State Diagram Library Manual

state-diagram.com

Please save resources, do not print this manual. Read it on–screen instead.

Contents

1 Disclaimer

Information in this document is provided as is, with all faults. State Diagram library contributors and manual authors expressly disclaim all warranties, representations, and conditions of any kind, whether express or implied, including, but not limited to, the implied warranties or conditions of merchantability, fitness for a particular purpose, and non-infringement. State Diagram contributors and manual authors do not assume any liability rising out of the application or use of any artifact described herein and specifically disclaims any and all liability, including without limitation indirect, incidental, special, exemplary, or consequential damages.

State Diagram contributors reserve the right to make changes without further notice to any artifacts described herein.

2 Introduction

State Diagram is a C++ library that supports specifying and executing hierachical, concurrent, finite state machines. Such state machines are particularly prevalent in embedded systems. The present manual assumes general knowledge of hierarchical, concurrent, finite state machines, their building blocks, and how they can be put to use.

As C++ relies heavily on object–oriented features, the specific approach to state machines implemented in State Diagram may, at first glance, appear like somewhat of a paradox: State Diagram permits specifying state machines almost without using any object–oriented C++ features at the API level. The classical object–oriented approach models any given concurrent region by introducing a class whose instances represent all the states that appear in that region. A transition trigger corresponds to a class method that can be called on any given source state to yield a target state — so that transitions are implicit in the pre– and post–conditions of such methods. State hierarchies are modelled by means of inheritance. By contrast to that, State Diagram takes a more elementary route that goes directly from hierarchical state machines to hierarchical graphs: There are pre–supplied classes, one per type of state, whose instances represent states regardless of where in the hierarchy they reside. Hierarchical relationships are expressed by declaring them for each parent/child relationship separately and explicitly. Transitions and transition decorations — triggers, guards, outputs, actions — are also declared as first–class citizens, as instances of other pre–supplied classes, rather than leaving them implicit in pre–/post–condition relationships and the like. This overall approach to specifying state machines is akin to "drawing" them not graphically but textually, as it were. Doing away with all the disadvantages that a genuinely graphical GUI incurs, textual drawing turns out to be intuitive, versatile, and expressive.

Doing away, too, with almost all object–oriented patterns in its outward API, State Diagram can be used in ways that are very much C–like. This property lessens the C++ learning curve if C is the language a user comes from, like many embedded programmers do. At the same time, and already by the very fact that it is a programming language library and not a graphical tool, State Diagram emphasizes coding efficiency, testability, and flexible reuse. Optionally, these objectives can then be helped even more by using whatever C++ has to offer, including all of its object–oriented features, on top of the basic API. State Diagram facilitates that very easily in nicely static–looking ways.

State Diagram supports a broad range of state machine features, to a large degree emulating UML statecharts in this regard. It is also very pure in that computations on data and operations yielding side effects are not part of its API. State Diagram allows for these aspects to be injected into state machines very simply by means of C++ lambdas. Hence the full power of C++ can be brought to bear to express computations on data and side effects directly. Cumbersome and laborious code emulation or code import features as it is typical for graphical state machine tools are totally foregone by State Diagram. Employing State Diagram means coding in C++ directly.

State Diagram has a very clean and non–ad–hoc yet versatile *signaling model*. According to this model, signals or, equivalently, transition triggers once activated stay so until a transition sequence, a so–called *macro step*, has run to completion. This model is inspired by a computer science concept that is known as *perfect synchrony*. Perfect synchrony combines versatility with mathematical tractability, a sweet spot that has been

chosen with a view to adding formal verification functionality to State Diagram in the future. State machine tools and frameworks other than State Diagram partly struggle with impaired mathematical tractability. The reason for that lies more often than not in a less–than–optimal signaling model that deviates from perfect synchrony.

State Diagram maps concurrency to executing cocurrent regions in lockstep. Thread libraries or inbuilt threads are not required to execute State Diagram state machines that have concurrency in them.

State Diagram uses the heap to allocate state machine components. It does not induce any further heap usage at any point thereafter unless a state machine employs local event vectors in transition outputs — see Section 6.

State Diagram is free and open source with generous licensing according to the Apache 2.0 license.

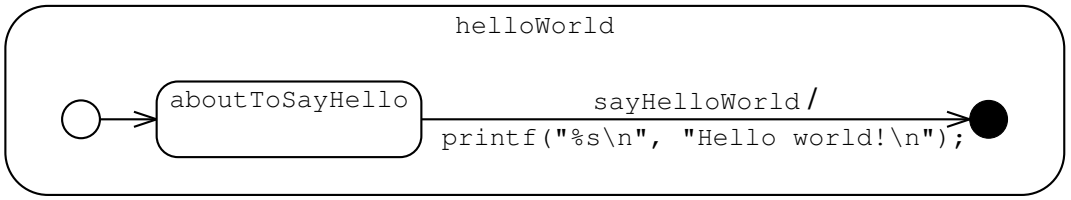
Simple on the outside, State Diagram is heavily patterned on the inside, thereby making it easy to review and modify its code base.

2.1 Getting Started

Download the .zip file at state-diagram.com, unpack it, and place folders `Include` and `Src` or their contents wherever it suits you. State Diagram comes without any build mechanics. The `#include`s making State Diagram available have to point to `state_diagram/state_diagram.h`. Then, everything inside State Diagram (except macros) lives inside `namespace state_diagram`. State Diagram was developed using GCC 9 with –std=c++2a, so that is the platform (or newer) that should be employed to use it. Attempting to compile and run State Diagram with GCC versions earlier than 9.1 is known to have triggered compiler bugs.

3 State Diagram in Action

Consider a state machine that prints "Hello world!"



This state machine can be specified using State Diagram like so:

```
#include <cstdio>
#include <state_diagram/state_diagram.h>

using namespace state_diagram;

int main()
{
  FSM_TOP(helloWorld);
  FSM_SIGNAL(void, sayHello, helloWorld);
  FSM_INIT(helloWorld);
  FSM_STATE(aboutToSayHello, helloWorld);
  FSM_FINAL(helloWorld);
  FSM_AUTO(helloWorld_INIT, aboutToSayHello);
  FSM_STEP
  (
    aboutToSayHello
  , helloWorld_FINAL
  , Trigger(sayHello)
  , Action([]{printf("%s\n", "Hello world!\n");})
  );
  M.init();
  M.step();
  M.step(sayHello);
  return 0;
}
```

Let us go through this example statement–by–statement:

- `FSM_TOP(helloWorld)` — defining a *top state* or, equivalently, a *state machine*, `helloWorld`.
- `FSM_SIGNAL(void, sayHello, helloWorld)` — defining a `void` (i.e. non–data carrying) *signal*, `sayHello` as belonging to the state machine. This signal is to *trigger* the action of printing “Hello world!” in the terminal window.
- `FSM_INIT(helloWorld)` — defining an *initial state* directly beneath the top state, `helloWorld_INIT`.
- `FSM_STATE(aboutToSayHello, helloWorld)` — defining an *ordinary state* directly beneath the top state, `aboutToSayHello`. In this case there is nothing beneath the ordinary state, whence it is a *leaf state*.

- `FSM_FINAL(helloWorld)` — defining a *final state* directly beneath the top state, `helloWorld_FINAL`.
- `FSM_AUTO(helloWorld_INIT, aboutToSayHello)` — defining a (triggerless) *auto*–transition from `helloWorld_INIT` to `aboutToSayHello`. This transition, by (a) being attached to an initial state directly beneath the top state and (b) not carrying any guards, becomes *enabled* immediately upon starting up the state machine. All it does consists of transferring control flow to state `aboutToSayHello`.
- `FSM_STEP(aboutToSayHello, printer_FINAL, Trigger(sayHellWorld), Action([]{printf("%s\n", "Hello world!\n");}))` — defining a (triggered) *step* transition from `aboutToSayHello` to `helloWorld_FINAL`. The transition carries an *action* that is parameterized by a lambda that achieves the desired side effect. Such an addition to a transition is called a *spec* — see below. The transition becomes enabled if (a) signal `sayHello` is *active* and (b) control flow resides at state `aboutToSayHello`. Upon execution, it performs the action and, simultaneously, transfers control flow to state `helloWorld_FINAL`.
- `helloWorld.init()` — initializing the state machine. Control flow now resides at state `helloWorld_INIT`.
- `helloWorld.step()` — letting the state machine execute the auto transition. Control flow now resides at state `aboutToPrint`.
- `helloWorld.step(sayHello)` — letting the state machine execute the step transition. This transition becomes enabled by control flow and because signal `sayHello` is being actived due to it appearing as an (optional) parameter to `step`. Executing the step transition lets the state machine print “Hello World!”.

Please note how almost C–like this example turns out to be. No new classes are introduced. Otherwise, object–orientedness makes an overt appearance only in the form of the .–notation being used to invoke functions `init` and `step` on the top state.

State Diagram does not impose any artificial non–orthogonalities. Among other things, orthogonality means that auto–transitions may carry specs. The hello–world example could, thus, be simplified as follows, doing away with the external signal and the step transition (and omitting the top matter, which stays the same).

```
int main()
{
  FSM_TOP(helloWorld);
  FSM_INIT(helloWorld};
  FSM_FINAL(helloWorld);
  FSM_AUTO
  (
    helloWorld_INIT
  , helloWorld_FINAL
  , Action([]{printf("%s\n", "Hello world!\n")})
  );
  M.init();
  M.step();
  return 0;
}
```

A more abstract example is the following one. It consists of a state machine `M` with an ordinary state `S` that is subdivided into two concurrent regions `R1` and `R2`. These two regions communicate via a local signal `A`. An entire run to termination requires two calls to member function `step`.

```
#include <cassert>
#include <state_diagram/state_diagram.h>

using namespace state_diagram;

int main()
{
  FSM_TOP(M);

  FSM_INIT(M);
  FSM_STATE(S, M);
  FSM_FINAL(M);
  FSM_AUTO(M_INIT, S);
  FSM_AUTO(S, M_FINAL);

  FSM_LOCAL_SIGNAL(void, A, S);

  FSM_REGION(R1, S);
  FSM_INIT(R1);
  FSM_FINAL(R1);
  FSM_AUTO(R1_INIT, R1_FINAL, Output(A));

  FSM_REGION(R2, S);
  FSM_INIT(R2);
  FSM_FINAL(R2);
  FSM_STEP(R2_INIT, R2_FINAL, Trigger(A));

  M.init();
  M.step();
  assert(M.step());
  return 0;
}
```

4 State Machines / State Hierarchies

A *state machine* or, equivalently, a *state hierarchy* always starts out at a *top state* as shown above. Top states have to and *ordinary states* can be sub–divided into one or more *orthogonal regions*, *regions* for short, whence top states and ordinary states are called *compound states*. Regions, in their turn, are made up of *sub–states* and *transitions*, and so on. If a compound state consists of at least two regions, then these regions are executed concurrently. A state that has exactly one region does not have to have this region explicitly attached to it; sub–states in this region can be attached to the state directly instead. A default region is then placed in between the state and its sub–states implicitly. Ordinary states are most often called *states* if no confusion can arise.

Function `init` has to be invoked on a top state to initialize it before calling function `step`, usually repeatedly, to let it go through maximal sequences of transition executions (that what is called *run to completion* in UML) — so that the state machine generates actions. Such a maximal execution sequence is called a *macro step*. As already mentioned, State Diagram uses the convention that a step transition that executes does not deactivate the trigger signal. Once activated, signals stay activated over the entire course of a macro step. They become deactivated once the macro step is completed. Then, in accordance with UML, a transition that originates at an ordinary state is enabled only if no transition originating at an immediate or non–immediate sub–state of that state is enabled. A compound state has *terminated* if, and only if, the current state of all its immediate sub–regions is a final state. Function `step` returns true if, and only if, the state machine has terminated due to the state machine undergoing the macro step triggered by it. The function takes any number of external signals as arguments, including zero signals, activating each one before letting the state machine go through a macro step in accordance with these activations.

Table 1 summarizes what kinds of states there are in State Diagram and what properties they have. Constructor argument `name` is always of type `std::string const&`; constructor argument `parent` is always either of type `CompoundState const&` or of type `Region const&` — so that each type of sub–state has two constructors: one with a state as the parent of the sub–state, the other one with a region as the parent of the sub–state. In case a sub–state is defined with a state as its parent, the region is inserted implicitly as a default region in between that parent and the sub–state. It is not permitted to attach any region explicitly to a state that already has a default region. Conversely, it is not permitted to attach any default region — implicitly — to a state that already has a region attached to it explicitly. Attempting either of these operations leads to an `Error` being thrown. Default regions always have a default name of `"REGION"`.

Table 1: States in State Diagram

Type in `namespace state_diagram`	Constructor arguments	Is sub–state	Is compound state	Can be transition source	Can be transition target	Pauses local control flow
`Top`	`name`	No	Yes	No	No	N/A
`Init`	`parent`	Yes	No	Yes	No	No
`State`	`name, parent`	Yes	Yes	Yes	Yes	Yes
`Connector`	`name, parent`	Yes	No	Yes	Yes	No
`Final`	`parent`	Yes	No	No	Yes	Yes

Type `Region` in `namespace state_diagram` has as constructor arguments a `name` of type `std::string const&` and an `parent` of type `CompoundState const&`.

Names must always be unique per parent with initial and final states always carrying default names of `"INIT"` or `"FINAL"`, respectively. Then, it is often the case that the name given as a constructor argument and the programmatic name coincide, as in

```
State const aboutToSayHello{"aboutToSayHello", helloWorld};
```

To help avoiding this tediousness, State Diagram provides a number of convenient macros for defining states and regions, see Table 2.

Table 2: Macros and constructors for defining states and regions.

Macro ↦ The corresponding constructor
`FSM_TOP(`*`name`*`)` ↦ `Top const` *`name`*`("`*`name`*`")`
`FSM_INIT(`*`parent`*`)` ↦ `Init const` *`parent`*`_INIT(`*`parent`*`)`
`FSM_STATE(`*`name`*`,` *`parent`*`)` ↦ `State const` *`name`*`("`*`name`*`",` *`parent`*`)`
`FSM_CONNECTOR(`*`name`*`,` *`parent`*`)` ↦ `Connector const` *`name`*`("`*`name`*`",` *`parent`*`)`
`FSM_FINAL(`*`parent`*`)` ↦ `Final const` *`parent`*`_FINAL(`*`parent`*`)`
`FSM_REGION(`*`name`*`,` *`parent`*`)` ↦ `Region const` *`name`*`("`*`name`*`",` *`parent`*`)`

An ordinary state always leads to local control flow pausing once the state has been reached. To enable transition sequencing during one and the same macro step, *connector states*, *connectors* for short, are provided. Connectors cannot be subdivided.

5 Signals and Transitions

External signals tell a state machine to do certain things; analogously to that, *local signals* emitted somewhere inside of a state machine tell some other part of the state machine to do certain things. Local signals, thus, serve as a means of communication between parts of a state machine that run concurrently to each other, and also as a means of establishing dependencies on sequential execution paths. Dependencies via local signals can only occur within macro-steps — see below.

The following table summarizes what convenience macros there are for defining signals. The underlying constructors can be gleaned from the macro expansions. There are no other signal constructors.

Table 3: Macros and constructors for defining signals.

Macro ↦ The corresponding constructor
`FSM_SIGNAL(`*`type`*`, `*`name`*`, `*`parent`*`)` ↦ `ExternalSignal<type> const `*`name`*`("`*`name`*`", `*`parent`*`)`
`FSM_LOCAL_SIGNAL(`*`type`*`, `*`name`*`, `*`parent`*`)` ↦ `LocalSignal<type> const `*`name`*`("`*`name`*`", `*`parent`*`)`

The parent of an external signal must be a top state; the parent of a local signal must be a top state, ordinary state, or a region, thereby assigning a scope to the signal being defined. The local signal in question must not be used outside this scope. The respective point of usage is always defined to be the source or the host of any transition where the signal appears as a trigger or transition output, see next.

Transitions come as *external*, *enter*, *exit*, and *internal* transitions. External transitions lead from state to state, which are called the *source* or the *target* of the transition, respectively; enter, exit, and internal transitions are always associated with an ordinary state, which is called the *host* of the transition. A *hosted* transition is an enter, exit, or internal transition; a *boundary transition* is an enter or exit transition. As partly shown in the examples, external transitions can be triggered or non-triggered and with or without specs.

A unary member function `add` is provided as part of all transition types to enable adding specs to transitions already defined, like so:

```
// Defining a transition with a programmatic name of
// helloWorld_INIT_TO_aboutToSayHello (see macro definition)
FSM_AUTO(printer_INIT, aboutToPrint);

// Adding an action spec to the transition
helloWorld_INIT_TO_aboutToSayHello
.add(Action([]{printf("%s\n", "Hello world!\n")}));
```

Member function `add` has an equivalent in the form of a `<<`–operator, whence specs can be added to transitions like so:

```
helloWorld_INIT_TO_aboutToSayHello
<< Action([]{printf("%s\n", "Hello world!\n")});
```

Both `add` and the `<<`–operator return a reference to `this`, which allows putting additions in sequence, like so. . .

```
helloWorld_INIT_TO_aboutToSayHello
.add(Action([]{printf("%s\n", "Hello world!")}))
.add(Action([]{printf("%s\n", "Hello hello!")}));
```

. . . or so:

```
helloWorld_INIT_TO_aboutToSayHello
<< Action([]{printf("%s\n", "Hello world!")})
<< Action([]{printf("%s\n", "Hello hello!")});
```

Analogously to external transitions, internal ones can be triggered or non-triggered. The difference between loops on the one hand, that is to say, external transitions where the source and the target are the same and internal transitions on the other hand consists of loops leading to entering and exiting the state in question. Internal transitions do not induce any entering or exiting of their host. A triggered external or internal transition is called a *step transition*, a non–triggered one is called an *auto transition*. Table 4 summarizes what macros and constructors there are for defining transitions. Triggers are always of type `Signal const&`, source states of type `SourceState const&`, target states of type `TargetState const&`, host states of type `State const&`, where types `SourceState`, `TargetState`, and `State` are given as indicated in Table 1.

Table 4: Macros and constructors for defining transitions.

Macro ↦ The corresponding constructor
`FSM_STEP(`*`source`*`, `*`target`*`)` ↦ `Step const` *`source`*`_TO_`*`target`*`(`*`trigger`*`, `*`source`*`, `*`target`*`)`
`FSM_AUTO(`*`source`*`, `*`target`*`)` ↦ `Auto const` *`source`*`_TO_`*`target`*`(`*`source`*`, `*`target`*`)`
`FSM_ENTER(`*`host`*`)` ↦ `Enter const` *`host`*`_ENTER(`*`host`*`)`
`FSM_EXIT(`*`host`*`)` ↦ `Exit const` *`host`*`_EXIT(`*`host`*`)`
`FSM_INTERNAL_STEP(`*`host`*`)` ↦ `InternalStep const` *`host`*`_INTERNAL_STEP(`*`trigger`*`, `*`host`*`)`
`FSM_INTERNAL_AUTO(`*`host`*`)` ↦ `InternalAuto const` *`host`*`_INTERNAL_AUTO(`*`host`*`)`

All transitions macros and constructors can have an arbitrary number of specs as additional parameters, in which case they go after the parameters shown in the table. The only mandatory specs are triggers on triggered transitions. Then, the source and the target in an external transition need not belong to the same region. The target may rather belong to any region that sits, in a direct line of descent, above or below the region to which the source belongs. An equivalent way of putting that consists of saying that external

transitions must never cross any concurrency boundaries.

An important distinction to make is that between regions being entered *implicitly* or *explicitly*. A region is entered implicitly when its parent state is entered without the region directly taking part in it; it is entered explicitly if one of the region's sub-states is entered from above via a transition that enters the region's parent state by crossing the state boundary and then continuing into the region. As a consequence of this definition, crossing a state boundary from above entails implicitly entering all sibling regions of the region being targeted.

Dually to regions being entered implicitly or explicitly, regions can also be exited implicitly or explicitly. A region is exited implicitly when its parent state is exited without the region directly taking part in it; it is exited explicitly if one of the region's sub–states is exited via a transition exiting the sub–state and then leaving the region by crossing the boundary of the parent state. As a consequence of this definition, crossing a state boundary from below entails implicitly exiting all sibling regions of the region from within which the transition originates.

Transitions that run in the upward direction are like exceptions in that they lead to a state machine's tree–like activation "stack" being unwound. The unwinding proceeds by exiting regions as described in the previous paragraph. It stops once the target region of the transition is reached. Upward transitions are implemented by means of throwing real exceptions, usually incurring a significant speed penalty provided that exceptions and stack unwinding are implemented in some standard manner. This way of implementing upward transitions was chosen consciously. It allows State Diagram to be optimized for executing horizontal and downward transitions. Practical applications of ordinary state machines usually depend on these two types of transitions much more than on upward transitions, probably because upward transitions really tend to be used like exceptions. In addition to that, formal verification is much easier to implement if upward transitions are completely absent. Users are encouraged to avoid upward transitions in view of formal verification functionality to be added to State Diagram in the future.

6 Signal Data / Transition Specs

At the point of definition, signals have to be declared either `void` or data–carrying. Data–carrying signals then provide typed member functions `set` and `get` as well as a `()`–operator that is equivalent to `set`. Data–carrying signals that are referred to as triggers in transition specs appear as `const` references to type `Event` or `LocalEvent`, where `const` references to type `Event` may refer to any signal and `const` references to type `LocalEvent` refer to local signals. Signals when referred to as events do not carry any overt data type information. To set and retrieve their data in a type–safe manner, `set` and `get` have to be endowed with the data type as a template argument. The following example shows most of these mechanisms at work.

```
FSM_TOP(M);
FSM_SIGNAL(string, A, M);
FSM_SIGNAL(string, B, M);

FSM_INIT(M);
FSM_STATE(S, M);
FSM_FINAL(M);

FSM_AUTO(M_INIT, S);
FSM_STEP(S, M_FINAL);
S_TO_M_FINAL
  << Trigger(A)
  << Trigger(B)
  << Action([](Event const& trigger){printf("%s\n",
trigger.get<string>().c_str());});

M.init();
M.step(A(string("Hello!")));

M.init();
M.step(B(string("Good bye!")));
```

Specs that are parameterized by an event reference can solely be used on (triggered) step transitions. In any (attempted) execution of a step transition, a reference to the trigger event used in the execution becomes the argument of any parameterized spec that belongs to the transition. In this way data from the signal behind the event can be transmitted to those specs. This discipline encompasses parameterized guard specs, in which case data of attempted triggers are used to evaluate those guard specs before the transition actually becomes enabled. Table 6 summarizes what transition specs there are and how they may be parameterized. Please note that there are two output spec constructors that use a macro `FSM_LEV`. These two kinds of specs are explained below.

Table 5: Transition specs.

Type of spec	Constructor parameter
`Trigger`	`Event const&`
`Guard`	`function<bool()> const&`
`Guard`	`function<bool(Event const&)> const&`
`Output`	`LocalSignal const&`
`Output`	`function<LocalEvent const&()> const&`
`Output`	`function<FSM_LEV()> const&`
`Output`	`function<LocalEvent const&(Event const&)> const&`
`Output`	`function<FSM_LEV(Event const&)> const&`
`Action`	`function<void()> const&`
`Action`	`function<void(Event const&)> const&`
`FreezeFlag(SHALLOW)`	
`FreezeFlag(FULL)`	
`Max1Flag()`	
`CompletionFlag()`	

As per commonly adopted convention, a step transition needs just one of its triggers activated for the transition to be enabled. For any kind of transition, all of its guard functions must evaluate to true for it to be enabled. Guard functions are evaluated prior to all other functions. Otherwise, the order in which spec functions are invoked is indeterminate — so that there must never be any reliance on any pre–supposed ordering other than that guards are evaluated first. Outputs serve to activate local events, which means that calling this type of spec an "Output" might be considered somewhat of a misnomer. State Diagram sticks to using that wording because of historical reasons. Also, outputs may be collated by returning a vector of references. To this end, macro `FSM_LEV` is provided, which expands as follows.

`FSM_LEV` $\mapsto$ `std::vector<std::reference_wrapper<LocalEvent const>>`

Use these vectors of references to local events if some of the underlying signals are data–carrying and, at the same time, computing the data involves shared sub–computations. In such a case only one output can emit all of these signals at once instead of repeating sub–computations across different outputs. As a caveat, the vector will probably entail heap allocation, so that is a price to be paid for avoiding that computations get duplicated.

Then, transitions can be configured still further by means of flags:

- Freeze flags (`FreezeFlag(SHALLOW)` and `FreezeFlag(FULL)`) can be added to external transitions. Their effect consists of freezing the source state of the transition once the transition is taken. Freezing means that, on implicit re–enter, sub–regions of the source state have their current states assigned to them according to what states were current when the freeze occurred. In the case of a *shallow freeze*, re–assignment of prior current states occurs only in direct sub–regions of the source state while indirect sub–regions are initialized in the normal way; in the case of a *full freeze*, re–assignment of prior current states occurs in all, that is to say, direct or indirect sub–regions of the source state. Freezes can be overridden by entering target regions explicitly.

  Freeze flags probably make up the biggest difference between State Diagram and UML Statecharts. For better or worse, State Diagram takes the view that it usually

depends on the way a state is exited whether the state is to be frozen or not. UML takes the view that whether a state or region is to be frozen or not is more of an intrinsic property. To this end, UML provides *freeze states*, which serve to mark such states or regions as to be freezed on exit. This concept does not exist in State Diagram.

- Max–1 flags (`Max1Flag()`) can be added to external and internal transitions. Their effect consists of limiting the number of times a transition can execute, as follows. If a transition carries a max–1 flag, then the number of of times the transition can execute during a macro step is limited to just one. Pragmatically, max–1 flags support looping. The following modification of the preceding example demonstrates a max–1 flag in action:

```
FSM_TOP(M);
FSM_SIGNAL(string, A, M);
FSM_SIGNAL(string, B, M);

FSM_INIT(M);
FSM_STATE(S, M);
// Doing away with the final state

FSM_AUTO(M_INIT, S);
// Attaching a max-1 loop to the ordinary state
FSM_INTERNAL_STEP(S);
S_INTERNAL_STEP
  << Max1Flag()
  << Trigger(A)
  << Trigger(B)
  << Action([](Event const& trigger)
 {printf("%s\n", trigger.get<string>().c_str());});

M.init();
M.step(A(string("Hello!")));
M.step(B(string("Good bye!")));
M.step(A(string("Hello again!")));
M.step(B(string("Good bye again!")));
```

- Completion flags (`CompletionFlag()`) can be added to external transitions. Their effect consists of modifying the execution discipline. In the absence of any completion flag, the execution of an external transition originating at state A merely requires that no transition from within A is enabled; in the presence of a completion flag, A is required to have terminated. Completion flags exist to support different patterns of composing states sequentially.

7 Miscelleanous

7.1 Perfect Synchrony / Accumulative Signal Activation

The following properties govern signal activation and how it influences step transitions being enabled or not:

1. One of the trigger signals referred to by a step transition must be active for the transition to be enabled (besides all other conditions required for enabled–ness).

2. There is no other way of testing whether a signal is active, especially not in transition guards. — Another way of putting that consists of saying that State Diagram does not support any negated signal activations in predicates governing the enabled–ness of transitions.

3. As already stated above, signals once activated stay activated over the entire course of a macro step.

Elsewhere, similar concepts have been termed *perfect synchrony* due to the way they provide a certain abstraction from temporality, putting the focus on causality instead. Perfect synchrony simplifies the mathematical treatment of state machines required, at later stages, in adding formal verification functionality to State Diagram. It makes pragmatic sense, too, in that ease of mathematical modeling goes hand–in–hand with ease of comprehension.

7.2 More on Signal Data / Data Variables

In keeping with perfect synchrony, signal data, once put in place during any macro step (by means of `set` or the `()` operator), must not be put in place again during that same macro step. Also, signal data must always be put in place before a signal becomes activated, and they must never be retrieved (by means of `get`), before they have been put in place. The idea behind these restrictions consists of being able to treat signal data in an accumulative fashion just like signal activation.

For these reasons, Signals cannot be used like variables, that is to say, to store data for use in subsequent macro steps. Ordinary program variables could be used to that effect, however, State Diagram provides *data variables*, *variables* for short, that enforce perfect synchrony. Employing State Diagram variables is not mandatory — they are just an option at the present stage of State Diagram development. They will facilitate formal verification once that has been added to State Diagram. Variables come as external and local variables as shown in Table 6.

Table 6: Macros and constructors for defining data variables.

Macro ↦ The corresponding constructor
`FSM_VAR(type, name, parent[, init])` `↦ ExternalVar<type> name("name", parent[, init])`
`FSM_LOCAL_VAR(type, name, parent[, init])` `↦ LocalVar<type> name("name", parent[, init])`

Data variables may be initialized by supplying an initial value as also shown in the table.

Data variables provide a `set` and a `get` function and a `<<`–operator that is equivalent to `set`. Calling these functions must adhere to similar ordering constraints as in the case of their namesakes that belong to class `Signal`, too. About the only — and crucial — difference consists of values being kept after a macro step has been completed — so that calling `get` is nearly always well-defined after a macro step has commenced. A `set` operation on the variable in question having occurred at least once. Without any initial call to `set`, `get`ting the variable's data value results in an error being thrown. A local variable that has been set looses this status, though not its value, once local control flow exits its scope — so that the variable can be re–set during the same macro step[1].

There exists an additional function on State Diagram variables, `setNxt`, that takes effect only once the macro step commences that directly follows the one during which it is called. This function and function `get` do not interfere with each other. A typical usage pattern of `set`, `get` and `setNxt` consists of using `set` to initialize a variable, followed by only ever using `get` and `setNxt` thereafter. Calling `setNxt` more than once during one and the same macro step leads to an error being thrown. Lastly, a next–step equivalent to the `<<`–operator is provided in the form of a member `nxt` with a `<<`–operator defined on it — so that `.set(`*`value`*`)` and `.setNxt(`*`value`*`)` can be written as `<<` *`value`* or `.nxt <<` *`value`*, respectively.

7.3 Arrays of Data Variables

Arrays of data variables can be defined as shown in Table 7.

Table 7: Macros and constructors for defining arrays of data variables.

Macro ↦ The corresponding constructor
`FSM_ARRAY(`*`type`*`, `*`name`*`, `*`size`*`, `*`parent`*`[, `*`init`*`])` ↦ `ExternalArray<`*`type`*`, `*`size`*`> `*`name`*`("`*`name`*`", `*`parent`*`[, `*`init`*`])`
`FSM_LOCAL_ARRAY(`*`type`*`, `*`name`*`, `*`size`*`, `*`parent`*`[, `*`init`*`])` ↦ `LocalArray<`*`type`*`, `*`size`*`> `*`name`*`("`*`name`*`", `*`parent`*`[, `*`init`*`])`

Remarks:

1. Just like with local data variables, the parent of a local array of data variables can be a compound state or a region. The given parent becomes the parent of every data variable that belongs to the array.

2. Arrays of data variables can be initialized at the point of definition. Technically, the initializer has to be a reference to a const instance of a `std::array<type, size>`. This requirement entails that the initializer can be given syntactically as a comma–separated sequence of initial values enclosed in curly brackets. An example of that would be an initializer of `{1, 2, 3, 4}` in case the array were of type `int` and size `4`.

[1] In formal verification this feature leads to what is known as the *reincarnation problem*.

3. The usual bracket operator serves to dereference individual data variables, as in `a[0]` given that `a` is an array of such.

7.4 Polyadic Signals

Polyadic signals can be declared as shown in Table 8. Making use of one of the convenience macros, `FSM_SIGNAL` or `FSM_LOCAL_SIGNAL`, requires making use of another auxiliary macro, `FSM_TT`, to declare the actual tuple type that is to serve as the signal payload.

Table 8: Macros and constructors for defining polyadic signals.

Macro ↦ The corresponding constructor
`FSM_SIGNAL(FSM_TT<`t_0`,...,`t_{k-1}`>, name, parent)` ↦ `ExternalSignal<`t_0`,...,`t_{k-1}`> const name("name", parent)`
`FSM_LOCAL_SIGNAL(FSM_TT<`t_0`,...,`t_{k-1}`>, name, parent)` ↦ `LocalSignal<`t_1`,...,`t_k`> const name("name", parent)`
(t_0`,...,`t_{k-1} abbreviates $type_0$`,...,`$type_{k-1}$ with $k \geq 2$)

When referring to a polyadic signal as an event, retrieving the signal's data contents requires adding type parameters to `get` to make the polyadicity explicit. It also requires appying an additional `get` to retrieve any individual data item. The latter `get` has to be parameterized with the data item's index, like so:

$$trigger\texttt{.get<}type_0\texttt{,...,}type_{k-1}\texttt{>().get<}i\texttt{>()}$$

A polyadic signal is emitted by applying a polyadic `()`–operator, like so:

$$signal\texttt{(}expr_0\texttt{,...,}expr_{k-1}\texttt{)}$$

7.5 Object Ownership

Explicitly created, API–level states, transitions, signals, and variables are never destroyed from within State Diagram. Implicitly created default regions and elements of external arrays reside at the API–level too in the sense that their types are API–level. Nevertheless, their ownership rests with the region parent or external array, respectively. Destruction of explicitly created, API–level state machine components must never occur before the entire state machine can be torn down (the reason of which being that State Diagram employs the pimpl–pattern to let API–level components own behind–the–scenes components that do nearly all of the "heavy lifting").

7.6 Errors

State Diagram signals errors by throwing instances of sub–classes of class `Error`. A `std::string` describing any error thrown can be obtained by calling member function `what()`; descriptive data items associated with the error can be obtained

from const fields that are specific per sub–class. Please consult the header file, `state_diagram/state_diagram.h`, and its HTML documentation to learn about what types of errors there are and what data they provide. A typical pattern of catching errors at design time consists of wrapping the state machine in question in a `try–catch` clause, like so:

```
try
{
  ... // Regular State Diagram stuff
}
catch (Error const & err)
{
  std::cerr << err.what();
  ... // Doing whatever else is required
}
```

7.7 Compile–Time Flags

State Diagram provides a number of compile–time flags. They can be used to modify how state machines behave, and also what the API looks like, as follows.

1. Defining `STATE_DIAGRAM_NO_RUNTIME_SHUFFLING` makes State Diagram forego injecting a certain amount of randomness into how it steps through state machines. Usually, there are degrees of freedom with respect to the order in which simultaneously enabled transitions from concurrent regions etc. can be executed. State Diagram normal mode of operation includes randomization to actuate part of this leeway though not all of it (because that would be to cumbersome). This feature helps obeying the UML guideline that was already mentioned herein, which says that execution order must not be relied upon whenever there is any leeway at all. Defining flag `STATE_DIAGRAM_NO_RUNTIME_SHUFFLING` disables all of this randomization at the benefit of reducing computational burdens on any state machines compiled with it. It should usually be defined only in a deployment phase after adherence to the UML guideline has been ensured by following proper development methodology.

2. Defining `STATE_DIAGRAM_NO_RUNTIME_CHECKS` switches off checks performed in initializing (member function `init`) and stepping through (member function `step`) state machines. Like the previous flag, this one should usually be defined only in a deployment phase once the absence of State Diagram–specific runtime errors has been made sure.

3. Defining `STATE_DIAGRAM_STRINGLESS` modifies the code base so as to remove type `std::string` from it. All name parameters are dropped, as well as all error messages of string type that come with exceptions. This flag serves to reduce the size of compiled State Diagram code. Like the previous two flags, it should be used in a deployment phase. Constructors that carry a name parameter loose it if the flag is defined, while the corresponding macros keep their name parameter. These macros are simply redefined so as to drop the name parameter in expanding to the now–redefined constructor. Hence, if user–level State Diagram code uses macros to declare named entities, as in `TOP(awsomeStateMachine)`, then defining

`STATE_DIAGRAM_STRINGLESS` does not require any code changes, whence you can still write `TOP(awsomeStateMachine)`.

If `STATE_DIAGRAM_STRINGLESS` is defined, then State Diagram exceptions acquire `code`, which is a const member of type `int`. This field's value is unique per type of exception. Please consult the header files, `state_diagram_error.h` and `state_diagram.h`, to find out which codes exist and what their identifiers are.

All of these flags are to be defined prior to including the State Diagram header file.

8 A Somewhat Larger Example: Programming a Microwave Oven

The following example provides a somewhat idealized account of how to program a microwave oven using State Diagram. The example starts off with a number of constants that are related to cooking time spans and watt figures. Setting the cooking duration in the seconds range is limited to a maximum of 59 seconds (`maxSecondsSettable`). In the the minutes range it is limited to a maximum of 30 minutes (`maxMinutesSettable`), 30 minutes in seconds also being the limit to the cooking duration overall (`maxSeconds`). In the seconds range the step size in dialing in the cooking duration is five seconds (`secondsStep`). The power output can be dialed in with a minimum of 150 watts (`minWattsSettable`), a maximum of 750 watts (`maxWattsSettable`), and a step size of 150 watts (`wattsStep`).

```
int16_t constexpr secondsPerMinute{60};
int16_t constexpr maxMinutesSettable{30};
int16_t constexpr maxSecondsSettable{secondsPerMinute - 1};
int16_t constexpr minSecondsSettable{0};
int16_t constexpr maxSeconds{maxMinutesSettable * secondsPerMinute};
int16_t constexpr secondsStep{5};

int16_t constexpr maxWattsSettable{750};
int16_t constexpr minWattsSettable{150};
int16_t constexpr wattsStep{150};
```

The example continues with a hypothetical low–level API. This API is assumed to reside outside the state machine. It consists of several functions that are given as lambdas, the function names being self–explanatory. The API provides the interface between the state machine and the microwave hardware. Ellipses appear where hardware–dependent code would appear in any concrete instantiation of the microwave example. Please note that these API calls form the entire interface that the microwave state machine possesses apart from functions `init` and `step` which are to called on the top state. Testing the microwave state machine could, thus, be accomplished by (a) replacing the ellipses with appropriate tracing code and (b) subjecting the state machine to `init`/`step` stimuli to test whether traces come out as expected.

```
auto const doorIsShut{[&]()->bool{...}};
auto const soundBeep{[&]{...}};
auto const turnOnDisplay{[&]{...}};
auto const displayMinutesSeconds{[&](int16_t const minutesSet,
int16_t const secondsSet){...}};
auto const displayWatts{[&](int16_t const wattsSet){...}};
auto const turnOffDisplay{[&]{...}};
auto const turnOnMagnetron{[&](int16_t const wattsSet){...}};
auto const turnOffMagnetron{[&]{...}};
auto const turnOnTurntable{[&]{...}};
auto const turnOffTurntable{[&]{...}};
auto const turnOnLight{[&]{...}};
auto const turnOffLight{[&]{...}};
```

On to the state machine itself, it reacts to eight external signals as shown next. Five of these signals correspond to buttons that are assumed to be available on the outside of the microwave. The cooking duration can be dialed up (signal `durationPlus`) or down (signal `durationMinus`), as can be the power output (signals `wattsPlus` and `wattsMinus`). A start/stop button is also present (signal `startStop`). Then, the microwave senses the door to the cooking chamber being opened (signal `doorOpen`) or closed (signal `doorShut`). Lastly, there is recurring tick (signal `tick`) that is assumed to be signaled each second. Technically, there has to be an external loop that invokes function `step` on `microwave`, the argument to `step` being the signal that corresponds to the event that the microwave is to react to next. The way the state machine is written it assumes that `step` will only ever be called with one signal argument at a time.

```
FSM_TOP(microwave);

FSM_SIGNAL(void, durationPlus, microwave);
FSM_SIGNAL(void, durationMinus, microwave);
FSM_SIGNAL(void, wattsPlus, microwave);
FSM_SIGNAL(void, wattsMinus, microwave);
FSM_SIGNAL(void, startStop, microwave);
FSM_SIGNAL(void, doorOpen, microwave);
FSM_SIGNAL(void, doorShut, microwave);
FSM_SIGNAL(void, tick, microwave);
```

Just below the top level, the microwave has states `microwave_INIT`, `standby`, and `on`. Once initialized, an auto transition puts the state machine into `standby` upon calling `step`. Calling step `step` once more with any external signal but `tick` puts the state machine into `on`. Completing a cooking process corresponds to state `on` terminating, whence transitions emanating at `on` carry a completion flag. The state machine transitions from `on` back to `standby` in case external signal `tick` arrives. It loops from `on` to `on` in case any other external signal arrives.

```
FSM_INIT(microwave);
FSM_STATE(standby, microwave);
FSM_STATE(on, microwave);

FSM_AUTO(microwave_INIT, standby);
FSM_STEP(on, standby, Trigger(tick), CompletionFlag());
FSM_STEP
(
  standby
, on
, Trigger(startStop)
, Trigger(durationPlus)
, Trigger(durationMinus)
, Trigger(wattsPlus)
, Trigger(wattsMinus)
, Trigger(doorOpen)
);
```

Most of the statemachine sits below state `on`. There are two local variables, `seconds` and `watts`. They hold the current count–down value or the current power setting, respectively. These variables are (re–initialized) whenever state `on` is entered.

```
FSM_LOCAL_VAR(int16_t, seconds, on);
FSM_LOCAL_VAR(int16_t, watts, on);

FSM_ENTER
(
  on
, Action([&]{
    seconds << minSecondsSettable;
    watts << minWattsSettable;
  })
);
```

Updates to local variables `seconds` and `watts` always take effect once the next macro step commences, using `nxt`–syntax. Other parts of the state machine need to be notified of any such update already during the current instant. To this end, two local signals, `secondsNxtSet` and `wattsNxtSet`, are attached to state `on` in addition to variables `seconds` and `watts`:

```
FSM_LOCAL_SIGNAL(int16_t, secondsNxtSet, on);
FSM_LOCAL_SIGNAL(int16_t, wattsNxtSet, on);
```

Signals `durationPlus`, `durationMinus`, `wattsPlus`, and `wattsMinus` are reacted–to by means of four transitions that are internal to state `on`, each one having one of these signals as its trigger. Each one of these transitions is also guarded by the countdown value having to be grater than zero since it being zero means that state `on` needs to terminate. Each one of these transitions also has a `Max1Flag`. As for dialing the countdown value up or down, there are two step lengths to this, 5 or 60 seconds. The changeover threshold lies at 60 seconds (the same as the bigger step length). The smaller step length is used for adjustments below this threshold. Newly dialed in countdown values are also normalized modulo the step size — so that they always come out to be 0, 5, 10, . . . , 50, 55, 60, 120, 180, . . . , 1680, 1740, or 1800.

```
InternalStep const on_ON_durationPlus
{
  on
, Trigger(durationPlus)
, Guard([&]{return seconds.get() > 0;})
, Output([&]()->LocalSignal<int16_t> const&{
    int const valSecondsNxt{[&]{
      if (seconds.get() < secondsPerMinute)
      {
        int const rawValSecondsNxt{seconds.get() + secondsStep};
        return rawValSecondsNxt - (rawValSecondsNxt % secondsStep);
      }
      int const rawValSecondsNxt{seconds.get() + secondsPerMinute};
      return rawValSecondsNxt <= maxSecondsSettable ?
        rawValSecondsNxt - (rawValSecondsNxt % secondsPerMinute) :
        maxSecondsSettable;
    }()};
    seconds.nxt << static_cast<int16_t>(valSecondsNxt);
    return secondsNxtSet(static_cast<int16_t>(valSecondsNxt));
  })
, Max1Flag()
```

```
};
InternalStep const on_ON_durationMinus
{
  on
, Trigger(durationMinus)
, Guard([&]{return seconds.get() > 0;})
, Output([&]()->LocalSignal<int16_t> const&{
    int const valSecondsNxt{[&]{
      if (seconds.get() <= secondsPerMinute)
      {
        if (seconds.get() <= secondsStep)
        {
          return static_cast<int>(seconds.get());
        }
        if ((seconds.get() % secondsStep) == 0)
        {
          return seconds.get() - secondsStep;
        }
        return seconds.get() - (seconds.get() % secondsStep);
      }
      else
      {
        if ((seconds.get() % secondsPerMinute) == 0)
        {
          return seconds.get() - secondsPerMinute;
        }
        return seconds.get() - (seconds.get() % secondsPerMinute);
      }
    }()};
    seconds.nxt << static_cast<int16_t>(valSecondsNxt);
    return secondsNxtSet(static_cast<int16_t>(valSecondsNxt));
  })
, Max1Flag()
};
InternalStep const on_ON_wattsPlus
{
  on
, Trigger(wattsPlus)
, Guard([&]{return seconds.get() > 0;})
, Output([&]()->LocalSignal<int16_t> const&{
    int const rawValWattsNxt{watts.get() + wattsStep};
    int const valWattsNxt{
      rawValWattsNxt <= maxWattsSettable ?
      rawValWattsNxt :
      maxWattsSettable
    };
    watts.nxt << static_cast<int16_t>(valWattsNxt);
    return wattsNxtSet(static_cast<int16_t>(valWattsNxt));
  })
, Max1Flag()
};
InternalStep const on_ON_wattsMinus
{
  on
, Trigger(wattsMinus)
```

```
, Guard([&]{return seconds.get() > 0;})
, Output([&]()->LocalSignal<int16_t> const&{
    int const rawValWattsNxt{watts.get() - wattsStep};
    int const valWattsNxt{
      rawValWattsNxt >= minWattsSettable ?
      rawValWattsNxt :
      minWattsSettable
    };
    watts.nxt << static_cast<int16_t>(valWattsNxt);
    return wattsNxtSet(static_cast<int16_t>(valWattsNxt));
  })
, Max1Flag()
};
```

Making up the microwave oven's "business end", the cooking chamber is next. It is controlled via a distinct region, `cookingChamber`, that sits directly below state `on`. The only states that require explanation are `stopped` and `paused`: The cooking process stops whenever the start/stop button is pressed; it pauses whenever the door is opened.

```
FSM_REGION(cookingChamber, on);
FSM_INIT(cookingChamber);
FSM_STATE(stopped, cookingChamber);
FSM_STATE(paused, cookingChamber);
FSM_STATE(cooking, cookingChamber);
FSM_FINAL(cookingChamber);
```

Region `cookingChamber` signals to other regions below `on` whether the cooking process starts, stops temporarily, or exits altogether. These signals must be local to state `on` for them to be visible in these other regions.

```
FSM_LOCAL_SIGNAL(void, start, on);
FSM_LOCAL_SIGNAL(void, stop, on);
FSM_LOCAL_SIGNAL(void, exit, on);
```

Transitions in region `cookingChamber` also make use of two auxiliary functions, which are given as lambdas.

```
auto startCooking{[&]{
  turnOnTurntable();
  turnOnMagnetron(watts.get());
}};
auto stopCooking{[&]{
  turnOffMagnetron();
  turnOffTurntable();
}};
```

There are six cooking chamber transitions, one of them being an internal step of state `cooking`. Transitions that emanate from state `stopped` make use of API predicate `doorIsShut` in their guard condition. It is assumed that this predicate behaves as one would expect with regard to which one of signals `doorOpen` and `doorShut` has been emitted last.

```
FSM_AUTO(cookingChamber_INIT, stopped);
FSM_STEP
(
  stopped
, cooking
, Trigger(startStop)
, Guard([&]{return doorIsShut() && (seconds.get() >= 1);})
, Action([&]{startCooking();}), Output(start)
);
FSM_STEP
(
  stopped
, cookingChamber_FINAL
, Trigger(startStop)
, Guard([&]{return seconds.get() == 0;})
, Action([&]{soundBeep();})
, Output(exit)
);
FSM_STEP
(
  paused
, cooking
, Trigger(startStop)
, Guard([&]{return doorIsShut() && (seconds.get() >= 1);})
, Action([&]{startCooking();})
, Output(start)
);
FSM_STEP
(
  paused
, cookingChamber_FINAL
, Trigger(startStop)
, Guard([&]{return seconds.get() == 0;})
, Action([&]{soundBeep();})
, Output(exit)
);
Step const cooking_TO_paused_BY_startStop
(
  cooking
, paused
, Trigger(startStop)
, Action([&]{stopCooking();})
, Output(stop)
);
Step const cooking_TO_paused_BY_doorOpen
(
  cooking
, paused
, Trigger(doorOpen)
, Action([&]{stopCooking();})
);
FSM_INTERNAL_STEP
(
  cooking
, Trigger(tick)
```

```
, Guard([&]{return seconds.get() >= 2;})
, Output([&]()->LocalSignal<int16_t> const&{
    int16_t const valSecondsNxt{seconds.get() - 1};
    seconds.nxt << valSecondsNxt;
    return secondsNxtSet(valSecondsNxt);
  })
, Max1Flag()
);
FSM_STEP
(
  cooking
, cookingChamber_FINAL
, Trigger(tick)
, Guard([&]{return seconds.get() <= 1;})
, Action([&]{
    seconds.nxt << static_cast<int16_t>(seconds.get() - 1);
    stopCooking();
    soundBeep();
  })
, Output(exit)
);
```

The second region below state `on` controls the light bulb that illuminates the cooking chamber. There are four states and six transitions to this region.

```
FSM_REGION(light, on);
FSM_INIT(light);
FSM_STATE(lightOff, light);
FSM_STATE(lightOn, light);
FSM_FINAL(light);

FSM_STEP
(
  light_INIT
, lightOn
, Trigger(start)
, Trigger(doorOpen)
, Action([&]{turnOnLight();})
);
FSM_STEP(light_INIT, light_FINAL, Trigger(exit));
FSM_STEP
(
  lightOff
, lightOn
, Trigger(start)
, Trigger(doorOpen)
, Action([&]{turnOnLight();})
);
FSM_STEP(lightOff, light_FINAL, Trigger(exit));
FSM_STEP
(
  lightOn
, lightOff
, Trigger(stop)
, Trigger(doorShut)
```

```
, Action([&]{turnOffLight();})
);
FSM_STEP
(
  lightOn
, light_FINAL
, Trigger(exit)
, Action([&]{turnOffLight();})
);
```

State `on` has one more region below itself. This region controls the display that provides information on the cooking duration and the power setting. There are three states and six transitions to this region.

```
FSM_REGION(display, on);
FSM_INIT(display);
FSM_STATE(displayOn, display);
FSM_FINAL(display);

FSM_AUTO(display_INIT, displayOn);
FSM_ENTER
(
  displayOn
, Action([&]{
    turnOnDisplay();
    displayMinutesSeconds(
      seconds.get() / secondsPerMinute
    , seconds.get() % secondsPerMinute
    );
    displayWatts(watts.get());
  })
);
InternalStep const displayOn_INTERNAL_ON_secondsNxtSet
(
  displayOn
, Trigger(secondsNxtSet)
, Action([&](Event const& secondsNxtSet){
    displayMinutesSeconds(
      secondsNxtSet.get<int16_t>() / secondsPerMinute
    , secondsNxtSet.get<int16_t>() % secondsPerMinute
    );
  })
, Max1Flag()
);
InternalStep const displayOn_INTERNAL_ON_wattsNxtSet
(
  displayOn
, Trigger(wattsNxtSet)
, Action([&](Event const& wattsNxtSet){
    displayWatts(wattsNxtSet.get<int16_t>());
  })
, Max1Flag()
);
FSM_EXIT(displayOn, Action([&]{turnOffDisplay();}));
FSM_STEP(displayOn, display_FINAL, Trigger(exit));
```

9 What's New

January 20, 2021:

1. More generous layout.
2. Resource–saving request.
3. Corrections to the introduction.
4. Correction to what is written regarding heap usage.
5. Other minor editing.

January 19, 2021:

1. Removing C++17 flag.
2. Introducing `STATE_DIAGRAM_STRINGLESS` flag.
3. Slightly more comprehensive introduction, minor section rearrangement, other minor editing.

December 25, 2020:

1. Adding a graphic that depicts the hello–world state machine.
2. Minor updates and modifications.

November 19, 2020:

1. Minor corrections.

August 13, 2020:

1. First version of this manual.