YAKINDU: Using YET

Web-Clip

1 ng YET

"." counter(h2, decimal)

In order to use execution tracing for target debugging or unit testing some implementation and configuration steps are necessary. In general the system which executes the statechart must be prepared to produce an execution trace and the system which analyses a YET trace must be prepared properly. Depending on the system simple configurations or manual implementation tasks are required. Here the following cases are described:

1. Preparing an embedded system for tracing
2. Trace and debug statechart execution
3. Unit test a statechart YET trace

1 . counter(h2, decimal) . counter(h3, decimal) . Preparing an embedded system for tracing

A generated state machine which should be debugged or remote controlled must be enabled for execution tracing. This consists of the following steps.

Preparing the generated state machine for execution tracing.
Set up the trace handling in the target application. This step may require application specific implementation work.

By default the tracing feature of state machines is disabled in order to save resources if it is not required. Nevertheless enabling tracing for a state machine is a simple configuration topic for code generators. The second step is more complex as it may require application specific implementation work.

Enable tracing when generating state machine
The generated state machine code must be instrumented for raising trace

events. So you first have to enable the tracing feature in your sgen generator file. This can be done by adding another feature called *“Tracing”* to the C generator configuration. The configuration below gives an example.

```
GeneratorModel for yakindu::c {

    statechart tictoc {
        feature Outlet {
            targetProject = "example"
                targetFolder = "sc"
                libraryTargetFolder = "sc/base"
        }
        feature Tracing {
            generic = true
        }
    }
}
```

The feature *“Tracing”* has the property *“generic”* with the value *“true”* (*“false”* is the default). As a result an additional C header file named *“sc_tracing.h”* is generated. It declares the generic trace API which will be used by all state machines with enabled tracing. These declarations include the type `sc_trace_handler`. A state machine uses such a trace handler instance to call trace callbacks which can be processed by a trace handler implementation. This API is independent of YET. It can be used to adapt any execution tracing or logging infrastructure.

Adding a YET tracer

While the step before enables a generic trace API for the state machine, an additional generator can be configured which generates a YET specific implementation of the `sc_trace_handler`. This is also straightforward. Simply add a new *sgen* file to the project. Corresponding to the example above it looks like:

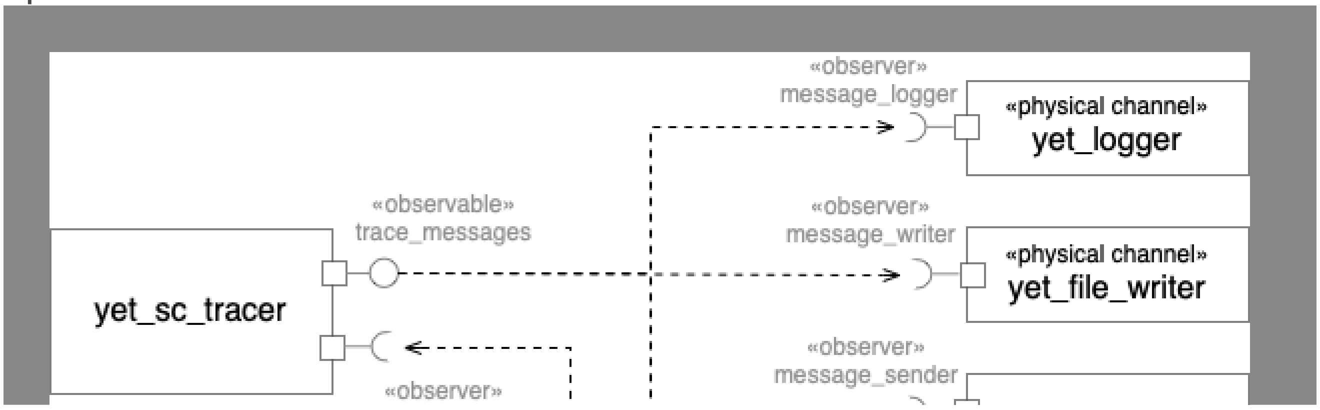
```
GeneratorModel for yakindu::c::yet {

        statechart tictoc {
                feature Outlet {
                        targetProject = "example"
                        targetFolder = "sc"
                        libraryTargetFolder = "sc/base"
                }
        }
}
```

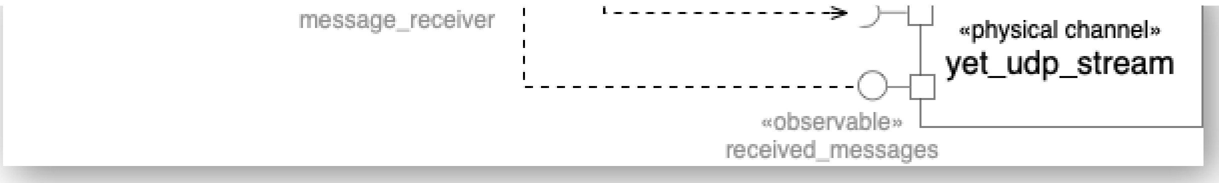
This generator model uses `yakindu::c::yet` generator instead of the `yakindu::c`. It just requires an `Outlet` feature and the same values should be used as in the normal C code generator model. This generator adds several additional source files to the project:

`sc_yet.[h|c]` - implements a set of functions which are used to create and parse YET execution events. This implementation is completely independent of statechart semantics and can be used to support execution tracing for other models.
`yet_sc_tracer.[h|c]` - this module implements all generic parts for YET statechart tracing which can be applied to all statecharts. So it is aware of statechart semantics and provides an implementation of `sc_trace_handler`.
`<Statechart>Tracer.[h|c]` - this module implements the statechart model specific parts of a YET tracer.
`<Statechart>Meta.[h|c]` - this module defines data structures which provide names as strings for the different statechart features like variables, events, and states. This is separated from the tracer module as these structures are independent from YET and can be reused for other purpose (like a generic MQTT adaption).

Please keep in mind, that the generic tracing feature currently only supports the C language.

Setting up tracing in the application
The generated infrastructure is capable of producing and consuming trace. What is missing is connecting this functionality to the outside world. This part is not covered by a code generator and must be implemented manually based on the API which is provided by the generated code and reusable software components.





Connecting trace infrastructure to physical channels

The generated code provides a `yet_sc_tracer` instance. This instance must be connected to physical channels which care about the physical handling of trace events. Three different implementations of physical channels are currently provided:

- `yet_logger` - logs trace events by writing them to the standard output stream of the process.
- `yet_file_writer` - writes trace events to a file using a single line for each entry. The result is a valid yet trace file.
- `yet_udp_stream` - implements a bidirectional transport of trace events based on UDP datagrams. Each datagram contains a single trace event.

The integration of the tracer instance on the one side and the physical channels on the other side are based on observable message streams. These support an asynchronous, reactive programming model and allow a loose coupling between the different components. The message streams are based on the principles of reactive extensions (ReactiveX) without making use of any external ReactiveX library.

The `yet_sc_tracer` provides an observable stream of trace messages where each trace message is a simple string. This stream can be observed by any number of observers and all physical channels provide an observer which can subscribe to the observable trace messages. In addition, the tracer defines the observer `message_receiver` which processes incoming trace events which are stimuli for the state machine. This can, for instance be connected to the observable `received_messages` which is provided by the bidirectional `yet_udp_stream`. The figure above shows how the observable streams are connected. On the code level the setup is straight forward. First, we need a state machine and initialize it.

```
    SomeStateMachine machine;
    someStateMachine_init(&machine);
```

The code for setting up the timer service is omitted here. Next, a tracer instance is defined and initialized.

```
    yet_sc_tracer tracer;
    someStateMachine_init_sc_tracer(&tracer, &machine);
```

Without a physical channel, tracing has no effect. So we need to define and initialize an instance for each discussed physical channel.

```
    yet_file_writer yet_file;
    yet_udp_stream  yet_stream;
    yet_logger      yet_log;

    yet_file_writer_init(&yet_file, "machine.yet");
    yet_udp_stream_init(&yet_stream, ip, port);
    yet_logger_init(&yet_log);
```

Now all instances are in place but we have to connect the observable streams with the observers as discussed above. This is done by defining a list of observers for each observable which must be connected.

```
    sc_observer* out_trace_observers[] = {
                 &(yet_log.message_logger),
                 &(yet_file.message_writer),
                 &(yet_stream.message_sender) };

    sc_observer* in_trace_observers[] = {
                 &(yet_log.message_logger),
                 &(tracer.scope.message_receiver) };

Then the observers must subscribe to the observable.

    SC_OBSERVABLE_SUBSCRIBE(&(yet_stream.received_messages),
                            in_trace_observers);
    SC_OBSERVABLE_SUBSCRIBE(&(tictocTracer.scope.trace_messages),
                            out_trace_observers);
```

The observer subscriptions here go beyond the scenario which is described by the previous figure as the `yet_log` will also care about the incoming trace events from the `yet_stream` instance. So it logs outgoing and incoming trace events. Some more things should be mentioned here:

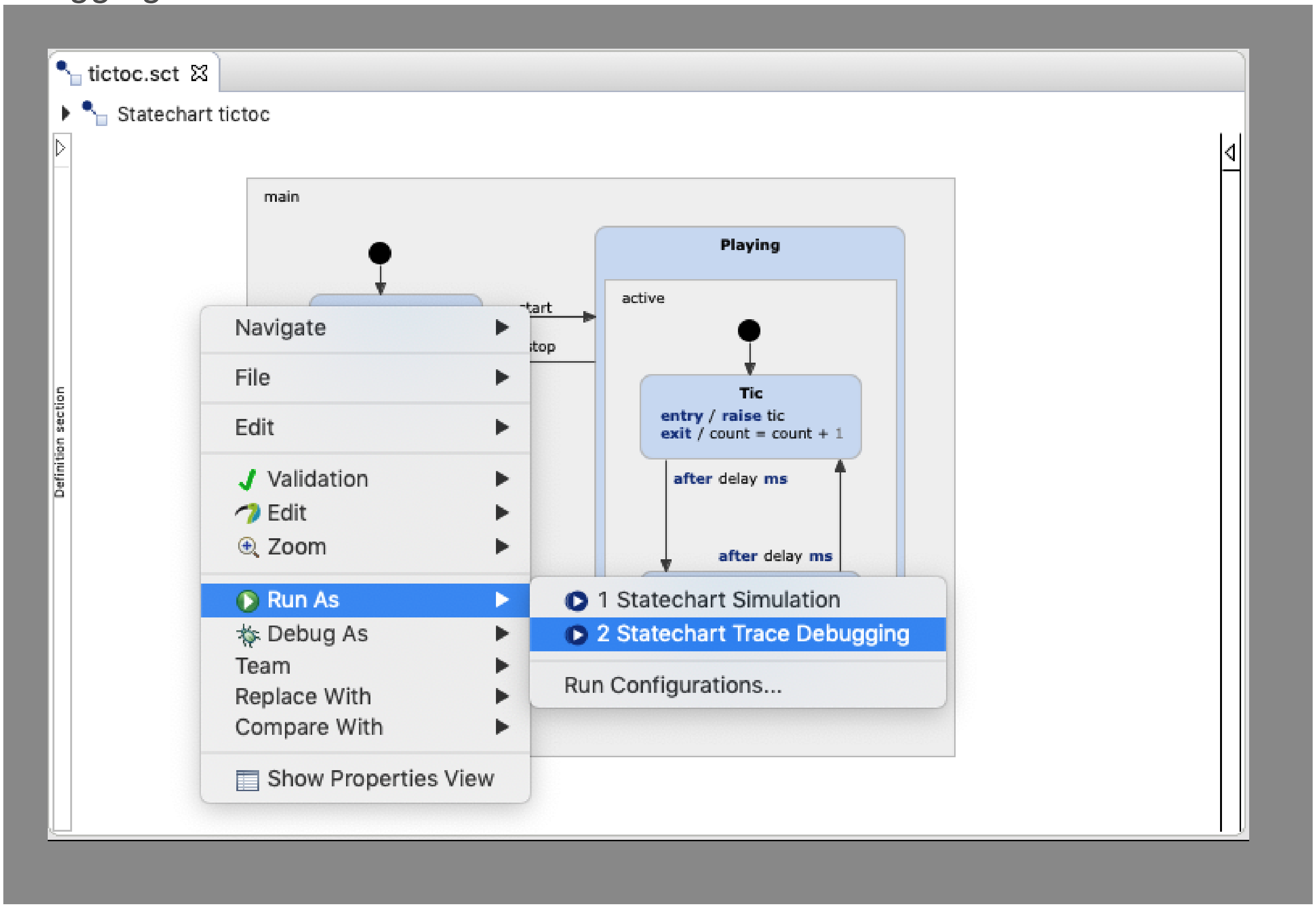
1. The involved instances do not know anything about the other instances.

So the configuration can be changed without impact on the implementation of these instances.
2. There can be any number of physical channel instances. E.g. multiple trace files could be defined.
3. It is easy to implement custom physical channels and add them to the scenario.
4. Observers and observables can also be used to implement other functionality than physical channels like buffers, filters, aggregators etc.. E.g. a simple observable could implement a trace event counter.

This provides a high flexibility for the application developer and the implementation of the existing physical channels is a good template for custom implementations.

2 . counter(h2, decimal) . counter(h3, decimal) . Trace and debug statechart execution

Debugging a statechart YET trace is simple. Within a statechart editor or on the model file entry in the project explorer choose *“Run As > Statechart Trace Debugging”* from the context menu.



Starting a trace debug session

This will launch the trace debugger and by default will try to read the execution trace from the file *"trace.yet"* from the root of the project folder. If it is not already in place then create this trace file e.g. by starting the application with enabled YET file tracing. The UI of the trace debugger is identical to the regular simulation UI.

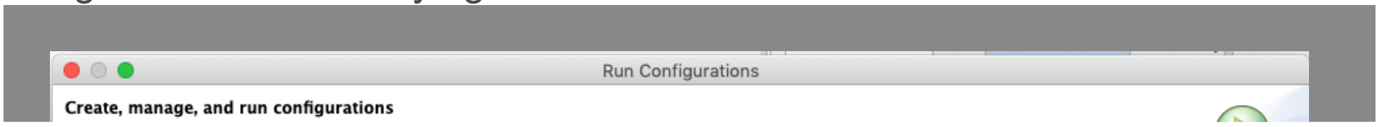
If you want to choose a different trace file you have to reconfigure the run configuration which was created by the *"Run As > Statechart Trace Debugging"* action. To do this choose *"Run As > Run Configurations…"* from the context menu. The *"Run Configurations"* dialog pops up and choose the proper entry in the category *"Statechart Trace Debugging"*. The tab *"Statechart Trace"* provides several configuration options.

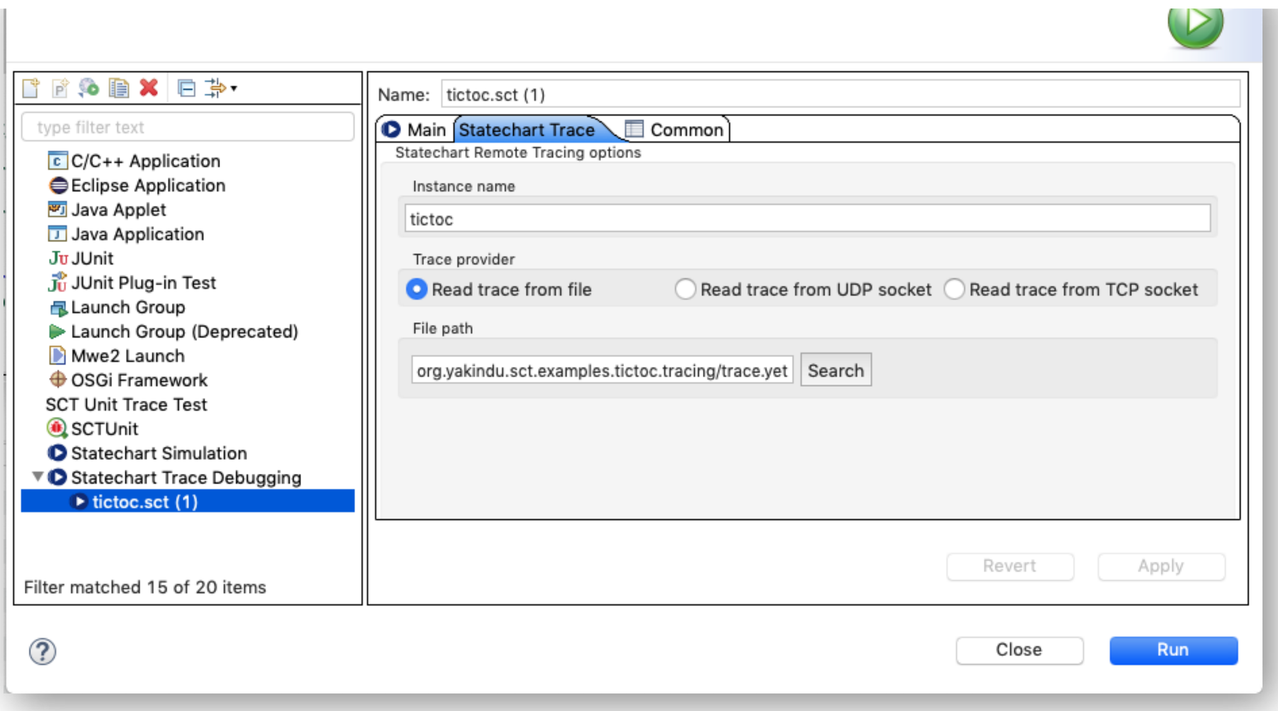
First the instance name is used to distinguish between multiple running instances of the same state machine if multiple state machines are executed on the target and if they share a common trace channel. The default is just the name of the statechart model.

The obviously most important option is how you want to read or receive traces. It is possible to choose one of three trace provider:

- *Read trace from file* - supports to configure what file should be used.
- *Read trace from UDP socket* - opens a UDP socket on the local machine with a configurable port (default 444). The physical trace channel on the embedded target can connect to this port to setup a bidirectional communication channel.
- *Read trace from TCP socket* - uses TCP instead of UDP to setup a bidirectional communication channel and can be configured in the same way.The default port is 8444.

If UDP or TCP based YET streams are used then the debugging UI supports raising events and modifying statechart variables.




type filter text

- C/C++ Application
- Eclipse Application
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Launch Group
- Launch Group (Deprecated)
- Mwe2 Launch
- OSGi Framework
- SCT Unit Trace Test
- SCTUnit
- Statechart Simulation
- Statechart Trace Debugging
  - tictoc.sct (1)

Filter matched 15 of 20 items

Name: tictoc.sct (1)

Main | Statechart Trace | Common

Statechart Remote Tracing options

Instance name: tictoc

Trace provider: [✓ Read trace from file] Read trace from UDP socket Read trace from TCP socket

File path: org.yakindu.sct.examples.tictoc.tracing/trace.yet Search

Configuration of trace debug session

After that click “Apply” and “Run”. The trace debugger UI will be activated. If UDP or TCP trace provider are used then no active statechart state may be highlighted in the Debugger UI. This is always the case if no remote target is connected or if the statechart is not yet activated on the remote target.

3 . counter(h2, decimal) . counter(h3, decimal) . Testing statechart execution traces

Instead of the interactive debugger a trace can also be consumed by unit tests. Within a SCTUnit editor or on test file entry in the project explorer choose *“Run As > Statechart SCT Unit Trace Test”* from the context menu. The unit test runner will start up an execute the tests.

All points regarding setting up a run configuration for trace tests is identical to the steps required for the interactive trace debugging as the same configuration dialogs are applied. The only difference is that the category *‘SCT Unit Trace Test’* is used instead of *‘Statechart Trace Debugging’* .