the model paths `$(...)` and the macro references `@name`. A descriptor is coloured as JSON.

Errors and warnings get a wavy underline, infos a dotted one. The gutter shows the line numbers and a marker on each line with a problem. Resting the mouse on an underlined place shows the problem. Brackets and quotes are closed as you type them, and the bracket that matches the one at the caret is marked.

File	What completion proposes

4.2.2 Completion

Completion opens by itself after two letters of a word, after an opening quote, after `$(` and after `->`. Complete Code opens it by hand: Ctrl+M on every system, with the Control key also on macOS, and on Windows also Ctrl+Space. The arrows and Page Up and Page Down choose an item, Enter, Tab or a double-click takes it, and Escape closes the list.

File	What completion proposes
KMeta	The `type` block at the top level. `prop`, `has`, `knows` and `list` in a type. `number`, `string` and `boolean` as the type of a property. The types of the file at the type of a relation, and in an event metamodel also the domain types. `behaves after "..."` after the description of a type, and the attachable state machines of the context inside its string, those attachable to the type first.
KModel	The `make object` block before the root object. `prop`, `has` and `knows` in an object. The properties and relations of the object's type inside `prop("`, `has("` and `knows("`. The types at `make object "key":"`, with the target type of the relation first. The keys of the objects of a fitting type after `knows("relation") -> "`.
KStates	The keywords that fit the place, the action operations, `IF IN SCOPE` and `WITH .. . AS`. The states inside the strings of a transition. The domain types after `ATTACHED TO "` and `ATTACHABLE TO "`. After `STATEMACHINE` the attached and the attachable header. `ATTACH "..."` after the parentheses of `SETOBJ` and `APPENDOBJ`. The attachable state machines of the context after `ATTACH "` and in the second argument of `EXCHANGE(`, those attachable to the type at that place first. The paths of the model in the strings of actions, `VALUE`, `IF IN SCOPE` and macro arguments, and after `$(` in `EVAL`. The events in the second argument of `EVENT(`, and the event domains used in the file. The macros inside `MACRO("` and after `@`. `CONTAINER STATE "..." USE "..."` among the states and in a state, `FINAL STATE "..."` among the states, `STATEMACHINE "..." FRAGMENT` as a header and `DONE` in a condition. The fragments of the project after `USE "`, those of the context first, without the fragment of the file itself. The constructs of an optional language feature only while the feature is on.
Descriptor	In an empty file the descriptor with its required keys. The keys an object does not have yet, with their documentation. The files that fit a key, those next to the descriptor first. The object keys of the model at `attachedTo`, the fitting ones first. The event domains of the state machines. Typical values of `tickDelayMs` and `eventTtlMs`.

4.2.3 Hover

Resting the mouse on a name shows what it is, and the problems at that place.

File	What hover shows
KMeta	The type, or a property with its type, such as `prop maxWatts: number`. The state machine after `behaves after` with its types.
KModel	The type of an object, a property or relation with its type, or an object with its key and type.

File	What hover shows
KStates	A state with its path, such as `state on > full`, a macro with its signature, the type a path leads to, or the type or event a name stands for. An attachable state machine with its types, such as `state machine "glowing" attachable to Bulb`. The fragment after `USE` with its file, such as `fragment "bulb_test" in bulb_test.frag.kstates`.
Descriptor	What a key means, the role of a named file, the object behind `attachedTo` with its type, and an event domain.

4.2.4 Go to definition

Go to Definition, Cmd+B / Ctrl+B or Cmd+click / Ctrl+click, leads from a name to its declaration, also in another file.

From	To
The type after `ATTACHED TO` in a state machine	The type in the metamodel.
A state name in a transition	The state. A state leads back to the places that name it.
A macro name	The macro.
The name of a state machine after `behaves after`, `ATTACH` or in `EXCHANGE(`	The attachable state machine, in its own file.
The name of a fragment after `USE`	The header of the fragment, in its own file. A fragment of the context of the file wins over one of another context.
An event name in `EVENT(`	The event type in the event metamodel.
A segment of a model path	The property or relation in the metamodel.
A type or a property or relation key in a model	Its declaration in the metamodel.
The target key of `knows` in a model	The object with that key.
A file path in a descriptor	The file.
The value of `attachedTo` in a descriptor	The object in the model.

The IDE has no search for the places that use a name.

4.2.5 Outline

The Outline tool window shows what the active file is made of, one row per element as `name : detail`, nested like the text. For a metamodel these are the types with their properties and relations, for a model the objects with their statements, and for a state machine the states with their entry and do blocks and inner states, the transitions, and the macros. An attachable state machine shows its name with its types as the detail. A fragment shows its header as a row of its own, `bulb_test : FRAGMENT`, above its states. A container state shows the fragment it uses as its detail, `self test : USE bulb_test`, and a final state reads `passed : FINAL`. A click on a row selects the element in the text.

4.2.6 Problems

The status bar shows `Problems` at its left end. While the project has errors or warnings it reads `Problems: N` with the icon of the worst severity, and a click shows or hides the Problems tool window. The window hides itself when no problem is left.

The Problems tool window has two groups. Local holds what the editor finds while you type. Global holds what only a validation of the whole project finds, see 4.4 Validating. Before the first validation it reads `Global (not validated yet)`. Below each group the problems are sorted by file, each file as `path (N)` and each problem as `message (line:col)`. The chips `Errors (N)`, `Warnings (N)` and `Infos` above the list show and hide each severity. A double-click or Enter on a problem leads to its place in the text.

The counts in the status bar are those of the group Local.

4.2.7 Find and replace

Find… (Cmd+F / Ctrl+F) opens the find bar. Its toggles are `Aa` for Match Case, `W` for Whole Words and `.*` for Regular Expression. Enter or the next button goes to the next match, Shift+Enter or the previous button to the one before, and Escape closes the bar. Replace… (Cmd+R / Ctrl+R) opens the bar with a second field and the buttons Replace and Replace All.

Go to Line… (Cmd+L / Ctrl+G) opens a small field that shows `Current line X of Y`. Enter or Go moves the caret.

4.2.8 Saving and changes outside the IDE

File > Save (Cmd+S / Ctrl+S) saves the active file, Save All (Cmd+Option+S / Ctrl+Alt+S) every file. A tab with unsaved changes asks before it closes, with Save, Don't Save and Cancel. A run saves every open file first.

When a file changes on disk while it is open without unsaved changes, the editor loads the new text as one change that Undo takes back. When the file has unsaved changes, a banner above the text reads `<file> changed on disk while it has unsaved changes.`, with Reload and Keep mine. A file deleted on disk gets the banner `<file> was deleted on disk.`, with Close and Keep mine. Saving over a file that changed on disk asks first, with Overwrite and Cancel.

4.2.9 Editing a descriptor

A descriptor is edited as JSON text with the help of completion, hover and go to definition from the tables above. The editor checks the keys and the kinds of their values and that every named file exists and is of the right kind, see 4.1.2 Environment descriptors. A bridge file opens in a form instead, see 4.7 Bridges.

4.2.10 Keys

Edit > Complete Code, Toggle Comment and Duplicate Line and Navigate > Go to Definition are in the menus as well. The context menu of the editor has Cut, Copy, Paste, Go to Definition and Toggle Comment.

Action	macOS	Windows and Linux
Complete Code	Ctrl+M	Ctrl+M, on Windows also Ctrl+Space
Go to Definition	Cmd+B or Cmd+click	Ctrl+B or Ctrl+click
Find, Replace	Cmd+F, Cmd+R	Ctrl+F, Ctrl+R
Find Next, Find Previous	Cmd+G, Cmd+Shift+G	F3, Shift+F3
Go to Line	Cmd+L	Ctrl+G
Toggle Comment	Cmd+/	Ctrl+/
Duplicate Line	Cmd+D	Ctrl+D
Delete Line	Cmd+Backspace	Ctrl+Y
Move Line Up, Move Line Down	Option+Shift+Up, Option+Shift+Down	Alt+Shift+Up, Alt+Shift+Down
Indent, Outdent	Tab, Shift+Tab	Tab, Shift+Tab
Word left, word right	Option+Left, Option+Right	Ctrl+Left, Ctrl+Right
Start of line, end of line	Home, End, Cmd+Left, Cmd+Right	Home, End
Start of file, end of file	Cmd+Home, Cmd+End, Cmd+Up, Cmd+Down	Ctrl+Home, Ctrl+End
Delete the word before, after	Option+Backspace, Option+Delete	Ctrl+Backspace, Ctrl+Delete
Select All	Cmd+A	Ctrl+A
Cut, Copy, Paste	Cmd+X, Cmd+C, Cmd+V	Ctrl+X, Ctrl+C, Ctrl+V
Undo, Redo	Cmd+Z, Cmd+Shift+Z	Ctrl+Z, Ctrl+Shift+Z
Save, Save All	Cmd+S, Cmd+Option+S	Ctrl+S, Ctrl+Alt+S
Close Tab	Cmd+W	Ctrl+W
Open Folder	Cmd+O	Ctrl+O

4.3 Diagrams

Each of the three kinds of model file has a diagram. The diagram is drawn from the text, and every change made in the diagram is an edit of the text. Undo in the editor takes such an edit back.

File	Diagram
`.kstates`, also a fragment	Statechart: the states with their inner states, the transitions, splits, joins and timeouts.
`.kmeta`, also an event metamodel	Class diagram: one box per type with its properties, and the relations as lines.
`.kmodel`	Object diagram: one box per object with its values, and the `has` and `knows` relations as lines.

4.3.1 Views

The bar below the editor has the buttons Text, Split and Diagram for a file with a diagram. Text shows only the text, Split shows the text and the diagram side by side, and Diagram shows only the diagram. The choice is not stored in the project file.

While the text has errors, the diagram shows the last text that could be read, with the note `The text has errors. This is the diagram of the last text that could be read.` A file that has nothing to draw yet shows a sentence such as `No state machine to draw yet. The diagram appears once the text can be read.` instead of a diagram.

4.3.1.1 Moving around

Drag the empty canvas or scroll to move the drawing. Ctrl+scroll or Cmd+scroll zooms. The buttons in a corner of the canvas are a minus, the zoom in percent, a plus and Fit. A click on the percent sets it back to 100%, and Fit shows the whole diagram. The zoom runs from 20% to 300%. A new diagram is fitted to its pane.

A click on a box selects its name in the text.

4.3.2 Reading a statechart

Element	Drawing
Header	A title line above the drawing, in its middle. An attached state machine reads `ATTACHED TO Lamp`, an attachable one `glowing ATTACHABLE TO Lamp, Bulb`, a fragment `bulb_test FRAGMENT`.
State	A box with round corners. A state with inner states shows its name in a row at its top, a state without inner states in its middle.
Container state	A box like a state without inner states. Below its name a row shows a page mark, a small sheet with a folded corner, and the name of the fragment the state uses. The inner states are those of the fragment, drawn in the diagram of the fragment.
Final state	A box like a state, with a ring around a dot in its upper right corner.
Initial state	A filled dot above the state.
Transition	A line from the source state to the target state, ending in an arrowhead.
Caption of a transition	The priority, which is the place of the transition in the text, and a label made from the condition, as in `1: switch/press`. A transition without a condition shows the priority only.
Split, join	A filled bar. The lines from the source states lead into the bar without an arrowhead, and the lines from the bar to the target states end in arrowheads.
Timeout	A dashed line from the state to the state its timeout leads to, with the caption `after 10 ticks` or `after 1 tick`.
Shallow history	A small circle with `H` behind the name of the state.
Deep history	A small circle with `H*` behind the name of the state.
Breakpoint at entry, at exit	A dot in the error colour on the left border of the state for its entry, on the right border for its exit.

Element	Drawing
Breakpoint at start, at end	A dot in the error colour in front of the caption of the transition for its start, behind it for its end.
Code	A small circle with a plus. In a state it is followed by the word `limit` for the tick limit, `entry` for the entry actions or `do` for the do actions. On a transition it stands at the right end of the caption and holds the condition.

The label of a condition is made of its clauses. `EVENT("switch", "press")` reads `switch/press`, `VALUE("p")` reads `p`, `EVAL { return x > 1 }` reads `[x > 1]` and `DONE` reads `DONE`. Several clauses are joined by `&`. A label longer than 40 characters is cut and ends with `...`. The mark `NOT LOOPING` is not drawn.

The title line shows the header of the text and changes with it. It cannot be edited, moved or selected in the diagram: the header is written in the text. The line always stands above the topmost element, so it moves up when a state is dragged above it, and it is not stored in the `.vis` file, see 4.3.8 Arranging by hand.

4.3.3 Reading a class or object diagram

Element	Drawing
Type	A box with the name of the type at the top and below a line one row per property, as `watts: number`. A type that behaves after an attachable state machine shows it in its first row, as `behaves after glowing`.
Object	A box with the title `key : Type`, underlined, at the top and below a line one row per property value, as `watts = 0`. Long values are cut with `...`.
`has` relation	A line with a filled diamond at the box that has the other.
`knows` relation	A line that ends in an open arrowhead at the box that is known.
Label of a relation	The name of the relation, with `[*]` behind it for a list.

A relation to a type the file does not declare, or a `knows` to a key the model does not have, is shown as a row of its box. The class diagram shows the types of its own file only.

4.3.4 Editing a statechart

A right-click on a state, on a line or its caption, on a timeout or on the empty canvas opens a menu of the changes that fit there. Where a change needs one or more states, a prompt such as `Click the state the transition leads to. Right-click to cancel.` appears, the states to choose from are marked, and you click one. A right-click or a click beside the states cancels. Rename and the other entries that need a name open a field at the pointer, filled with the current value.

Entries that write a construct of an optional language feature show only while the feature is on, see 4.1.7 Language features. An entry that removes such a construct always shows.

The examples below start from this state machine.

```
STATEMACHINE ATTACHED TO "Lamp" {
```

```
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            STATE "full" { }
            STATE "dimmed" { }
        }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

4.3.4.1 On the canvas

Entry	Shown	Effect on the text
Add state	Always	Adds `STATE "state" { }` at the end of `STATES`, drawn where you clicked. The name is `state`, or `state 2` and so on if the name is taken.
Add initial state	When no state of the file is initial	Adds `INITIAL STATE "state" { }` in the same way. A file has one initial state, at any depth, see 2.3.2.2 The initial state.
Add container state	The project has a fragment the file can use	Shows the fragments of the project as a second menu and adds `CONTAINER STATE "name" USE "name" { }` for the one you choose, named after the fragment. A fragment does not offer itself.
Add final state	In a fragment	Adds `FINAL STATE "done" { }` at the end of `STATES`, or `done 2` and so on if the name is taken.
Add transition	When there are states	Asks for the start, then the target, and adds a transition without a condition at the end of `TRANSITIONS`. A final state is never a start.
Add split	Splits and joins on, at least three states	Asks for the start and two targets, and adds `SPLIT "a" -> "b", "c" { }`. A final state is never the start.
Add join	Splits and joins on, at least three states	Asks for two starts and the target, and adds `JOIN "a", "b" -> "c" { }`. A final state is never a start.
Edit macros	Macros on	Opens the inside of the `MACROS` block in a code cell. Code saved into an empty cell creates the block.

Add state on the canvas of the example adds one line:

```
    STATES {
```

```
        INITIAL STATE "off" { }
        STATE "on" {
            STATE "full" { }
            STATE "dimmed" { }
        }
        STATE "state" { }
    }
```

Add transition from `full` to `dimmed` adds one line at the end of the transitions:

```
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "off" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "full" -> "dimmed" { }
    }
```

4.3.4.2 On a state

Entry	Shown	Effect on the text
Open page NAME	On a container state that names a fragment	Opens the diagram of the fragment, see 4.3.4.5 Pages. It changes nothing.
Rename	Always	Renames the state in its declaration, in every transition and in every timeout that names it. A `"` in the new name becomes `'`.
Edit as text	Always	Opens the whole state in a code cell titled `State <name>`.
Use another fragment	On a container state, when the project has another fragment	Shows the other fragments as a second menu and writes the one you choose after `USE`.
Add inner state	Not on a container or final state	Adds `STATE "state" { }` at the end of the state's body.
Add inner container state	Not on a container or final state, when the project has a fragment the file can use	Shows the fragments as a second menu and adds a container state that uses the one you choose at the end of the state's body.
Add transition from here	Not on a final state	Asks for the target, and adds a transition without a condition.
Add entry actions	The state has no `ENTRY`, not on a container or final state	Adds `ENTRY { }` after the breakpoints and the tick limit of the state.
Add do actions	The state has no `DO`, not on a container or final state	Adds `DO { }` after the entry actions.
Make initial state	The state is not initial, not on a final state	Writes `INITIAL` in front of the state and takes it from the state that had it, wherever that state stands in the file.

Entry	Shown	Effect on the text
Remember the last inner state (shallow history)	Histories on, the state has inner states or is a container state	Writes `SHALLOW HISTORY` in front of `STATE`, or of `CONTAINER STATE`.
Remember the last innermost state (deep history)	Histories on, the state has inner states or is a container state	Writes `DEEP HISTORY` in the same place.
Remove history	The state has a history	Takes the history words out.
Add tick limit	Tick limits on, the state has none, not on a final state	Adds `TICK LIMIT { DELAY: 1 TICK }` after the breakpoints of the state.
Add breakpoint at entry, Remove breakpoint at entry	Add needs breakpoints on	Writes or changes `BREAKPOINT { ENTRY }` as the first line of the body. A block without words is taken out.
Add breakpoint at exit, Remove breakpoint at exit	Add needs breakpoints on	The same with `EXIT`.
Make outer state	The state is an inner state	Moves the state with its inner states to the top level.
Make inner state of	Another state can take it, not on a final state	Asks for the new outer state and moves the state with its inner states into it. A container state or a final state never takes a state.
Delete state	Always	Deletes the state, its inner states and every transition that names one of them.

The empty room inside a state with inner states belongs to that state.

Add entry actions on `off`:

```
INITIAL STATE "off" {
    ENTRY { }
}
```

Add breakpoint at entry on `on`, with breakpoints switched on:

```
STATE "on" {
    BREAKPOINT { ENTRY }
    STATE "full" { }
    STATE "dimmed" { }
}
```

Remember the last inner state on `on`, with histories switched on:

```
SHALLOW HISTORY STATE "on" {
    STATE "full" { }
    STATE "dimmed" { }
}
```

Make outer state on `dimmed`:

```
    STATES {
        INITIAL STATE "off" { }
        STATE "on" {
            STATE "full" { }
        }
        STATE "dimmed" { }
    }
```

Rename `off` to `dark` changes the declaration and both transitions:

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "dark" { }
        STATE "on" {
            STATE "full" { }
            STATE "dimmed" { }
        }
    }
    TRANSITIONS {
        TRANSITION "dark" -> "on" {
            CONDITION { EVENT("switch", "press") }
        }
        TRANSITION "on" -> "dark" {
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

4.3.4.3 On a transition

A transition is hit on its line or on its caption.

Entry	Shown	Effect on the text
Edit as text	Always	Opens the whole transition in a code cell titled `Transition <caption>`.
Wait until the fragment is done (DONE)	The transition or split leaves a container state, or a source of the join is one, and the condition has no `DONE`	Writes `DONE` as the first clause of the condition, or adds `CONDITION { DONE }` after the breakpoints.
Remove DONE	The condition has `DONE`	Takes `DONE` out. A condition that holds nothing else goes as a whole.
Add target	Splits and joins on, on a split	Asks for a state and adds it to the targets of the split, as `, "x"`.
Add source	Splits and joins on, on a join	Asks for a state and adds it to the sources of the join. A final state is never a source.
Move earlier, to priority N	Not on the first transition	Swaps the transition with the one before it in the text.
Move later, to priority N	Not on the last transition	Swaps the transition with the one after it.

Entry	Shown	Effect on the text
Mark as not looping, Allow looping	Always	Writes `NOT LOOPING` behind the states, or takes it out.
Add breakpoint at start, Remove breakpoint at start	Add needs breakpoints on	Writes or changes `BREAKPOINT { START }` as the first line of the body.
Add breakpoint at end, Remove breakpoint at end	Add needs breakpoints on	The same with `END`.
Delete transition	Always	Deletes the transition.

Mark as not looping on the first transition:

```
TRANSITION "off" -> "on" NOT LOOPING {
    CONDITION { EVENT("switch", "press") }
}
```

Add breakpoint at start on the first transition:

```
TRANSITION "off" -> "on" {
    BREAKPOINT { START }
    CONDITION { EVENT("switch", "press") }
}
```

4.3.4.4 On a timeout

A timeout stands in the tick limit of its state, as `TIMEOUT: AFTER 10 TICKS TRANSITION TO "off"`. It has no priority, no looping mark and no breakpoints.

Entry	Shown	Effect on the text
Edit tick limit as text	Always	Opens the inside of the tick limit in a code cell, the same cell its marker opens.
Lead to another state	Tick limits on	Asks for a state and changes the name after `TRANSITION TO`.
Remove timeout	Always	Takes the timeout out of the tick limit. A tick limit without a delay goes as a whole.

No menu entry adds a timeout. Write it into the tick limit through its code cell.

4.3.4.5 Pages

A container state runs a fragment, which is a state machine of its own file, as its inner states, see 2.3.12 Pages. The statechart draws each file on its own, so the inner states of a container state are a page of their own: the diagram of the fragment. How a container state differs from a state with inner states is described in 2.3.13 Compound states and container states.

Open page on a container state, or a double-click on it, opens the file of its fragment and shows its diagram, next to the text or alone, as the diagram it came from was shown. The fragment is found by its name, one of the context of the file first. When no file of the project declares the name, a dialog says so and nothing opens.

The diagram of a fragment has a bar above the drawing: `Used by:` and the names of the

files whose container states use the fragment. A click on a name opens that file and shows its diagram the same way. A fragment that no file uses reads `No state machine uses this fragment.`

The examples below start from the lamp of `examples/pages`. Add container state on its canvas, with `warm_up` chosen in the second menu, adds one line:

```
    STATES {
        INITIAL STATE "off" { }
        CONTAINER STATE "self test" USE "bulb_test" { }
        STATE "on" {
            CONTAINER STATE "warming" USE "warm_up" { }
            STATE "steady" { }
        }
        CONTAINER STATE "warm_up" USE "warm_up" { }
    }
```

Remove DONE on the transition from `self test` to `off`, whose condition holds only `DONE`, takes the condition out:

```
        TRANSITION "self test" -> "off" { }
```

Wait until the fragment is done on the transition from `wiring` to `failed` in `bulb_test.frag.kstates` writes `DONE` in front of the clause it has:

```
        TRANSITION "wiring" -> "failed" {
            CONDITION { DONE EVENT("service", "smash") }
        }
```

4.3.5 Editing a class diagram

The examples start from this metamodel.

```
type "Bulb" "A light bulb" {
    prop("watts", "number")
}
type "Lamp" "A lamp" {
    has("bulb", "Bulb")
}
```

4.3.5.1 On the canvas

Add type adds a type as the first type of the file, so the root type, which is the last one, stays the same. The name is `Type`, or `Type2` and so on if the name is taken.

```
type "Type" "" {
}
type "Bulb" "A light bulb" {
    prop("watts", "number")
}
type "Lamp" "A lamp" {
    has("bulb", "Bulb")
}
```

4.3.5.2 On a type

Entry	Effect on the text
Rename	Renames the type and every relation that leads to it.
Edit as text	Opens the whole type in a code cell.
Describe	Opens a field with the description, the second string of the type, and writes it back.
Behaves after	Asks for the name of an attachable state machine and adds `behaves after "name"` after the description, see 2.1.8 Behaviour of a type. Shown while the type has no such clause.
Add property	Asks for the name and adds `prop("name", "string")` at the end of the type.
Add has relation to	Asks for the target type and adds `has("bulb", "Bulb")`, named after the target with a lower-case first letter and a number if the name is taken.
Add knows relation to	The same with `knows`.
Make root type	Moves the type to the end of the file. Not shown on the last type.
Delete	Deletes the type and every relation that leads to it.

Add has relation to `Bulb` on `Lamp`, which has `bulb` already:

```
type "Lamp" "A lamp" {
    has("bulb", "Bulb")
    has("bulb2", "Bulb")
}
```

Rename `Bulb` to `Light`:

```
type "Light" "A light bulb" {
    prop("watts", "number")
}
type "Lamp" "A lamp" {
    has("bulb", "Light")
}
```

4.3.5.3 On a row or a line

A right-click on a row gives Edit as text, a one-line field with the statement behind the row, such as `prop("watts", "number")`, and Delete, which deletes the statement. The row `behaves after glowing` gives Behaves after, a field with the name of the state machine that changes the name in the clause, and Remove behaves after, which takes the clause out of the header of the type.

Behaves after `glowing` on `Bulb`:

```
type "Bulb" "A light bulb" behaves after "glowing" {
    prop("watts", "number")
}
``` A right-click on the line of a relation gives
Rename relation, Edit as text, a code cell titled `Relation <name>`, and Delete relation.

## 4.3.6 Editing an object diagram
```
```

````
The examples start from this model.

```kmodel
make object "lamp":"Lamp" {
    prop("mode") -> "off"
    has("bulb") -> make object "bulb":"Bulb" {
        prop("watts") -> "0"
    }
}
````

4.3.6.1 On an object

Entry	Effect on the text
Rename	Renames the key of the object and every `knows` that leads to it.
Edit as text	Opens the whole object in a code cell.
Add property	Asks for the name and adds `prop("name") -> ""` at the end of the object.
Add object	Asks for the relation, then for the type, and adds `has("relation") -> make object "relation":"Type" { }`. The key of the new object is the name of the relation, with a number if the key is taken.
Add reference to	Asks for the object and adds `knows("ref") -> "key"`. Shown when the model has another object.
Delete	Deletes the `has` statement that makes the object, the objects it holds and every `knows` that leads to one of them. Not shown on the outermost object.

Add object on `lamp`, with the relation `spare` and the type `Bulb`:

```
make object "lamp":"Lamp" {
    prop("mode") -> "off"
    has("bulb") -> make object "bulb":"Bulb" {
        prop("watts") -> "0"
    }
    has("spare") -> make object "spare":"Bulb" { }
}
```

Add reference to `lamp` on `bulb`:

```
make object "lamp":"Lamp" {
    prop("mode") -> "off"
    has("bulb") -> make object "bulb":"Bulb" {
        prop("watts") -> "0"
        knows("ref") -> "lamp"
    }
}
```

The type of a new object is typed in. The menu does not list the types of the metamodel.

4.3.6.2 On a row or a line

A row and a line have the same entries as in a class diagram. Deleting a `has` line deletes the object it makes. The empty canvas of an object diagram has no menu entries.
```

4.3.7 Code cells

Code that a diagram shows folded up opens in a code cell over the diagram: the tick limit, the entry actions and the do actions of a state, and the condition of a transition. A click on its marker opens it. Edit as text opens a whole state, transition, type, object or relation in the same kind of cell.

Cell	Title
Tick limit	`Tick limit of <state>`
Entry actions	`Entry actions of <state>`
Do actions	`Do actions of <state>`
Condition	`Condition of <sources> to <targets>`
Edit as text	`State <name>`, `Transition <caption>`, `Relation <name>` or the title of the box

The head of a cell shows its title and the buttons Save and Close. Cmd+Enter / Ctrl+Enter saves too. Drag the head to move the cell, and a side or a corner to size it. The next cell opens at the same size.

Saving writes the code between the braces of its block. A tick limit whose code is deleted is taken out of its state. A transition without a condition has an empty cell, and code saved into it becomes `CONDITION { ... }`. A condition whose code is deleted is taken out, and the transition keeps an empty body `{ }`. In a cell of Edit as text, saving an empty text deletes the construct.

A cell is the text editor on its part of the file, so an action block in a cell completes and checks `ATTACH` and `EXCHANGE` with the names of the attachable state machines like the editor. It completes with Ctrl+M, on Windows also Ctrl+Space, shows what a name is when the mouse rests on it, and underlines errors and warnings with the language features of the project. It shows only the problems of its own code. Go to definition does nothing in a cell. A cell closes without saving when the text of the file changes under it.

4.3.8 Arranging by hand

The IDE places every element of a diagram by itself. A drag changes that.

Drag	Effect
A box	Moves it.
The bar of a split or join	Moves the bar.
A side or a corner of a state with inner states	Sizes the state. It never gets smaller than its inner states need.
The caption or label of a line	Moves the caption. It keeps its distance to the line when the line moves.
A waypoint	Moves the waypoint and bends the line.

While you drag, a grid of dots shows, and the element moves from point to point. A right-click on a line gives Add waypoint, and a right-click on a waypoint gives Remove waypoint. The waypoints of a line show while the pointer is near the line. Each line between a state and the bar of a split or join bends on its own.

The first change by hand fixes every element where it is. From then on only new elements are placed by the IDE. The arrangement is kept next to the file, with `.vis` added to its name, as in `lamp.kstates.vis`. It is a JSON file that the IDE writes and that opens read-only. The text always wins: what the `.vis` file says about an element that the text no longer has is dropped. To go back to the automatic layout, delete the `.vis` file.

4.3.9 Renaming between diagram and text

A rename in the diagram renames in the text, and the element keeps its place. A rename of a state, a type or an object in the text is recognized as well, and the element keeps its place in the diagram. A rename stays in its file. Other files that use the name, such as a model that names a renamed type or a descriptor that names a renamed object, are not changed and must be updated by hand.

4.3.10 While a model runs

While a run is selected in the Runtime tool window and observation is on, the statechart of each running state machine marks its active states with a strong border, and the innermost active state is lightly filled. A fired transition is not marked.

When several objects run the state machine of the file, as the instances of a definition do, the statechart marks the states of one of them. While the statechart is shown, the bar below the editor then has the button row `Live:` with one button per instance, labelled with the key of its object and the definition, such as `gen_Bulb_0 (glowing)`. The button of the instance shown is selected, and a click on another shows that one. At first the statechart shows the instance the runtime announced first. When the chosen instance ends, it shows the first one left, and when none is left, it marks nothing. See 4.5.4.7 State machine instances.

While a run is paused at a breakpoint, the statechart marks the place in the warning colour: a dashed frame around the state, or the lines of the transition drawn wide with a dashed frame around its caption and its bar. It then shows the instance that halted, and the other buttons of the row take no clicks until the run goes on. See 4.6 Debugging.

A state machine composed of pages runs as one flat state machine, see 4.5.4.8 Pages. The statechart of its own file marks a container state as active while the run is inside it. The states of the fragment are marked in the statechart of the fragment's file, under the names written there. Each container state that uses the fragment is a use of its own, and so is each object that runs it. While the statechart of a fragment is shown, the row `Live:` names every use by the key of the object and the container state, such as `lamp: self test` or `lamp: self test/wiring` for a fragment that another fragment uses, and it shows the row also for a single use. At first the statechart shows the first use.

A pause inside a fragment shows in the statechart of the fragment, at the state or transition of the use that halted, and that use is shown while the run is paused. The statechart of the file around it marks the container state as active but shows no pause.

4.3.11 Limits

- Macros are edited as text only, in the cell of Edit macros.
- `ATTACH` and `EXCHANGE` are written in the cells of the entry and do actions. No menu entry adds them.
- No menu entry adds a timeout.
- A rename together with another change in one edit of the text is not recognized as a rename.
- A change of the arrangement cannot be undone, and there is no command that goes back to the automatic layout.
- Renaming or deleting a model file leaves its `.vis` file behind.
- A state whose name contains `/from:` or `/to:`, or ends in `#` and digits, can lose the waypoints of its lines.
- The badge of a history can overlap the code markers in a narrow state.
- The caption of a transition can cover another line.
- The choice of Text, Split or Diagram is not stored in the project file.
- A file with nothing to draw has no canvas, so its menu is not there.
- The diagram of a state machine does not show the states of the fragments its container states use. Each fragment has its own diagram.
- The bar of the users of a fragment names files by their names only, so two files of the same name in different folders read alike.

4.4 Validating

The IDE checks a project at two depths. While you type, it reads every file and checks it within its context. A validation checks the whole project on top of that, across the files of each context, which takes longer and therefore only happens when you ask for it or start a run.

4.4.1 Checks while you type

Once the text has not changed for 300 ms, the editor reports what the readers of the languages find: syntax errors, and the checks of each file against the files it depends on, such as a model against its metamodel or the types and events a state machine names. The editor also checks the environment descriptors and the contexts of the folder, see 4.1 Projects. These problems are underlined in the text and listed in the group Local of the Problems tool window, see 4.2.6 Problems.

4.4.2 What a validation adds

A validation reads the project again and runs the verifier of the language tools over every context. The verifier checks the state machines against the domain metamodel, the event metamodel and the model: whether the paths in actions and conditions exist on the type the state machine is attached to, whether an operation and a value fit what the path leads to, whether the `$(...)` references in `EVAL` code resolve, whether the macro calls fit their macros, whether an event and its fields are known in the domain where they are used, and whether every state can be reached. It gets every state machine of a context, the attached ones and the attachable ones, and checks the names of attachable state machines across the

files: that no two have the same name, and that the names after `behaves after`, `ATTACH` and in `EXCHANGE` exist and fit the type at that place, see 2.4.6.4 Attachment. A finding about a `behaves after` lies in the metamodel. While a state machine of the context cannot be read, no name is reported as unknown. Its codes start with `project.`, as in `project.unknown-path` or `project.unreachable-state`. 2.4 Checks lists every check with its code and severity.

The verifier also gets every fragment of a context and checks each composition of a state machine with the fragments its container states use, directly or through other fragments, see 2.3.12 Pages and 2.4.6.5 Pages. A fragment that the context does not hold is `project.unknown-fragment` at the name after `USE`, a name that is no fragment is `project.not-a-fragment`, and fragments that use each other are `project.fragment-cycle`. The other codes of compositions are `project.qualified-name-clash`, `project.conflicting-macros`, `project.unknown-macro` and the warning `project.macro-from-other-file`. A finding lies in the file whose text it concerns: a path in a state of a fragment that does not exist is reported in the fragment, also when the verifier found it through the state machine that uses it. The rules of container states, final states and `DONE` within one file are reported while you type, as `kstates.container-use` or `kstates.misplaced-done`.

4.4.3 Starting a validation

To	Do this
Validate the project in the window	Click Validate in the bar below the editor, or choose Run > Validate Project. While it runs, the button reads `Validating...`.
Validate from a terminal	Run `karpfen-ide validate <folder>`, see 4.8 Command line.

4.4.4 The dialog

The answer is the dialog Validate Project. At the top it gives a summary, such as `2 errors and 1 warning`, `No problems found.` or `No errors or warnings.` when only infos are left. Below it lists every finding of the project, the problems of the editor and those only the validation found, one row per finding as `<file>:<line> <message>`, in the colour of its severity. Errors come first, then warnings, then the rest, each sorted by file and line. The rows do not lead to their places. Close is the only button.

4.4.5 The group Global

After the dialog is closed, what only the validation found stays in the group Global of the Problems tool window. Before the first validation the group reads `Global (not validated yet)`. Validate, Run > Validate Project and every run fill it anew. It keeps its findings while you edit, until the next validation. These findings are not underlined in the text. A double-click on one leads to its place.

4.4.6 Validation before a run

Every run saves all open files and validates the project first. An error anywhere in the project blocks the run, also in a file that the descriptor does not name, such as a state machine that belongs to no context. The dialog then opens with the title `Run Blocked: the Project Has Errors`. Warnings and infos do not block a run and open no dialog.

After the validation, the run prepares the deployment and checks what only matters for a run, such as a state machine attached to a key that no object of the model has. Such a problem blocks the run too. The run is then `blocked`, and the Log pane lists the problem under `Findings that block the run`, see 4.5.6 A failed run.

4.5 Running

Run deploys an environment descriptor to a karpfen runtime and starts it. The IDE runs the runtime in a Docker container of its own and shows the run in the Runtime tool window: the model as it changes, the events, the active states and the log of the runtime. What the runtime does with a model is the subject of 3 Execution.

4.5.1 The runtime container

A model runs in the IDE's own Docker container and nowhere else. Docker Desktop or another local Docker engine must be running.

- Each open folder gets its own container. It listens on `127.0.0.1` only, on a free port.
- The first run builds the runtime image from the runtime that comes with the IDE. This takes a few minutes once.
- The image holds Python 3 with numpy, scipy, sympy, pandas, regex, python-dateutil and chardet for the `EVAL` blocks.
- The container starts with the first run and keeps running after the run stops, so the next run starts fast.
- Closing the folder or the IDE stops and removes the container. The image stays.

Run > Start Runtime starts the container without a run, and Run > Stop Runtime stops and removes it. The buttons Start runtime and Stop runtime in the Runtime tool window do the same. The containers and the image carry the label `io.karpfen.ide.runtime=true`, see 1.12 Closing and cleaning up.

To run a model on a runtime of your own, without the IDE, see 3.3 Running the runtime.

4.5.2 Starting a run

Run > Run (Shift+F10) starts a run. On a Mac keyboard you may have to hold Fn. The run goes through these steps.

1. It picks the descriptor: the one in the active editor, otherwise the only descriptor of the project, otherwise it asks in the dialog Run with the sentence `Choose the environment descriptor to run:`, with the last choice selected. A project without a descriptor gets an error that says how to create one.
2. It saves every open file, because it uploads the saved files.
3. It validates the whole project. An error blocks the run, see 4.4.6 Validation before a run.

4. It opens the run dialog.
5. It stops an earlier run of the same descriptor that still runs.
6. It prepares the deployment from the saved files and checks it.
7. It starts the container if it is not running and waits for the runtime.
8. It deploys the descriptor and starts the environment.

4.5.3 The run dialog

The dialog Run opens with every run, also with Run again.

In the dialog	Effect
One box per bridge file of the project	The ticked bridges start with the run, see 4.7 Bridges.
Capture run to file	The run writes the whole log of the runtime to a file, see 4.5.8 Captured logs.
Run, Cancel	Starts the run, or ends it here.

The boxes ticked last time are ticked again. The choices are kept in `.karpfenproject`.

4.5.4 The Runtime tool window

The Runtime tool window opens with every run and then takes three quarters of the window's height. The status bar shows `Runtime: stopped`, `building image`, `starting`, `running`, `failed` or `stopping`, and a click on it shows the tool window.

The button at the right end of the tool window's title bar moves it into a window of its own, for example onto a second screen. Its place in the main window closes, and the editor gets the room. In the separate window the same button shows the dock icon and moves the tool window back to its place, with the height it had. Closing the separate window also docks the tool window. The separate window has the same menus as the main window, so Run, Stop and the keys of the debugger work there too. It follows the theme and the zoom of the main window, and it closes with the project and with the main window.

While the tool window is detached, a new run, a click on the runtime status in the status bar and View > Runtime show it in its own window and bring that window to the front. View > Runtime hides a shown tool window, which closes the separate window. Showing it again opens the separate window again, because the tool window stays detached until it is docked.

The runs, the selected run, observation and the log carry over between both places. The scroll positions and the widths of the Model, Events and Log panes start anew. Whether the tool window is detached is kept in `.karpfenproject` as `pane.runtime = shown 400 detached`, see 4.1.6 The project file, so a project that was closed with it detached opens it in its own window again. The place and size of the separate window are kept in `~/.karpfen-ide/settings.json`. A place on a screen that is no longer connected is dropped, and the window opens in the middle of the screen.

4.5.4.1 The runtime line

The first line says what the container does, with the buttons Start runtime and Stop runtime.

Text	Meaning
`Docker: not started`	No container runs.
`Docker: checking the Docker engine`	The IDE looks for the Docker engine.
`Docker: building the runtime image`	The first run builds the image. A detail of the build step may follow.
`Docker: creating the container`, `Docker: starting the container`	The container is being set up.
`Docker: waiting for the runtime`	The container runs, and the runtime in it starts.
`Docker: running at 127.0.0.1:<port>`	The runtime is ready.
`Docker: stopping`	The container is being stopped and removed.
`Docker: <problem>`	The container could not be started. The text is red and can be selected.

4.5.4.2 The runs

The second line lists the runs, each with its name, the key of its environment once known, and its phase. The name is the file name of the descriptor without `.kenv.json`. A click selects a run, and the panes below show the selected run.

Phase	Meaning
`preparing`	The run prepares the deployment from the saved files.
`blocked`	The preparation found errors. The run never reached the runtime.
`waiting for the runtime`	The run waits for the container.
`deploying`	The run uploads the files and configures the environment.
`running`	The environment runs.
`ready`	The environment is activated but not started.
`stopped`	The environment was stopped.
`failed`	The runtime, the deployment or the environment failed, or the environment ended on its own.

The buttons Run again, Stop and Clear finished follow. Run again starts the selected run anew, through the steps of 4.5.2 Starting a run. Stop stops it. Clear finished removes every run that is blocked, stopped or failed from the list.

With breakpoints switched on, a third line holds the debugger, see 4.6 Debugging.

Below these lines three panes show the selected run side by side: Model, Events and Log.

4.5.4.3 The Model pane

The Model pane shows the model of the running environment as a tree. Expand all and Collapse all open and close every row.

Row	Meaning
`relation: key : Type`	An object, under the relation that holds it.
`name = value`	A property and its value.
`relation -> key`	A `knows` reference. A click on the key selects that object.
`created at run time`	Marks an object that the state machines created.

The switch Observation decides how the tree follows the run. With observation on, the runtime reports every change, and the tree, the active states and the statecharts follow at once. With observation off, the tree stays as it is, and Pull reads the current model once. A run starts with observation on. The status next to the switch reads, for example, `Observation on, live at revision 12` or `Observation off, pulled at revision 30`. Other states are `predicted, nothing read yet`, `synchronizing`, and `final at revision N` after the environment stopped. Once a run has ended, the switch and Pull are greyed out, since the runtime no longer holds its environment.

4.5.4.4 The Events pane

The Events pane sends events into the run and lists the events of the environment.

1. Under 1. Domain, choose a domain. The domains are those the state machines use. A domain with an event that the event metamodel declares is in bold. A domain whose events no event metamodel declares is marked `(used only)`.
2. Under 2. Event, choose an event. Each reads as `name: PayloadType`, or with `(used only)` when the event metamodel does not declare it.
3. Under 3. Payload, edit the payload as JSON. It starts as a template of the payload type and is checked after every change. An event without a payload type reads `This event has no payload type. It is sent without a payload.`
4. Click `Send <event>`. The button is greyed out while the payload has an error or while events cannot be sent.

After a send, the line next to the button reads `Sent. It shows in the list below once the runtime has published it.` or `Not sent:` with the reason. A line below it says whether events can be sent now:

Text	Meaning
`Events cannot be sent yet, because the environment is not deployed.`	The deployment has not reached the environment.
`The environment has not started yet. An event sent now waits until it starts.`	The environment is activated but not started.
`The environment is running. An event is delivered at once.`	Events go in now.
`Events cannot be sent:` and a reason	The run has ended or the connection is lost.

The list `Events in this environment, newest first` shows the events sent from here, from other clients and from bridges, and the events the state machines raise. Its columns are Time, Domain, Event, Source and Payload. The source is `external` for an event from outside and `internal` for an event a state machine raised. A click on an event shows its whole payload. An empty list reads `No events yet. Events you send and events the state machines raise appear here.`

An event waits until a state machine takes it or until its time to live runs out, see `eventTtlMs` in 4.1.2 Environment descriptors and 3.1 Execution semantics.

4.5.4.5 The Log pane

The Log pane has the switch Simplified, which is on at first, Follow, which keeps the newest line in sight, and Clear. Above the log it shows what the run has to tell:

- `The runtime could not be started:` with the problem of the container.
- `Captured to karpfen-runs/<file>` when the run is captured, or `The run could not be captured to a file:` with the reason.
- `Findings that block the run` with the problems of the preparation. A click on one leads to its place.
- The active states of each state machine instance, as `lamp: on > full`, outermost state first. An instance of a definition also names its definition, as `gen_Bulb_0 (glowing): lit`, see 4.5.4.7 State machine instances.
- One line about the deployment: `Deployment done, 16 steps`, `Deploying: <step>` or `Deployment failed at: <step>`. A click on it shows the details: the environment, the connection, every step, and the problems of the run.
- One line per bridge of the run, see 4.7.6 The bridges of a run.

All of this text can be selected and copied.

With Simplified on, the log shows only the steps of the state machines and the errors, one line each with the time, the object or `engine`, and the text:

```
14:17:03.887  lamp        attaches its state machine
14:17:03.888  lamp        starts in off
14:17:03.889  wallSwitch  attaches its state machine
14:17:03.890  wallSwitch  starts in down
14:17:03.892  engine      Engine started with 2 state machine(s), tickDelay=500ms
14:17:03.895  lamp        runs the entry actions of off
14:17:04.896  lamp        off -> full
14:17:04.898  wallSwitch  down -> up
14:17:05.397  lamp        runs the entry actions of on
14:17:05.399  lamp        runs the entry actions of full
```

The times are those of the runtime's clock, which is UTC in the container.

A state machine that splits runs on several paths at once until they join again, see 2.3.4.3 Splits and joins. The runtime numbers the paths of each object. While an object runs on more than one path, each of its lines begins with the path it belongs to, and the split and the join name the paths they start and end:

```
14:17:04.896  lamp        split off -> test bulb, test switch into paths 1, 2
14:17:04.898  lamp        path 1: runs the entry actions of test bulb
14:17:04.899  lamp        path 2: test switch -> check position
14:17:05.397  lamp        join bulb ok, switch ok -> on from paths 1, 2
14:17:05.399  lamp        runs the entry actions of on
```

An object that runs on one path shows no path, so a run without splits never shows one.

An empty simplified log reads `No step of a state machine yet. Switch Simplified off to see the whole output of the runtime.` With Simplified off, the log shows every line of the runtime's output, with its error output in red.

4.5.4.6 The deployment steps

The details of the deployment list each step with its state, `pending`, `running`, `done`, `skipped` or `failed`, and a detail. These are the steps of the lamp.

```
Checking the runtime at http://127.0.0.1:65457: done, The runtime is up and supports the
    ↪ IDE
Creating the environment: done, env-1794579423685
Connecting to the environment: done, Connected as karpfen-ide-254666e10161
Uploading lamp.kmeta: done
Uploading lamp-events.kmeta: done
Uploading lamp.kmodel: done
Attaching lamp.kstates to lamp: done
Attaching switch.kstates to wallSwitch: done
Setting the tick delay to 500 ms: done
Setting the event time to live to 5000 ms: done
Listening to 2 event domains: done, service, switch
Switching on runtime features: skipped, The project switches on no language feature that
    ↪ needs one
Activating the environment: done
Switching observation on: done, The runtime read the model as analyzed
Starting the environment: done
Confirming the start: done, Every state machine reported its state
```

A descriptor with definitions has a step `Uploading the definition glowing.kstates` for each, among the attaching steps in the order of the descriptor. Confirming the start waits for every instance the start built, also those of the definitions.

A descriptor with fragments has a step `Uploading the fragment bulb_test.frag.kstates` for each, also in the order of the descriptor, see 4.5.4.8 Pages.

With histories and tick limits switched on, the step reads `Switching on History, TickLimit`, see 4.1.7 Language features. With Parallel branches on threads switched on as well, it reads `Switching on History, TickLimit, TrueParallel`. A runtime that does not know TrueParallel fails the step with `The runtime at <address> does not know the feature TrueParallel, so it is older than the IDE.`

4.5.4.7 State machine instances

A run has one state machine instance for each object that runs a state machine. The descriptor attaches state machines to objects by their keys and lists attachable state machines as definitions, see 4.1.2.3 Definitions. The runtime gives objects instances of definitions and takes them away while the run goes on, as 3.1.13 State machine management and lifecycle describes:

- At the start, each object whose type behaves after a definition gets an instance of it.
- An object that a rule creates gets an instance of the definition its type behaves after, or of the one its `ATTACH` names.
- `EXCHANGE` ends the instance of an object and gives the object an instance of another definition, which starts in its initial state.
- An object that leaves the model takes its instance with it.

The Runtime tool window follows the instances. The active states above the log name the object, and the definition in parentheses for an instance of a definition. A new instance shows the initial state of its definition until its first step, and an instance that ends disappears with

its states. The diagram of a definition shows one instance at a time, see 4.3.10 While a model runs. With Simplified on, the log says when an object gets or loses an instance:

```
14:20:11.000  bulb        attaches glowing
14:20:11.001  bulb        starts in lit
14:20:16.500  bulb        exchanges glowing for flickering
14:20:16.501  bulb        starts in on
14:20:18.500  gen_Bulb_0  attaches glowing
14:20:18.501  gen_Bulb_0  starts in lit
14:20:19.000  bulb        detaches flickering
```

At each start the log has an `attaches` line for every instance, and a state machine the descriptor attached reads `attaches its state machine`, or `exchanges its state machine for flickering` when an `EXCHANGE` ends it. The initial state of an instance follows right after the line that gives the object the instance.

The preparation checks the definitions against the descriptor before anything reaches the runtime:

- A type that behaves after a state machine that the descriptor does not list blocks the run with `deploy-definition-not-listed`, because the runtime would refuse to activate the environment.
- An `ATTACH` or `EXCHANGE` that names a state machine the descriptor does not list gives the warning `deploy-attached-definition-not-listed` at the state machine. The runtime fails such an action when it runs and writes the failure to its log. A name that an `EVAL` or a `MACRO` computes is not checked.

The example in `examples/spares` shows all of this. Each bulb runs `glowing` from the start, the event `stock` of the domain `service` puts a new bulb with `glowing` on the shelf, `flicker` exchanges the state machine of the bulb in the lamp for `flickering`, and `replace` screws in a new bulb, so the old one leaves with its instance.

4.5.4.8 Pages

A state machine composed of pages has container states that run fragments, see 2.3.12 Pages. The descriptor lists every fragment a run needs, without `attachedTo`, also the fragments that only other fragments use, see 4.1.2.5 Fragments. The deployment uploads each of them as it stands, before it activates the environment. When the runtime activates the environment, it puts the fragments into the container states and runs one flat state machine for each object.

In the flat state machine the states of a fragment carry the name of their container state in front, separated by a slash. When the container state `self test` of the lamp in `examples/pages` uses `bulb_test`, whose container state `wiring` uses `wiring_check`, its states read `self test/wiring` and `self test/wiring/measuring`. The Runtime tool window shows these names everywhere it names states: in the active states above the log, as `lamp: self test > self test/wiring > self test/wiring/measuring`, in the simplified log, and in the debugger line. One fragment that two container states use runs as two copies, each with the name of its own container state.

The preparation checks the pages against the descriptor before anything reaches the runtime:

- A fragment that a state machine of the run uses, directly or through another fragment, and that the descriptor does not list blocks the run with `deploy-fragment-not-listed` at the descriptor. The problem names the fragment, its file and the file that uses it.

The project validation, which runs before, reports the same cause as `project.unknown-fragment`, see 4.4 Validating.

- A composition that does not flatten blocks the run with `deploy-not-flattened`, with the words of dsl-tools, at the state it concerns. Examples are a container state that uses an attachable state machine and a macro that two files of the composition define differently.

If the runtime refuses the activation nevertheless, the run fails with one problem per line of the refusal, with the runtime's own words, at the file of the state machine the line names.

How the statecharts show a run in pages is described in 4.3.10 While a model runs.

4.5.5 Stopping a run

Stop in the line of the runs, or Run > Stop (Cmd+F2 / Ctrl+F2), stops the selected run. The run turns `stopped`, the Model pane keeps the last model, and the Send button is greyed out. The container keeps running. A new run of the same descriptor stops the run before it.

A stop ends the run for good. The IDE reads the model a last time and then drops the environment, so the runtime forgets it and no longer lists it; Run again deploys a fresh environment. The runtime by itself would only freeze a stopped environment, as 3.1.13.9 Stop and start, forcing and clients describes. Every way a run ends drops its environment the same way: Stop, a new run that replaces the run before it, closing the project, Stop runtime, quitting the IDE, and the end of a run on the command line. After the drop the IDE asks the runtime nothing more about the environment, so the Model pane, the Events pane, the debugger and the bridges show what the run saw last.

A drop can fail, for example because the runtime is gone or is an older runtime that cannot drop environments. The run still ends as `stopped`, and the Log pane notes the problem `environment-left-behind` with the reason, such as `The runtime at http://127.0.0.1:65457 cannot drop environments, so it is older than the IDE; the environment env-1794579423685 stays there.` The IDE waits at most ten seconds for a drop.

When another client stops the environment of a run, the run also turns `stopped`. The Log pane notes `The runtime reports that the environment stopped.` and that the environment was stopped by somebody else than the session, so the run ended. The IDE then reads the model a last time and drops the environment as after its own stop.

When another client drops the environment of a run, the run turns `stopped` as well. The Log pane notes that the environment was dropped by somebody else than the session, so the run ended, and the Model pane keeps what the run saw last, since nothing of the environment is left to read.

When the runtime ends while a run is live, for example because its container was removed outside the IDE, the run turns `failed` with the problem `environment-lost`.

What happens	Where it shows

4.5.6 A failed run

A run can end before or after it reaches the runtime.

What happens	Where it shows
The project has errors.	The dialog `Run Blocked: the Project Has Errors`, see 4.4 Validating.
The preparation finds a problem, such as a state machine attached to a missing key.	The run is `blocked`, and the Log pane lists the problems under `Findings that block the run`.
Docker is not running.	The run is `failed`. The runtime line and the Log pane read `Cannot connect to the Docker engine at ...` and `Start Docker Desktop or the local Docker engine, then try again.`
The container stopped.	The runtime line names the container and asks to run again.
A step of the deployment fails.	The run is `failed`, and the Log pane reads `Deployment failed at: <step>`. The details list the problems.
The runtime refuses an uploaded file.	One problem per error of the file, at its line and column, as `lamp.kstates:13:1: <message>`. A click leads to the place.
The runtime refuses the form of a state machine.	One problem at the file, with the runtime's text, such as `A state machine without attachedTo must have the attachable form ....`
The runtime refuses to activate the environment, because a type behaves after a state machine it does not hold or that is not attachable to the type.	One problem at the descriptor with the runtime's text.
The runtime refuses to activate the environment, because a state machine does not flatten with the fragments it holds.	One problem per line of the runtime's text, at the file of the state machine it names, as `The runtime refused the activation, because lamp.kstates cannot be flattened: <message>.`
The environment ends on its own.	The run is `failed` with the problem `engine-halted`. The log of the runtime has the reason.

The problems of the preparation and the deployment have these codes.

Code	Severity	Meaning
`deploy-no-context`	Error	The descriptor could not be read. Its own problems say why.
`deploy-save-failed`	Error	A file could not be saved before the deployment.
`deploy-unsaved-file`	Error	A file was edited while the run prepared. Run again.

Code	Severity	Meaning
`deploy-context-errors`	Error	A file of the context has errors, one problem per file at its first error.
`deploy-no-metamodel`, `deploy-no-model`	Error	The context has no metamodel or no model, or it could not be read.
`deploy-unknown-attachment`	Error	A state machine is attached to a key that no object of the model has.
`deploy-definition-not-listed`	Error	A type behaves after a state machine that the descriptor does not list as a definition.
`deploy-attached-definition-not-listed`	Warning	An `ATTACH` or `EXCHANGE` names a state machine that the descriptor does not list as a definition.
`deploy-fragment-not-listed`	Error	A state machine of the run uses a fragment, directly or through another fragment, that the descriptor does not list.
`deploy-not-flattened`	Error	A state machine of the run does not flatten with the fragments the descriptor lists.
`deploy-duplicate-attachment`	Error	Two state machines are attached to the same key.
`deploy-attachment-type`	Warning	The attached object's type differs from the type after `ATTACHED TO`.
`deploy-reference-loop`	Error	An object is reachable from itself through `has` and `knows`, which the runtime cannot read.
`deploy-domain-is-attachment`	Warning	An event domain is named like an attached object key.
`deploy-tick-delay-range`, `deploy-tick-delay-zero`, `deploy-event-ttl-never-expires`	See 4.1.2.1 Keys	The timing of the descriptor.
`deploy-model-differs`	Warning	The model the runtime read back differs from the model the IDE read, one problem per object.
`runtime-unreachable`	Error	The runtime could not be reached.
`runtime-incompatible`	Error	The runtime does not answer like a karpfen runtime that supports the IDE.
`runtime-no-answer`	Error	The runtime did not answer in time, or the connection broke during a call.
`runtime-rejected`	Error	The runtime refused a call, with its reason. A refused upload gives one problem per error of the file.
`runtime-error`	Error	The runtime failed with an internal error.
`handshake-rejected`	Error	The runtime refused the WebSocket connection.
`connection-lost`	Error or warning	The connection could not be opened or was given up. A warning after the deployment.
`engine-halted`	Error	The environment ended on its own.

Code	Severity	Meaning
`engine-silent`	Warning	Not every state machine instance reported its state after the start.
`observation-out-of-sync`	Warning	Observation is on, but the model could not be read again. The tree stays as it was until the next Pull.
`environment-lost`	Error	The runtime no longer knows the environment, usually because it ended or restarted.
`environment-left-behind`	Info	An environment stays on the runtime, because dropping it failed or took too long.

4.5.7 Several runs

The list of runs keeps every run until Clear finished removes it. Runs of different descriptors can run at the same time in the same container. Each has its own environment and its own panes. A finished run keeps its panes after its environment was dropped, until Clear finished removes it.

4.5.8 Captured logs

With Capture run to file ticked, the run writes the whole log of the runtime into the folder `karpfen-runs` of the project, in a file named after the date and time of the start, such as `2026-11-13_14-32-05.log`. If the name is taken, `-2`, `-3` and so on are added. The folder does not show in the Project tool window. The Log pane names the file of the selected run.

The file starts with lines that begin with `#`: the descriptor, the start, the address of the runtime and, once known, the environment. Then every line of the runtime's log follows from the deployment to the end of the run, also the lines of each tick that the Log pane leaves out, each marked `stdout` or `stderr`.

The file holds the log of the whole runtime while its run lives. With two runs at once, each file also holds the lines of the other run. Lines that the runtime writes before the deployment are not in the file.

```
# karpfen run capture
# Descriptor: lamp.kenv.json
# Started: 2026-11-13 14:32:05
# Runtime: the IDE's Docker container at http://127.0.0.1:65457
# Environment: env-1794579423685
```

4.6 Debugging

A run halts at the breakpoints of its state machines when the project has the language feature Breakpoints switched on. The debugger then shows where the run halts and lets it go on, a breakpoint or a tick at a time.

4.6.1 Switching breakpoints on

Choose Run > Language Features > Switch On Breakpoints. The project file gets the line `feature.breakpoints = on`, see 4.1.7 Language features. While the feature is off, every `BREAKPOINT` block is an error that blocks the run, and a run has no debugger.

With the feature on, the run switches on the runtime feature Breakpoint, see 3.2 Runtime features.

4.6.2 Setting breakpoints

A breakpoint is a block of the text, see 2.3.9 Breakpoints. In a state, `BREAKPOINT { ENTRY EXIT }` halts before the state is entered, before it is left, or both. In a transition, split or join, `BREAKPOINT { START END }` halts before the transition changes the active states, after it, or both.

```
STATEMACHINE ATTACHED TO "Lamp" {
    STATES {
        INITIAL STATE "off" {
            BREAKPOINT { EXIT }
        }
        STATE "on" { }
    }
    TRANSITIONS {
        TRANSITION "off" -> "on" {
            BREAKPOINT { START }
            CONDITION { EVENT("switch", "press") }
        }
    }
}
```

In a statechart, the menu of a state has Add breakpoint at entry and Add breakpoint at exit, and the menu of a transition has Add breakpoint at start and Add breakpoint at end. Each has a Remove entry once the breakpoint is set, see 4.3.4 Editing a statechart. A breakpoint shows as a dot in the error colour: on the left border of a state for its entry, on the right border for its exit, in front of the caption of a transition for its start and behind it for its end.

4.6.3 The debugger line

A run with breakpoints switched on has a third line in the Runtime tool window, below the list of runs. It starts with `Debugger`.

The line reads	Buttons	Meaning
`Running`	Pause	The run goes on.
`Paused before the entry of on in lamp`, `Paused before the exit of off in lamp`	Resume, Step	The run halts at a breakpoint of a state. The line ends with the key of the object whose instance halted.
`Paused before the transition off to on in lamp`, `Paused after the transition off to on in lamp`	Resume, Step	The run halts at a breakpoint of a transition. The states of a split or join are joined by `and`.
`Paused between two ticks`	Resume, Step	The run halts after a Pause or after a Step that ended its tick.

Command	Key	Effect
Pause	none	The run finishes the tick it is in and then waits.
Resume	F9	The run goes on up to the next breakpoint.
Step	F8	The run goes on to the next breakpoint or to the end of the tick, whichever comes first. From between two ticks it runs one tick.

Pause, Resume and Step are also in the Run menu. They act on the selected run.

4.6.4 What a pause shows

While a run is paused, the open statechart of the state machine marks the place in the warning colour: a dashed frame around the state, or the lines of the transition drawn wide with a dashed frame around its caption and its bar, see 4.3.10 While a model runs. The debugger line names the place in words.

4.6.5 Breakpoints of definitions

A breakpoint in an attachable state machine halts every instance of it, since each object that runs the definition runs the same text, see 4.5.4.7 State machine instances. The debugger line names the object whose instance halted, as in `Paused before the entry of lit in gen_Bulb_0`, and the statechart of the definition shows that instance while the run is paused, whichever instance was chosen before. The runtime forgets what an instance left behind when it ends, so a later instance on the same object starts without it, see 3.1.13.3 When instances join and leave.

4.6.6 Breakpoints in pages

A breakpoint in a fragment halts every use of it, since each container state that uses the fragment runs a copy of its text, see 4.5.4.8 Pages. The debugger line names the state by its name in the flat state machine, as in `Paused before the entry of self test/wiring/measuring in lamp`. The statechart of the fragment shows the use that halted, with the place marked under the name the fragment gives it, see 4.3.10 While a model runs.

4.6.7 Limits

- A run without a window has no debugger. Its breakpoints never halt it, see 4.8 Command line.
- Pause has no key.
- A breakpoint takes effect with the next run, because it is part of the uploaded text.

4.7 Bridges

A bridge connects a running model to another system. An inbound route turns a message from outside into an event of the model. An outbound route sends a message whenever a property, the active states of a state machine or an event has something new. Bridges run inside the IDE, with a run, and stop with it.

Kind	Inbound messages arrive as	Outbound messages go out as
`http`	`POST` or `PUT` requests to a path of the bridge's own HTTP server	`POST` requests to a URL
`mqtt`	Messages on an MQTT topic of a broker	Messages published on an MQTT topic
`ros`	Messages on a ROS topic, through a rosbridge server	Messages published on a ROS topic

4.7.1 The bridge file

A bridge is a JSON file named `*.kbridge.json` in the project. The Project tool window lists the bridge files in a section of their own at the bottom.

Key	Value
`kind`	`http`, `mqtt` or `ros`.
`settings`	An object of texts with the settings of the kind, see 4.7.3 Settings.
`inbound`	The inbound routes, a list of objects.
`outbound`	The outbound routes, a list of objects.

The keys of an inbound route:

Key	Value
`from`	Where the message arrives: a path of the HTTP server such as `/press`, an MQTT topic filter, or a ROS topic.
`event`	The event it becomes, as domain and name with a slash between them, such as `switch/press`.
`payload`	Optional. An object that gives each field of the event's payload a value, see 4.7.2 From message to event and back.
`policy`, `limit`, `periodMs`	Optional. What happens when messages come too fast, see below.
`type`	ROS only. The type of the messages on the topic, such as `std_msgs/msg/Bool`.

The keys of an outbound route:

Key	Value
`property`	Follows a property, as object key and property with a slash between them, such as `bulb/watts`.
`state`	Follows the active states of the state machine attached to the object, such as `lamp`.
`event`	Follows an event that a state machine raises, as domain and name, such as `switch/press`.
`to`	Where the message goes: a URL for HTTP, a topic for MQTT and ROS.
`policy`, `limit`, `periodMs`	Optional, as for inbound routes.
`type`	ROS only. The type of the messages, `std_msgs/msg/String` if the route names none.

An outbound route has exactly one of `property`, `state` and `event`.

`policy`	Effect
`queue`	Keeps the messages in their order. When more than `limit` wait, the oldest one is dropped. `limit` is 64 unless the route sets it. This is the policy of a route without `policy`.
`latest`	Keeps only the newest message. A message that still waits when the next one arrives is dropped.
`sample`	Hands on at most one message per `periodMs` milliseconds, always the newest one. `periodMs` is required.

A route with a problem is left out, and the other routes of the file stay.

4.7.2 From message to event and back

An inbound route without `payload` hands the body of the message on. A body that is a JSON object becomes the payload of the event. Any other body, also an empty one, gives an event without a payload. With `"payload": {}` the event never has a payload.

An inbound route with `payload` builds the payload field by field. A value that starts with `$` is a path into the body, and any other value is taken as it is. A path is `$` for the whole body, `$.a.b` for a member, and `$.a[0]` for an item of a list. With a path, the body must be JSON and the path must lead to a value. Otherwise the message is refused and counted as failed, with a reason such as `The message is not JSON.`

```
{ "from": "/dim", "event": "switch/dim", "payload": { "bright": "$.bright" } }
```

An outbound route sends JSON. A property route sends the value of the property whenever it changes. A state route sends the names of the active states as a JSON array, outermost first, such as `["on","full"]`, whenever they change. An event route sends the payload of the event, or `{}` for an event without one. Property and state routes hear the model only while the run has observation on, see 4.5.4.3 The Model pane.

4.7.3 Settings

The settings are texts. A missing value or one that makes no sense takes its default.

4.7.3.1 HTTP

Setting	Default	Meaning
`host`	`127.0.0.1`	The address the server listens on. The default lets only programs on the same machine in.
`port`	`8090`	The port of the server. `0` lets the system pick a free one.
`timeoutMs`	`3000`	How long an outbound `POST` waits for its answer.

The server answers a `POST` or `PUT` to the path of an inbound route with 202 at once, because the model takes the event on a later tick. A path without a route gets 404, and another method on the path of a route gets 405. A `GET` of `/` lists the paths of the inbound routes, as `{"inbound":["/dim","/press","/smash"]}`. An outbound message is a `POST` of JSON to

the URL of its route. An answer in the 200 range counts as delivered. The bridge has no authentication.

```
{
  "kind": "http",
  "settings": { "port": "8090" },
  "inbound": [
    { "from": "/press", "event": "switch/press" }
  ],
  "outbound": [
    { "property": "bulb/watts", "to": "http://localhost:9000/lamp/watts" }
  ]
}
```

4.7.3.2 MQTT

Setting	Default	Meaning
`broker`	`tcp://localhost:1883`	The broker. `ssl://`, `tls://` or `mqtts://` encrypts the connection, with 8883 as the default port.
`clientId`	`karpfen-` and the file name without `.kbridge.json`	The name of the bridge at the broker. A broker allows one connection per name.
`username`, `password`	none	The login at the broker. The password is stored as plain text in the file.
`qos`	`1`	The quality of service of subscriptions and published messages: 0, 1 or 2.
`retain`	`false`	`true` makes the broker keep the last message of each topic.
`timeoutMs`	`5000`	How long the bridge waits for the broker.

The bridge speaks MQTT 3.1.1. The `from` of an inbound route is a topic filter and may use `+` for one level and `#` for the rest. The body of a message is read as UTF-8 text. When the connection breaks, the bridge connects again on its own, and outbound messages fail in the meantime.

```
{
  "kind": "mqtt",
  "settings": { "broker": "tcp://localhost:1883" },
  "inbound": [
    { "from": "home/lamp/press", "event": "switch/press" }
  ],
  "outbound": [
    { "state": "lamp", "to": "home/lamp/state", "policy": "latest" }
  ]
}
```

Setting	Default	Meaning

4.7.3.3 ROS

Setting	Default	Meaning
`url`	`ws://localhost:9090`	The WebSocket address of the rosbridge server.
`ros1`	`false`	`true` for a ROS 1 system.
`timeoutMs`	5000	How long connecting to the server may take.

The ROS side needs the package `rosbridge_suite`, started. A type is written as in ROS 2, `std_msgs/msg/Bool`, or as in ROS 1, `std_msgs/Bool`, and the bridge writes it the way the system expects. An inbound route without a type works only for a topic that exists in ROS already. An outbound value that is a JSON object is the message itself. Any other value goes into the member `data`, as the standard types such as `std_msgs/msg/Float64` expect, and for `std_msgs/msg/String` a value that is no text is written as its JSON text, such as `["on"]`. When the connection breaks, the bridge connects again every few seconds.

```
{
  "kind": "ros",
  "settings": { "url": "ws://localhost:9090" },
  "inbound": [
    { "from": "/lamp/dim", "event": "switch/dim", "payload": { "bright": "$.data" }, "type":
    ↪  "std_msgs/msg/Bool" }
  ],
  "outbound": [
    { "property": "bulb/watts", "to": "/lamp/watts", "type": "std_msgs/msg/Float64", "
    ↪ policy": "latest" }
  ]
}
```

4.7.4 The bridge form

A bridge file opens in a form, not as text. A double-click on it in the Project tool window opens it. Run > Configure Bridge asks `Choose the bridge to configure:` with the entry `New bridge` among the bridges.

Part of the form	Content
Name, Kind	The file name without `.kbridge.json`, and HTTP, MQTT or ROS. Changing the name renames the file.
Settings	The settings of the kind: Host, Port and Timeout in ms; Broker, Client id, User name, Password, QoS (0, 1 or 2), Timeout in ms and The broker keeps the last message of each topic; Rosbridge server, Timeout in ms and Legacy ROS 1 system. An empty field shows its default.
Messages that become events	The inbound routes, each as Inbound route N with From, Event, Payload and for ROS the Message type. Payload is either The body of the message or a list of fields made with Add payload field. Add inbound route adds one.

Part of the form	Content
What the model reports	The outbound routes, each as Outbound route N with Follows (A property, Active states or An event), the property, state machine or event, To, and for ROS the Message type. Add outbound route adds one.
If too fast	The policy of a route: Queue, Keep the latest or One per period.
Pick	Lists the events, properties and state machines of the project to choose from.
Remove	Removes a route or a payload field.
Save, Cancel	Writes the file, or closes the form without writing.

A file that the form cannot read in full opens with the note `The file had problems. What could not be read is left out, and saving writes the file without it.`

4.7.5 Starting bridges with a run

The run dialog has one box per bridge file of the project, see 4.5.3 The run dialog. The ticked bridges start once the environment runs and stop with it. A bridge that fails does not stop the run. The `bridges` key of a descriptor is not used.

4.7.6 The bridges of a run

The Log pane lists the bridges of the selected run below the deployment, one line each.

Line	Meaning
`http.kbridge.json: not running`	The bridge has not started or has stopped.
`http.kbridge.json: starting`	The bridge connects.
`http.kbridge.json: running, 3 in, 1 out`	The bridge runs and has handed on 3 messages to the model and 1 message out.
`http.kbridge.json: failed. <reason>`	The bridge could not start.

Below a bridge, a route with trouble gets a line such as `/dim to switch/dim: 2 dropped, 1 failed`, with the last error in parentheses.

4.7.7 Trying the HTTP bridge

The lamp example has `http.kbridge.json` with three inbound routes. Run the lamp with the box `http.kbridge.json` ticked, then in a terminal:

```
curl http://127.0.0.1:8090/
curl -X POST http://127.0.0.1:8090/press
curl -X POST http://127.0.0.1:8090/dim -d '{"bright": false}'
```

The first command prints the paths `{"inbound":["/dim","/press","/smash"]}`. The second presses the wall switch, and the lamp goes on. The third makes it one step dimmer. The events show in the Events pane with the source `external`.

4.8 Command line

The IDE also works in a terminal, without a window. `validate` checks a project and ends. `run` validates a project, runs it with its bridges in the IDE's Docker container and prints what happens until it is stopped.

4.8.1 Commands

```
Usage:
  karpfen-ide [<folder>]
  karpfen-ide validate <folder> [--format text|json]
  karpfen-ide run <folder> [--descriptor <file>] [--bridge <file>]... [--capture] [--
    ↪ verbose]
```

Command	Effect
`karpfen-ide <folder>`	Opens the window on the folder. Without a folder the window opens on its welcome view.
`karpfen-ide validate <folder>`	Validates the whole project, prints the findings and ends.
`karpfen-ide run <folder>`	Validates the project, runs it and prints what happens until it is stopped.
`karpfen-ide validate --help`, `karpfen-ide run --help`	Prints the usage and ends with status 0. `-h` does the same.

Options may stand before or after the folder. A value follows its option as the next argument or after `=`, as in `--bridge=http.kbridge.json`. `--` ends the options, so a folder after it may start with `-`. A folder that is itself named `validate` or `run` must be written with a path in front, as in `./run`, when it is the first argument.

A wrong command line prints `karpfen-ide:` and what is wrong, then the usage, on the error output, and ends with status 2. Wrong are an unknown option, a missing, second or empty folder, a folder that does not exist, an option without its value, a second `--descriptor` or `--format`, a format other than `text` or `json`, a bridge named twice, and an option that the command does not take.

```
karpfen-ide: unknown option '--frmat'
```

4.8.2 Starting it

From	Command
The sources	`./gradlew :app:ide:run --args="validate /absolute/path/to/project"`
The application on macOS	`karpfen-ide.app/Contents/MacOS/karpfen-ide validate <folder>`
The application on Linux	`karpfen-ide/bin/karpfen-ide validate <folder>`
The application on Windows	`karpfen-ide\karpfen-ide.exe validate <folder>`

From the sources the folder must be an absolute path. `./gradlew :app:ide:`

`createDistributable` builds the application into `app/ide/build/compose/binaries/main/app/`. On Windows the application is built with a console, so the commands have somewhere to print, and opening the window by a double-click also opens a console window.

4.8.3 Validating

```
karpfen-ide validate examples/lamp
```

The command runs the same checks as Validate in the window, see 4.4 Validating. It prints one line per finding, errors first, and then a summary. A project without findings prints `No problems found.`

```
lamp.kstates:13: error: mismatched input 'STATMACHINE' expecting 'STATEMACHINE'
switch.kstates:8: warning: This EVAL has no return, so it yields no value; return the value
    ↪ it computes
1 error and 1 warning
```

The language features are those of the `.karpfenproject` file of the folder, see 4.1.7 Language features. `karpfen-ide validate examples/spares` checks the example with attachable state machines, whose project file switches on macros. A finding about a `behaves after` is printed at the line of the metamodel, as in `spares.kmeta:4: error: ...`.

With `--format json` it prints one JSON document instead, for a program to read.

```
karpfen-ide validate examples/lamp --format json
```


```
{
    "root": "/home/me/karpfen-IDE/examples/lamp",
    "errorCount": 1,
    "warningCount": 1,
    "problems": [
        {
            "path": "lamp.kstates",
            "line": 13,
            "column": 1,
            "severity": "error",
            "message": "mismatched input 'STATMACHINE' expecting 'STATEMACHINE'",
            "code": "syntax-error"
        },
        {
            "path": "switch.kstates",
            "line": 8,
            "column": 33,
            "severity": "warning",
            "message": "This EVAL has no return, so it yields no value; return the value it
    ↪ computes",
            "code": "kstates.eval-without-return"
        }
    ]
}
```

Member	Meaning
`root`	The folder of the project, as an absolute path.
`errorCount`, `warningCount`	How many errors and warnings the project has.
`problems`	The findings, sorted by path, line and column.
`path`	The file, relative to the folder.
`line`, `column`	Where the finding starts. Both count from 1.
`severity`	`error`, `warning`, `info` or `hint`.
`message`	What is wrong, in words.
`code`	The stable name of the kind of finding.

4.8.4 Running

```
karpfen-ide run examples/lamp --descriptor lamp.kenv.json --bridge http.kbridge.json --
    ↪ capture
```

Option	Meaning
`--descriptor <file>`	The descriptor to run, relative to the folder or absolute. It can be left out when the project has exactly one.
`--bridge <file>`	A bridge file of the project to start with the run. Give the option once per bridge. Without it no bridge starts.
`--capture`	Writes the log of the run to a file in `karpfen-runs/`, as Capture run to file does in the window, see 4.5.8 Captured logs.
`--verbose`	Also prints every line of the runtime's log.

The choice of files is checked against the project, and a wrong choice ends with status 2, a message and the usage.

Case	Message
The project has no descriptor.	`the project has no environment descriptor (*.kenv.json)`
The project has several descriptors, and `--descriptor` is missing.	`the project has several environment descriptors; choose one with --descriptor:` and the list
`--descriptor` names no descriptor of the project.	`<file> is no environment descriptor of the project;` and the list
`--bridge` names no bridge file of the project.	`<file> is no bridge file of the project;` and the list

A run goes through these steps.

1. It validates the whole project. With an error it prints the findings as `validate` does and does not run. With warnings only it prints them the same way, then `The run goes on despite the warnings.`, and runs.
2. It starts the runtime in the IDE's Docker container. The first run builds the runtime image.
3. It deploys the descriptor and starts the bridges.

4. It prints what happens until you stop it with Ctrl+C or a terminate signal.
5. It stops the bridges and the environment, removes the runtime container and ends.

Stopping the environment in step 5 also drops it, so the runtime forgets it before the container goes. A run whose environment another client stops or drops ends with `The run ended on its own.`, as 4.5.5 Stopping a run describes.

This is what a run of the lamp with one bridge prints. The switch is pressed once through the bridge, and then the run is stopped. The first run prints a line for each step of building the image as well.

```
Running lamp.kenv.json with http.kbridge.json
Docker: not started
Run lamp: preparing
Run lamp: waiting for the runtime
Docker: checking the Docker engine
Docker: creating the container
Docker: starting the container
Docker: waiting for the runtime
Docker: running at 127.0.0.1:65457
Run lamp: deploying
http.kbridge.json: not running
Checking the runtime at http://127.0.0.1:65457: done, The runtime is up and supports the
    ↪ IDE
Creating the environment: done, env-1794579423685
Connecting to the environment: done, Connected as karpfen-ide-254666e10161
Uploading lamp.kmeta: done
Uploading lamp-events.kmeta: done
Uploading lamp.kmodel: done
Attaching lamp.kstates to lamp: done
Attaching switch.kstates to wallSwitch: done
Setting the tick delay to 500 ms: done
Setting the event time to live to 5000 ms: done
Listening to 2 event domains: done, service, switch
Switching on runtime features: skipped, The project switches on no language feature that
    ↪ needs one
Activating the environment: done
Switching observation on: done, The runtime read the model as analyzed
Starting the environment: done
14:17:03.888  lamp  starts in off
14:17:03.890  wallSwitch  starts in down
14:17:03.892  engine  Engine started with 2 state machine(s), tickDelay=500ms
14:17:03.895  lamp  runs the entry actions of off
lamp: off
wallSwitch: down
Confirming the start: done, Every state machine reported its state
Run lamp: running
http.kbridge.json: starting
http.kbridge.json: running, 0 in, 0 out
14:17:04.896  lamp  off -> full
14:17:04.898  wallSwitch  down -> up
lamp: on > full
wallSwitch: up
14:17:05.397  lamp  runs the entry actions of on
14:17:05.399  lamp  runs the entry actions of full
14:17:05.414  wallSwitch  runs the entry actions of up
Stop requested.
Shutting down the run and its bridges, then the runtime.
Run lamp: stopped
```

```
http.kbridge.json: not running
14:17:06.405  engine  Engine stopped after 5 ticks
Docker: not started
Shut down.
```

Line	Meaning
`Running <descriptor> with <bridges>`	What the run was asked to do. `, captured to a file` is added with `--capture`, and `, parallel branches on threads of their own` while the project has Parallel branches on threads switched on, see 4.1.7 Language features.
`Docker:  ...`	What the runtime container does, in the words of the Runtime tool window.
`Run <name>:  <phase>`	The phase of the run, as in 4.5.4.2 The runs.
`<step>:  done`, `skipped` or `failed`, with a detail	A step of the deployment.
`<object>:  <states>`, `<object> (<definition>):  <states>`	The active states of the state machine instance on that object, whenever the runtime reports new ones. An instance of a definition also names its definition. An instance that ends and a later one on the same object are two instances, so the first states of the later one are printed too.
`<bridge file>:  ...`	The state of a bridge, and once it runs how many messages it took in and sent out.
`<time> <object> <text>`	A line of the simplified log, as the Log pane shows it.
`<severity>:  <file>:<line>:<column>: <message>`	A problem of the preparation or the run.
`[stdout] ...`, `[stderr] ...`	A line of the runtime's log, with `--verbose`.
`Captured to karpfen-runs/<file>`	Where the run is captured.
`Stop requested.`, `Shutting down ...`, `Shut down.`	The end of the run. What the bridges and the runtime say while they stop stands in between.

A descriptor with definitions uploads each of them in a step `Uploading the definition <file>`. The objects get and lose instances of the definitions while the run goes on, and the simplified log says so in lines such as `bulb attaches glowing`, `bulb exchanges glowing for flickering` and `bulb detaches flickering`, see 4.5.4.7 State machine instances. This is part of what a run of `examples/spares` prints when the event `flicker` comes in:

```
bulb (glowing): lit
spare (glowing): lit
14:20:16.500  bulb  exchanges glowing for flickering
14:20:16.501  bulb  starts in on
bulb (flickering): on
```

A descriptor with fragments uploads each of them in a step `Uploading the fragment <file>`. The states of a fragment carry the name of their container state in front, so a run of `examples/pages` prints lines such as `lamp:  self test > self test/wiring > self test/wiring/measuring`, see 4.5.4.8 Pages.

A run that does not get going says why and ends with one of these lines.

Line	Status
`The run does not start:  the project has errors.`	1
`The run did not start.`	3
`The run does not start:  the deployment is blocked.`	3
`The run failed.`	3
`The run ended on its own.`	3

4.8.5 What a run reads from the project

A run without a window takes one thing from `.karpfenproject`: which features are on, the optional language features and the features of the project such as Parallel branches on threads. Everything else in that file, such as the bridges ticked in the window, is ignored. Nothing is written, neither to `.karpfenproject` nor to the settings of the user. A run without a window has no debugger, so breakpoints never halt it, also when the project has them switched on. It has no time limit and ends only when it is stopped.

4.8.6 Exit statuses

Status	Meaning
0	The validation found no error. Warnings are allowed. Also after `--help`.
1	The project has errors. A run did not start.
2	The command line was wrong. A message and the usage are printed.
3	A run could not start, failed or ended on its own, or the folder could not be opened as a project.

A run that is stopped by a signal ends with the status the system gives a process that got that signal. A run started through Gradle therefore makes the Gradle task fail when it is stopped.