mbeddr C User Guide

This document focuses on the C programmer who wants to exploit the benefits of the extensions to C provided by mbeddr out of the box. We assume that you have some knowledge of regular C (such asK&R C, ANSI C or C99). We also assume that you realize some of the shortfalls of C and are "open" to the improvements in mbeddr C. The main point of mbeddr C is the ability to extend C with domain-specific concepts such as state machines, components, or whatever you deem useful in your domain. We have also removed some of the "dangerous" features of C that are often prohibited from use in real world projects.

This document does not discuss how to develop new languages or extend existing languages. We refer to the *Extension Guide* instead. It is available from **http://mbeddr.com**.

Contents

Part I

Comprehensive Tutorial

1 Tutorial

This tutorial showcases many of the features of mbeddr in an integrated example. The sources ZIP **com.mbeddr.tutorial.zip** is available from the download page at **mbeddr.com**. It is also part of the complete distro package. In the git repository the sources can be found in the **code/applications/tutorial** folder. Simply open the **.mpr** file to play with the example.

Notice that the tutorial does not discuss every aspect of every mbeddr extension — please refer to the respective chapters in the user guide.

This tutorial assumes that you have successfully run through the Hello World tutorial in Section **??**. If you haven't, you can use one of the Wizards in the **Code** menu to create a **Minimal Test Case**. From this, the starting point for our tutorial looks as follows:

- We have a default type size configuration.
- We have a build configuration with one **Executable**.
- We have an **ImplementationModule** with a **main** function.

1.1 Function Pointers

1.1.1 The Basic Program

As the first example, we will add a configurable event handler using function pointers. We create a new module **FunctionPointers** using the context menu **New -> c.m.core.module -> ImplementationModule** on the current model.

Inside it, we will add a **struct** called **DataItem** that contains two members. You create the **struct** by just typing **struct** inside the module. You add the members by simply

starting to type the **int8** types.

```
struct DataItem {
  int8 id;
};
```

We then create two functions that are able to process the **DataItem**s. Here is one function that does nothing (intentionally). You enter this function by starting out with the **DataItem** type, then typing the name and then using the **(** to actually create the function (the thing has been a global variable up to this point!):

```
DataItem process_doNothing(DataItem e) {
  return e;
}
```

Other functions with the same signature may process the data in some specific way; We can generalize those into a function type using a **typedef**. Note that entering the function type **()=>()** is in fact a little bit cumbersome. The alias for entering it is **funtype**:

```
typedef (DataItem)=>(DataItem) as DataProcessorType;
```

We can now create a global variable that holds an instance of this type and that acts as a global event dispatcher. We also create a new, empty **test case** that we will use for making sure the program actually works. In the test we assign a reference to **process_doNothing** to that event handler.

```
DataProcessorType processor;
exported test case testProcessing {
  processor = :process_doNothing;
}
```

We can now write the first simple test:

```
exported test case testProcessing {
  processor = :process_doNothing;
  DataItem i1;
  i1.id = 42;
  DataItem i2 = processor(i1);
  assert(0) i2.id == 42;
}
```

Let us complete this into a runnable system. In the **Main** module we change our **main** function to run our new test. Note how we import the **FunctionPointers** module; we call the test case, which is visible because it is **exported**:

```
module Main imports FunctionPointers {
  exported int32 main(int32 argc, string*[] argv) {
    return test testProcessing;
```

```
  }
}
```

Looking at the build configuration we see an error that complains that the binary is inconsistent, because the **FunctionPointers** module is not included. We can fix this with a quick fix. This results in the following binary:

```
executable MbeddrTutorial isTest: true {
  used external libraries
  used mbeddr libraries
  included modules
    Main (mbeddr.tutorial.main.m1)
    FunctionPointers (mbeddr.tutorial.main.m1)
}
```

1.1.2 Building and Running

We can now build the system using **Ctrl-F9** or by selecting **Rebuild** from the context menu of the solution in the logical view on the left. If you have installed **gcc** correctly the binary should actually be compiled automatically. Here is the info message you should get in the MPS messages view:

```
make finished successfully for mbeddr.tutorial.main/mbeddr.tutorial.main.m1
```

Let us run this program from the command line. To get to the respective location in the file system, select the solution in the logical view, open the properties and copy the **Solution File** location. In my case it is:

```
/Users/markusvoelter/Documents/mbeddr/mbeddr.core/code/applications/tutorial/solutions/
    mbeddr.tutorial.main/mbeddr.tutorial.main.msd
```

We can open a console and **cd** to this directory, removing the last segment **mbeddr.tutorial.main.msd** from the path. In that directory we can **cd** to **source_gen**, and from there we can navigate down to the directory for the model: **cd mbeddr/tutorial/m1**

Using **ls** there, we can see the following:

```
$ ls -ll
total 104
-rw-r--r--  1 markusvoelter  staff   1189 Oct 30 21:11 FunctionPointers.c
-rw-r--r--  1 markusvoelter  staff    159 Oct 30 21:11 FunctionPointers.h
-rwxr-xr-x  1 markusvoelter  staff   9028 Oct 30 21:11 Main
-rw-r--r--  1 markusvoelter  staff    338 Oct 30 21:11 Main.c
-rw-r--r--  1 markusvoelter  staff    162 Oct 30 21:11 Main.h
-rw-r--r--  1 markusvoelter  staff    433 Oct 30 21:11 Makefile
drwxr-xr-x  4 markusvoelter  staff    136 Oct 30 21:11 bin
-rw-r--r--  1 markusvoelter  staff    943 Oct 30 21:11 module_dependencies.gv
```

```
-rw-r--r--  1 markusvoelter  staff  14069 Oct 30 21:11 trace.info
```

Importantly, we see a **Makefile**, so we can call **make**. This will build the binary:

```
$ make
rm -rf ./bin
mkdir -p ./bin
gcc -std=c99   -c -o bin/Main.o Main.c
mkdir -p ./bin
gcc -std=c99   -c -o bin/FunctionPointers.o FunctionPointers.c
gcc -std=c99 -o Main bin/Main.o bin/FunctionPointers.o
```

This results in an executable **MbeddrTutorial**. We can run it by calling **./MbeddrTutorial** or by calling **make test**. The output should look as follows:

```
$ make test
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
```

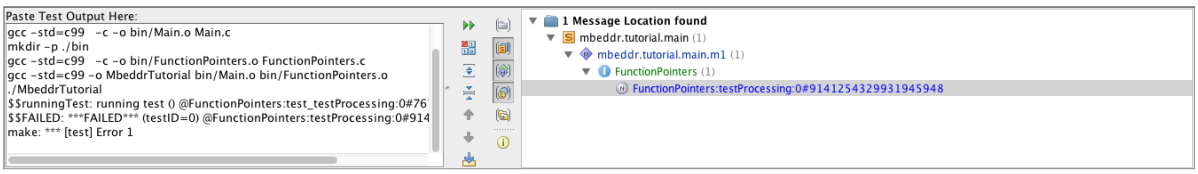
So the test succeeded, everything seems fine. Let us try to introduce an error by somehow breaking the assertion.

```
assert(0) i2.id == 0;
```

After regenerating the code (**Ctrl-F9** or **Rebuild**) we can call **make test** again and we get:

```
$ make test
mkdir -p ./bin
gcc -std=c99   -c -o bin/Main.o Main.c
mkdir -p ./bin
gcc -std=c99   -c -o bin/FunctionPointers.o FunctionPointers.c
gcc -std=c99 -o MbeddrTutorial bin/Main.o bin/FunctionPointers.o
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
$$FAILED: ***FAILED*** (testID=0) @FunctionPointers:testProcessing:0#9141254329931945948
make: *** [test] Error 1
```

As you can see the test fails. It says that in the **FunctionPointers** module, **testProcessing** test case, assert number **0**. You can either navigate to the offending **assert** manually. Alternatively you can copy the console output into the clipboard and then paste it into the text box into the window that opens from the **Analyze -> mbeddr Analyze Error Output** menu:



Double clicking on the respective line opens the editor on the offending **assert** statement.

1.1.3 Lambdas

In contrast to regular C, mbeddr also provides lamdas, i.e. anonymous functions. They can be passed to functions that take function types as an argument. However, they can also be assigned to variables that have a function type, such as the **processor** above. Here is an example:

```
exported test case testLambdaProcessing {
  Trackpoint i1 = {
    id = 1
    timestamp = 0 s
    x = 0 m
    y = 0 m
    alt = 50 m
  };

  processor = [tp|
    tp.alt = 100 m;
    tp;
    ];

  assert(0) processor(i1).alt == 100 m;
}
```

A lamda is expressed as **[arg1, arg2, ...|statements]**. The type of the arguments is inferred from the context, they don't have to be specified. If several statements are required (as in the example above), they are layouted vertically. If only an expression is required, it is shown in line:

```
Trackpoint i1 = {
  ...
  alt = 100 m
};

processor = [tp|tp;];

assert(0) processor(i1).alt == 100 m;
```

1.2 Physical Units

Let us go back to the definition of **DataItem** and make it more useful. Our application is supposed to work with tracking data (captured from a bike computer or a flight logger).

We change the **struct** as follows (note that renaming the thing is trivial — you just change the name!):

```
struct Trackpoint {
  int8 id;        // sequence ID of the track point
  int8 timestamp; // timestamp as taken from GPS time
  int8 x;         // longtitude, simplified as a number
  int8 y;         // latitude, simplified as a number
  int8 alt;       // altitude as of the GPS
  int8 speed;     // current speed, if available
};
```

We can now enhance our test case the following way:

```
exported test case testProcessing {
  Trackpoint i1 = {
    id = 1
    x = 0
    y = 0
    alt = 100
  };
  processor = :process_doNothing;
  Trackpoint i2 = processor(i1);
  assert(0) i2.id == 1 && i2.alt == 100;
```

We can now use physical units to add more semantics to this data structure. We add the **com.mbeddr.physicalunits** devkit to the model properties. Then we can add units to the members. To add a unit, simply press **/** at the right side of one of the **int8** types:

```
struct Trackpoint {
  int8 id;
  int8/s/ timestamp;
  int8/m/ x;
  int8/m/ y;
  int8/m/ alt;
  int8 speed;
};
```

s and **m** are SI base units, so they are available by default. For the **speed** member we need to add **m/s**. Since this is not an SI base unit we first have to define it. To do so we create a new **UnitContainer** root in the current model (you can find the **UnitContainer** in the **c.m.ext.physicalunits** submenu of the **New** context menu on the current model). In it we can create a **derived unit**. Entering the **mps** and **speed** is trivial. Entering the meters per second is a bit cumbersome right now: press **m** for the meters, press **Enter**, press **s** for the seconds and then type **-1** directly on the right side of the **s** to enter the exponent:

```
Unit Configuration
                    -1
derived unit mps = m s   for speed
```

We can now go back to the **Trackpoint** and make the **speed** property use a unit: **int8/mps/ speed;**

Adding these units results in errors in the existing code because you cannot simply assign a plain number to a variable or member whose type includes a physical unit (**int8/m/ length = 3;** is illegal). Instead you have to add units to the literals as well. You can simply type the unit after the literal to get to the following:

```
Trackpoint i1 = {
  id = 1
  timestamp = 0 s
  x = 0 m
  y = 0 m
  alt = 100 m
};
...
assert(0) i2.id == 1 && i2.alt == 100 m;
...
assert(1) i3.id == 1 && i3.alt == 0 m;
```

If you try to rebuild now you will run into build errors:

```
no configuration item "physical units" found in this model. Please add a configuration item
     in your Build Configuration.
```

To fix this we have to go to the build configuration and add the respective configuration item:

```
Configuration Items
  reporting: printf (add labels false)
  physical units (config = Units Declarations (mbeddr.tutorial.main.m1))
```

If we rebuild now, everything should generate normally and we should be able to run the test again. Nothing should have changed so far. However, if we were to write the following code, we would get an error:

```
int8 someInt = i1.x + i1.speed; // error, adding apples and pears
```

The problem with this code is that you cannot add a length (**i1.x**) and a speed (**i1.speed**). And the result is certainly not a plain **int8**, so you cannot assign the result to **someInt**. Adding **i1.x** and **i1.y** will work, though. Also, you can calculate with units to write the following code. This gives you an additional level of type safety.

```
Trackpoint i4 = {
  id = 1
  timestamp = 10 s
  x = 100 m
  y = 0 m
  alt = 100 m
};
int8/mps/ speed = (i4.x - i2.x) / (i4.timestamp - i2.timestamp);
```

1.3 Components

Let us now introduce components to further structure the system. We start by factoring the **Trackpoint** data structure into a separate module and export it to make it accessible from importing modules.

```
module DataStructures imports nothing {
  exported struct Trackpoint {
    int8 id;
    int8/s/ timestamp;
    int8/m/ x;
    int8/m/ y;
    int8/m/ alt;
    int8/mps/ speed;
  };
}
```

1.3.1 An Interface with Contracts

We now define an interface that handles **Trackpoint**s. To be able to do that we have to add the **com.mbeddr.components** devkit to the current model. We can then enter a client-server interface in a new module **Components**. We use pointers for the trackpoints here to optimize performance. Note that you can just press ∗ on the right side of **Trackpoint** to make it a **Trackpoint∗**:

```
module Components imports DataStructures {

  exported cs interface TrackpointProcessor {
    Trackpoint* process(Trackpoint* p);
  }

}
```

To enhance the semantic "richness" of the interface we can add preconditions. To do so, use an intention **Add Precondition** on the operation itself. Please add the following pre- and postconditions (note how you can of course use units in the precondition):

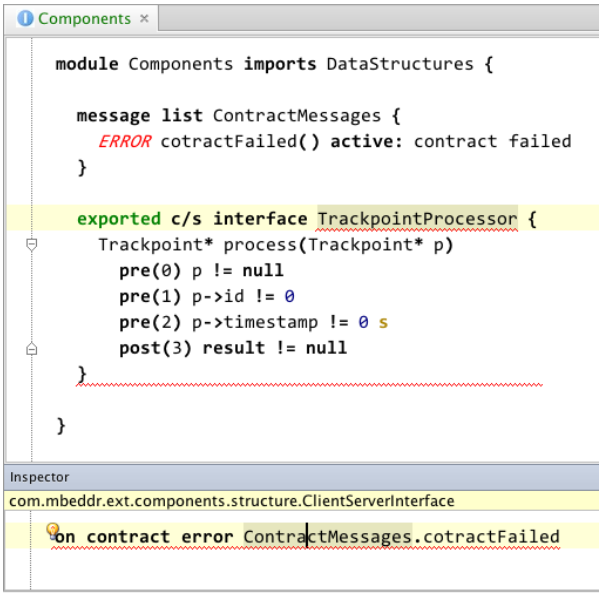
```
Trackpoint* process(Trackpoint* p)
  pre(0) p != null
  pre(1) p->id != 0
  pre(2) p->timestamp != 0 s
  post(3) result->id != 0
```

After you have added these contracts, you will get an error message on the interface. The problem is this: if a contract (pre- or postcondition) fails, the system will report a message (this message can be deactivated in case you don't want any reporting).

However, for the program to work you have to specify a message on the interface. We create a new message list and a messge:

```
messagelist ContractMessages {
  ERROR contractFailed() active: contract failed
}
```

You can now open the inspector for the interface and reference this message from there:



There are still errors. The first one complains that the message list must be exported if the interface is exported. We fix it by exporting the message list (via an intention). The next error complains that the message needs to have to integer arguments to represent the operation and the pre- or postcondition. We change it thusly:

```
exported messagelist ContractMessages {
  ERROR contractFailed(int8 op, int8 pc) active: contract failed
}
```

1.3.2 A First Component

Let us create a new component by typing **component**. We call it **Nuller**. It has one provided port called **processor** that provides the **TrackpointProcessor** interface:

```
exported component Nuller extends nothing {
  provides TrackpointProcessor processor
}
```

We get an error that complains that the component needs to implement the operations defined by the **TrackpointProcessor** interface; we can get those automatically generated by using a quick fix on the provided port. This gets us the following:

```
exported component Nuller extends nothing {
  provides TrackpointProcessor processor
  Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
    return null;
  }
}
```

The **processor_process** runnable is triggered by an incoming invocation of the **process** operation defined in the **TrackpointProcessor** interface. The **Nuller** simply sets the altitute to zero:

```
Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
  p->alt = 0 m;
  return p;
}
```

Let us now write a simple test case to check this component. To do that, we first have to create an instance of **Nuller**. We create an instance configuration that has an instance of this component. Also, we add an adapter. An adapter makes a provided port of a component instance (**Nuller.processor**) available to a regular C program under the specified name **n**:

```
instances nullerInstances extends nothing {
  instance Nuller nuller
  adapt n -> nuller.processor
}
```

Now we can write a test case that accesses the **n** adapter — and through it, the **processor** port of the **Nuller** component instance **nuller**. We create a new **Trackpoint**, using 0 as the **id** — intended to trigger a contract violation (remember **pre(1) p->id != 0**). To enter the **&tp** just enter a **&**, followed by **tp**:

```
exported test case testNuller {
  Trackpoint tp = {
    id = 0
  };
  n.process(&tp);
}
```

Before we can run this, we have to make sure that the **instances** are initialized (cf. the warning you get on them). We do this right in the test case:

```
exported test case testNuller {
  initialize instances;
  Trackpoint tp = {
    id = 0
```

```
  };
  n.process(&tp);
}
```

To make the system work, you have to import the **Components** module into the **Main** module so you can call the **testNuller** test case from the **test** expression in **Main**. In the build configuration, you have to add the missing modules to the executable (using the quick fix). Finally, also in the build configuration, you have to add the **components** configuration item:

```
Configuration Items:
  reporting: printf (add labels false)
  physical units (config = Units Declarations (mbeddr.tutorial.main.m1))
  components: no middleware
              wire statically: false
```

You can now rebuild and run. As a result, you'll get contract failures:

```
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
$$runningTest: running test () @Components:test_testNuller:0#767515563077315487
$$contractFailed: contract failed (op=0, pc=1) @Components:null:-1#1731059994647588232
$$contractFailed: contract failed (op=0, pc=2) @Components:null:-1#1731059994647588253
```

We can fix these problems by changing the test data to conform to the contract:

```
Trackpoint tp = {
  id = 10
  timestamp = 10 s
  alt = 100 m
};
n.process(&tp);
assert(0) tp.alt == 0 m;
```

Let us provoke another contract violation by returning from the implementation in the **Nuller** component a **Trackpoint** whose **id** is 0:

```
Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
  p->alt = 0 m;
  p->id = 0;
  return p;
}
```

Running it again provokes another contract failure. Notice how the contract is specified on the *interface*, but they are checked for each *component* implementing the interface. There is no way how an implementation can violate the interface contract without the respective error being reported!

1.3.3 Collaborating and Stateful Components

Let us look at interactions between components. We create a new interface, the **TrackpointStore**. It can store and return trackpoints[1]. Here is the basic interface:

```
exported cs interface TrackpointStore1 {
  void store(Trackpoint* tp)
  Trackpoint* get()
  Trackpoint* take()
  boolean isEmpty()
}
```

Let us again think about the semantics: you shouldn't be able to get or take stuff from the store if it is empty, you should not put stuff into it when it is full, etc. These things can be expressed as pre- and postconditions. The following should be pretty self-explaining. The only new thing is the **query** operation. Queries can be used from inside pre- and postconditions, but cannot modify state[2]

```
exported cs interface TrackpointStore1 {
  void store(Trackpoint* tp)
    pre(0) isEmpty()
    pre(1) tp != null
    post(2) !isEmpty()
  Trackpoint* get()
    pre(0) !isEmpty()
  Trackpoint* take()
    pre(0) !isEmpty()
    post(1) result != null
    post(2) isEmpty()
  query boolean isEmpty()
}
```

These pre- and postconditions mostly express a valid sequence of the operation calls: you have to call **store** before you can call **get**, etc. This can be expressed directly with protocols:

```
exported cs interface TrackpointStore2 {

  // store goes from the initial state to a new state full
  void store(Trackpoint* tp)
    protocol init(0) -> new full(1)

  // get expects the state to be full, and remains there
  Trackpoint* get()
    protocol full -> full

  // take expects to be full and then becomes empty (i.e. init)
  Trackpoint* take()
    post(0) result != null
    protocol full -> init(0)
```

[1] Sure, it is completely overdone to separate this out into a separate interface/component, but for the sake of the tutorial it makes sense.

[2] Currently this is not yet checked. But it will be.

```
  // and isEmpty has no effect on the protocol state
  query boolean isEmpty()
}
```

The two interfaces are essentially equivalent, and both are checked at runtime and lead to errors if the contract is violated.

We can now implement a component that provides this interface. Most of the following code should be easy to understand based on what we have discussed so far. There are two new things. There is a field **`Trackpoint* storedTP;`** that represents component state. Second there is an **`on-init`** runnable: this is essentially a constructor that is executed as an instance is created.

```
exported component InMemoryStorage extends nothing {
  provides TrackpointStore1 store
  Trackpoint* storedTP;

  void init() <- on init {
    storedTP = null;
  }

  void trackpointStore_store(Trackpoint* tp) <- op store.store {
    storedTP = tp;
  }
  Trackpoint* trackpointStore_get() <- op store.get {
    return storedTP;
  }
  Trackpoint* trackpointStore_take() <- op store.take {
    Trackpoint* temp = storedTP;
    storedTP = null;
    return temp;
  }
  boolean trackpointStore_isEmpty() <- op store.isEmpty {
    return storedTP == null;
  }
}
```

To keep our implementation module **`Components`** well structured we can use sectios. A **`section`** is a named part of the implementation module that has no semantic effect beyond that. Sections can be collapsed.

```
module Components imports DataStructures {

  exported messagelist ContractMessages {...}

  section processor {...}

  section store {
    exported cs interface TrackpointStore1 {
      ...
    }
    exported cs interface TrackpointStore2 {
      ...
    }
```

```
    exported component InMemoryStorage extends nothing {
      ...
    }
  }

  instances nullerInstances {...}
  test case testNuller {...}
  instances interpolatorInstances {...}
  exported test case testInterpolator { ... }
}
```

We can now implement a second processor. For subsequent calls of **process**, it computes the average of the two last speeds of the passed trackpoints. Let us start with the test case. Note how **p2** has its speed changed to the average of the **p1** and **p2** originally.

```
exported test case testInterpolator {
  initialize interpolatorInstances;
  Trackpoint p1 = {
    id = 1
    timestamp = 1 s
    speed = 10 mps
  };
  Trackpoint p2 = {
    id = 1
    timestamp = 1 s
    speed = 20 mps
  };

  ip.process(&p1);
  assert(0) p1.speed == 10 mps;

  ip.process(&p2);
  assert(1) p2.speed == 15 mps;
}
```

Let us look at the implementation of the **Interpolator**. Here it is.

```
exported component Interpolator extends nothing {
  provides TrackpointProcessor processor
  requires TrackpointStore1 store
  init int8 divident;
  Trackpoint* processor_process(Trackpoint* p) <- op processor.process {
    if (store.isEmpty()) {
      store.store(p);
      return p;
    } else {
      Trackpoint* old = store.take();
      p->speed = (p->speed + old->speed) / divident;
      store.store(p);
      return p;
    }
  }
}
```

A few things are worth mentioning. First, the component **requires** another one. More specifically it only expresses a requirement towards an interface, **TrackpointStore1** in

our case. Any component that implements this interface can be used to fulfil this requirement (we'll discuss how below). Second, we use an **init** field. This is a regular field from the perspective of the component (i.e. it can be accessed from within the implementation), but it is special in that a value for it has to be supplied when the component is instantiated. Third, this example shows how to call operations on required interfaces (**store.store(p);**).

The only remaining step before running the test is to define the instances. Here is the code:

```
instances interpolatorInstances extends nothing {
  instance InMemoryStorage store
  instance Interpolator ip(divident = 2)
  connect ip.store to store.store
  adapt ip -> ip.processor
}
```

Two interesting things. First, notice how we pass in a value for the init field **divident** as we define an instance of **Interpolator**. Second, we use **connect** to connect the required port **store** of the **ip** instance to the **store** provided port of the **store** instance. If you don't do this you will get an error on the **ip** instance since it *requires* this thing to be connected (there are also **optional** required ports which may remain unconnected).

You can run the test case now. On my machine here it works successfully :-)

1.3.4 Mocks

Let us assume we wanted to test if the **Interpolator** works correctly with the **TrackpointStore** interface. Of course, since we have already described the interface contract semantically we would find out quickly if the **Interpolator** would behave badly. However, we can make such a test more explicit. Let us revisit the test from above:

```
exported test case testInterpolator {
  initialize interpolatorInstances;
  Trackpoint p1 = {...};
  Trackpoint p2 = {...};

  ip.process(&p1);
  assert(0) p1.speed == 10 mps;

  ip.process(&p2);
  assert(1) p2.speed == 15 mps;
}
```

In this test, we expect the following: when we call **process** first, the store is still empty, so the interpolator stores a new trackpoint. When we call **process** again, we expect the

interpolator to call **take** and then **store**. In both cases we expect **isEmpty** to be called first. We can test for this behavior explicitly via a mock. A mock is a component that specifies the behavior it expects to see on a provided port during a specific test case.

The crucial point about mocks is that a mock implements each operation *invocation* separately (the **step**s below), whereas a regular component or even a stub just describe each operation with *one* implementation. This makes a mock implementation much simpler — it doesn't have to replicate the algorithmic implementation of the real component. Let us look at the implementation:

```
mock component StorageMock report messages: true {
  provides TrackpointStore1 store
  Trackpoint* lastTP;
  total no. of calls is 5
  sequence {
    step 0: store.isEmpty return true;
    step 1: store.store {
        assert 0: parameter tp: tp != null
      }
      do { lastTP = tp; }
    step 2: store.isEmpty return false;
    step 3: store.take return lastTP;
    step 4: store.store
  }
}
```

This mock component expresses that we expect 5 calls in total. Then we describe the sequence of calls we expect. The first one must be a call to **isEmpty** and we return **true**. Then we expect a **store**, and for the sake of the example, we check that **tp** is not **null**. We also store the **tp** parameter in a field **lastTP** so we can return it later (you can add the parameter assertions and the **do** body with intentions). We then expect another **isEmpty** query, which we now answer with **false**. At this point we expect a call to **take**, and another call to **store**. Notice how we return **null** from **take**: this violates the postcondition! However, pre- and postconditions are *not* checked in mock components because their checking may interfere with the expectations! Also, we have slightly changed the test case so we don't stumble over the **null**. We don't **assert** anything about the result of the **process** calls:

```
ipMock.process(&p1);
ipMock.process(&p2);
```

Two more steps are required for this thing to work. The first one is the instances and the wiring. Notice how we now connect the interpolator with the mock:

```
instances interpolatorInstancesWithMock extends nothing {
  instance StorageMock storeMock
  instance Interpolator ip(divident = 2)
  connect ip.store to storeMock.store
  adapt ipMock -> ip.processor
}
```

The second thing is the test case itself. Obviously, we want the test case to fail if the mock saw something other than what it expects on its port. We can achieve this by using the **validate mock** statement in the test. Here is the complete test case (notice the **validate mock** in the last line):

```
exported test case testInterpolatorWithMock {
  initialize interpolatorInstancesWithMock;
  Trackpoint p1 = {
    id = 1
    timestamp = 1 s
    speed = 10 mps
  };
  Trackpoint p2 = {
    id = 1
    timestamp = 1 s
    speed = 20 mps
  };
  ipMock.process(&p1);
  ipMock.process(&p2);
  validatemock (0) interpolatorInstancesWithMock:storeMock;
}
```

1.4 Decision Tables

Let us implement another interface, one that lets us judge flights (we do this in a new section in the **Components** module). The idea is that clients add trackpoints, and the **FlightJudger** computes some kind of score from it (consider some kind of biking/flying competition as a context):

```
exported cs interface FlightJudger {
  void reset()
  void addTrackpoint(Trackpoint* tp)
  int16 getResult()
}
```

Here is the basic implementation of a component that provides this interface.

```
exported component Judge extends nothing {
  provides FlightJudger judger
  int16 points = 0;
  void judger_reset() <- op judger.reset {
    points = 0;
  }
  void judger_addTrackpoint(Trackpoint* tp) <- op judger.addTrackpoint {
    points += 0; // to be changed
  }
  int16 judger_getResult() <- op judger.getResult {
    return points;
  }
}
```

Of course the implementation of **addTrackpoint** that just adds **0** to the **points** doesn't make much sense yet. The amount of points added should depend on how fast and how high the plane (or whatever) was going. The following screenshot shows an embedded decision table that computes points (Notice we mix the components language, the decision tables and the units in one integrated program):

```
void judger_addTrackpoint(Trackpoint* tp) ← op judger.addTrackpoint {
  points += int16, 0
```

	tp->alt < 2000 m	tp->alt >= 2000 m
tp->speed < 150 mps	0	10
tp->speed >= 150 mps	5	20

```
} runnable judger_addTrackpoint
```

Let us write a test. Of course we first need an instance of **Judge**:

```
instances instancesJudging extends nothing {
  instance Judge j
  adapt j -> j.judger
}
```

Below is the test case. It contains two things you maybe haven't seen before. There is a second form of the **for** statement that iterates over a range of values. The range can be exclusive the ends or inclusive (to be changed via an intention). In the example we iterate from 0 to 4, since 5 is excluded. The **introduceunit** construct can be used to "sneak in" a unit into a regular value. This is useful for interacting with non-unit-aware (library) code. Note how the expression for **speed** is a way of expressing the same thing without the **introduceunit** in this case. Any expression can be surrounded by **introduceunit** via an intention.

```
exported test case testJudging {
  initialize instancesJudging;
  j.reset();
  Trackpoint[5] points;
  for (i in [0..5[) {
    points[i].id = i;
    points[i].alt = introduceunit[1800 + 100 * i -> m];
    points[i].speed = 130 mps + 10 mps * i;
    j.addTrackpoint(&points[i]);
  }
  assert(0) j.getResult() == 0 + 0 + 20 + 20 + 20;
}
```

1.5 Accessing Libraries

So far we have not accessed any functions external to the mbeddr program — everything was self-contained. Let us now look at how to access external code. We start with the

simplest possible example. We would like to be able to write the following code:

```
module LibraryAccess imports nothing {
  exported test case testPrintf {
  printf("Hello, World\n");
    int8 i = 10;
    printf("i = %i\n", i);  }
  }
```

To make this feasible, we have to integrate C's standard **printf** function. We could import all of **stdio** automatically (we'll do that below). Alternatively, if you only need a few API functions from some library, it is simpler to just write the necessary proxies manually. Let's use the second approach first.

1.5.1 Manual Library Import

External functions are represented in a so-called **external module**. After you create one and give it a name, it looks like this:

```
// external module contents are exported by default
external module stdio_stub imports nothing resources nothing {
}
```

An external module is always associated with one or more "real" header files. The trick is that when an implementation module imports an external module in mbeddr, upon generation of the C code, the referenced real header is included into the C file. So the first thing we need to do is to express that this **stdio_stub** external module represents the **stdio.h** file:

```
// external module contents are exported by default
external module stdio_stub imports nothing resources header <stdio.h> {
}
```

In general, we also have to specify the **.o** or **.a** files that have to be linked to the binary during the **make** process (in the **resources** section of the external module). In case of **stdio.h**, we don't have to specify this — gcc somehow does this automatically.

So what remains to do is to write the actual **printf** prototype. This is a regular function signature. The ellipsis can be added via an intention. The same is true for the **const** modifier. This leads us to this:

```
// external module contents are exported by default
external module stdio_stub imports nothing resources header: <stdio.h> {
  void printf(const char* format, ...);
}
```

To be able write the test case, we have to import the **stdio_stub** into our **LibraryAccess** implementation module. And in the build configuration we have to add the **LibraryAccess** and the **stdio_stub** to the binary. We should also call the **testPrintf** test case from **Main**.

1.5.2 Automatic Header Import

Later we will need **malloc** and **free** so we can work with dynamic memory. We could create a **stdlib_stub** external module with these two functions manually, like we did it for **stdio** above. However, for the sake of example, we use the automatic import here.

First make sure the **com.mbeddr.core.cstub** language is configured for your model. Then create a new **HeaderImportSpecHFile** in your model. Once created, you can specify an import path. This directly should contain all header files that need to be imported. Note that you should *not* just point it to **use/include**, because importing all of these headers can take a long time! Instead copy the headers you need into a separate directly and specify that one in the import spec. This tutorial comes with a directly **headers** that contains only **stdlib.h**. You can now press the **(Re-)Import Headers** button.

Two things will happen. First, a dialog will open up that reports problems with the import. You can ignore the errors for now. Second, a new external module named **stdlib** is added to the model. Double-click it to open, and rename it to **stdlib_stub**. Once open, you will notice that there are a lof or errors in the file. This is for two reasons. The first one is that this header imports other headers that are not accessible — they were not in the **headers** directory. Consequently all kinds of symbols referred to from the **stdlib** headers are not defined. The second reason is that parsing and importing header files is generally a delicate operation, and for reasons that are beyond this tutorial, it is very likely that some things cannot be parsed. We discuss details . However, for our purposes the import is successful — **malloc** and **free** are correctly imported.

1.6 State Machines

Next to components and units, state machines are one of the main C extensions available in mbeddr. They can be used directly in C programs, or alternatively, embedded into components. To keep the overall complexity of the example manageable, we will show state machine use directly in C.

1.6.1 Implementing a State Machine

To use state machines, the respective model has to use the **com.mbeddr.statemachines** devkit. We then create a new module **StateMachines** and add it to the build configuration. We can also create a test case and call it from **Main** (there is an intention on the **test** expression that automatically adds all test cases that are visible).

Our example state machine once again deals with judging a flight. Here are the rules:

- Once a flight lifts off, you get 100 points
- For each trackpoint where you go more than 100 mps, you get 10 points
- For each trackpoint where you go more than 200 mps, you get 20 points
- You should land as short as possible; for each trackpoint where you are on the ground, rolling, you get 1 point deducted.
- Once you land successfully, you get another 100 points.

This may not be the best example for a state machine, but it does show all of the state machine's features while staying with our example so far :-) So let's get started with a state machine.

```
statemachine analyzeFlight initial = <no initial> {
}
```

We know that the airplane will be in various states: on the ground, flying, landing (and still rolling), landed, and crashed. So let's add the states. Once you add the first of these states, the **initial** state will be set; of course it can be changed later:

```
statemachine analyzeFlight initial = beforeFlight {
  state beforeFlight {
  }
  state airborne {
  }
  state landing {
  }
  state landed {
  }
  state crashed {
  }
}
```

Our state machine is also a little bit untypical in that it only takes one event **next**, which represents the next trackpoint submitted for evaluation. Note how an event can have

arguments of arbitrary C types, **Trackpoint** in the example. There is one more event that resets the analyzer:

```
statemachine analyzeFlight initial = beforeFlight {
  in next(Trackpoint* tp)
  in reset()
  state beforeFlight ...
}
```

We also need a variable **points** that keeps track of the points as they accumulate over the flight analysis:

```
statemachine analyzeFlight initial = beforeFlight {
  in next(Trackpoint* tp)
  in reset()
  var int16 points = 0
  state beforeFlight ...
}
```

We can now implement the rules outlined above using transitions and actions. Let us start with some simple ones. Whenever we enter **beforeFlight** we reset the points to 0. We can achieve this with an entry action in **beforeFlight**:

```
state beforeFlight {
  entry { points = 0; }
}
```

We also understand that all states other than **beforeFlight**, the **reset** event must transition back to the **beforeFlight**. Note that as a consequence of the entry action in the **beforeFlight** state, the points get resetted in all three cases:

```
state airborne {
  on reset [ ] -> beforeFlight
}
state landing {
  on reset [ ] -> beforeFlight
}
state landed {
  on reset [ ] -> beforeFlight
}
```

We can now implement the rules. As soon as we submit a trackpoint where the altitude is greater than zero we can transition to the airborne state. This means we have successfully taken off, and we should get 100 points in bonus. **TAKEOFF** is a global constant representing 100 (**#define TAKEOFF = 100;**):

```
state beforeFlight {
  entry { points = 0; }
  on next [tp->alt > 0 m] -> airborne
  exit { points += TAKEOFF; }
}
```

Let us look at what can happen while we are in the air. First of all, if when we are airborne we receive a trackpoint with zero altitude and zero speed (without going through an orderly landing process), we have crashed. If we are at altitute zero with a speed greater than zero, we are in the process of landing. The other two cases deal with flying at over 200 and over 100 mps. In this case we stay in the **airborne** state (by transitioning to itself) but we increase the points:

```
state airborne {
  on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
  on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
  on next [tp->speed > 200 mps] -> airborne { points += VERY_HIGH_SPEED; }
  on next [tp->speed > 100 mps] -> airborne { points += HIGH_SPEED; }
  on reset [ ] -> beforeFlight
}
```

Note that the transitions are checked in the order of their appearance in the state machine; if several of them are ready to fire, the first one is picked. So be careful to put the "tighter cases" first. The landing process is essentially similar:

```
state landing {
  on next [tp->speed == 0 mps] -> landed
  on next [ ] -> landing { points--; }
  on reset [ ] -> beforeFlight
}
state landed {
  entry { points += LANDING; }
  on reset [ ] -> beforeFlight
}
```

1.6.2 Interacting with Other Code — Outbound

So how do we deal with the **crashed** state? Assume this flight analyzer is running on some kind of server, analyzing flights that are submitted via the web (a bit like **http://onlinecontest.org**). If we detect a crash, we want to notify a bunch of people of the event, so they can call the the BFU or whatever. In any case, assume we want to call external code. You can do this in two ways. The first one is probably obvious. We simply create a function which we call from an entry or exit action:

```
statemachine FlightAnalyzer initial = beforeFlight {
  ...
  state crashed {
    entry { raiseAlarm(); }
  }
}
...
void raiseAlarm() {
  // send emails or whatever
}
```

Another alternative, which is more suitable for formal analysis (as we will see later) involves out events. From the entry action we **send** an out event, which we define earlier.

```
statemachine FlightAnalyzer initial = beforeFlight {
  out crashNotification()
  ...
  state crashed {
    entry { send crashNotification(); }
  }
}
```

Sending an event this way has no effect (yet), but we express from within the state machine that something that corresponds semantically to a crash notification will happen at this point (as we will see, this allows us to write model checkers that verify that a crash notification will go out). What remains to be done is to bind this event to application code. We can do this by adding a binding to the out event declaration:

```
out crashNotification() => raiseAlarm
```

The effect is the best of both worlds: in the generated code we do call the **raiseAlarm** function, but on the state machine level we have abstracted the implementation from the intent.

1.6.3 Interacting with Other Code — Inbound

Let us write some test code that interacts with a state machine. To do a meaningful test, we will have to create a whole lot of trackpoints. So to do this we create helper functions. These in turn need **malloc** and **free**, so we first create an additional external module that represents **stdlib.h**:

```
// external module contents are exported by default
external module stdlib_stub imports nothing resources header: <stdlib.h>{
  void* malloc(size_t size);
  void free(void* ptr);
}
```

We can now create a helper function that creates a new trackpoint based on an altitude and speed passed in as arguments:

```
Trackpoint* makeTP(uint16 alt, int16 speed) {
  static int8 trackpointCounter = 0;
  trackpointCounter++;
  Trackpoint* tp = ((Trackpoint*) malloc(sizeof Trackpoint));
  tp->id = trackpointCounter;
  tp->timestamp = introduceunit[trackpointCounter -> s];
  tp->alt = introduceunit[alt -> m];
```

```
  tp->speed = introduceunit[speed -> mps];
  return tp;
}
```

We can now start writing (and running!) the test. We first create an instance of the state machine (state machines act as types and can be instantiated). We then initialize the state machine by using the **sminit** statement:

```
exported test case testSM1 {
  FlightAnalyzer f;
  sminit(f);
}
```

Initially we should be in the **beforeFlight** state. We can chech this with an assertion:

```
assert(0) smIsInState(f, beforeFlight);
```

We also want to make sure that the **points** are zero initially. To be able to write a constraint that checks this we first have to make the **points** variable **readable** from the outside. An intention on the variable achieves this. We can then write:

```
assert(1) f.points == 0;
```

Let us now create the first trackpoint and pass it in. This one has speed, but no altitude, so we are in the take-off run. We assume that we remain in the **beforeFlight** state and that we still have 0 points:

```
smtrigger(f, next(makeTP(0, 20)));
assert(2) smIsInState(f, beforeFlight) && f.points == 0;
```

Now we lift off by setting the alt to 100 meters:

```
smtrigger(f, next(makeTP(100, 100)));
assert(3) smIsInState(f, airborne) && f.points == 100;
```

So as you can see it is easy to interact from regular C code with a state machine. For testing, we have special support that checks if the state machine transitions to the desired state if a specific event is triggered. Here is some example code (note that you can use the **test statemachine** construct only within test cases):

```
test statemachine f {
  next(makeTP(200, 100)) -> airborne
  next(makeTP(300, 150)) -> airborne
  next(makeTP(0, 90)) -> landing
  next(makeTP(0, 0)) -> landed
}
```

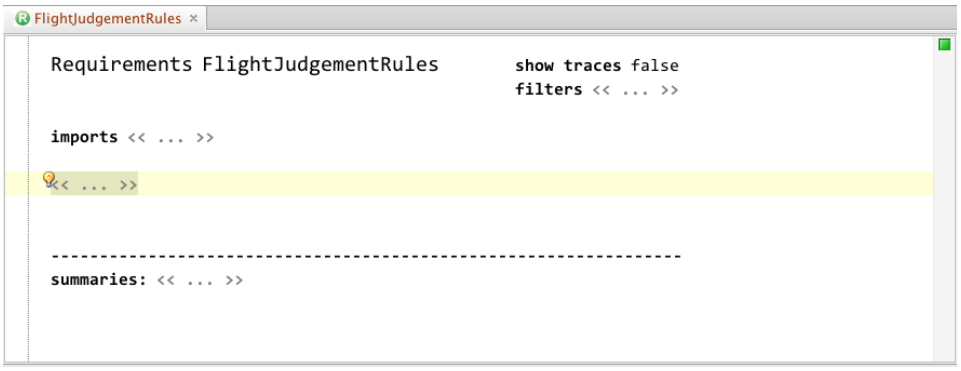
This concludes our discussion of state machines. If you are a C buff, you may have noticed that our application contains a memory leak. We allocate all kinds of heap data (in the `makeTP` function), but we never free it again. Read the chapter to do something about that.

2 Requirements

In the previous chapter we implemented a state machine that judged flights based on a set of rules. These rules were:

- Once a flight lifts off, you get 100 points
- For each trackpoint where you go more than 100 mps, you get 10 points
- For each trackpoint where you go more than 200 mps, you get 20 points
- You should land as short as possible; for each trackpoint where you are on the ground, rolling, you get 1 point deducted.
- Once you land successfully, you get another 100 points.

Let is capture these rules as requirements explictly in the tool. To do so, create a new model **`mbeddr.tutorial.main.req`** and add the **`com.mbeddr.cc.reqtrace`** devkit to it. Inside the model, create a new **`RequirementsModule`**. Call it **`FlightJudgementRules`**:

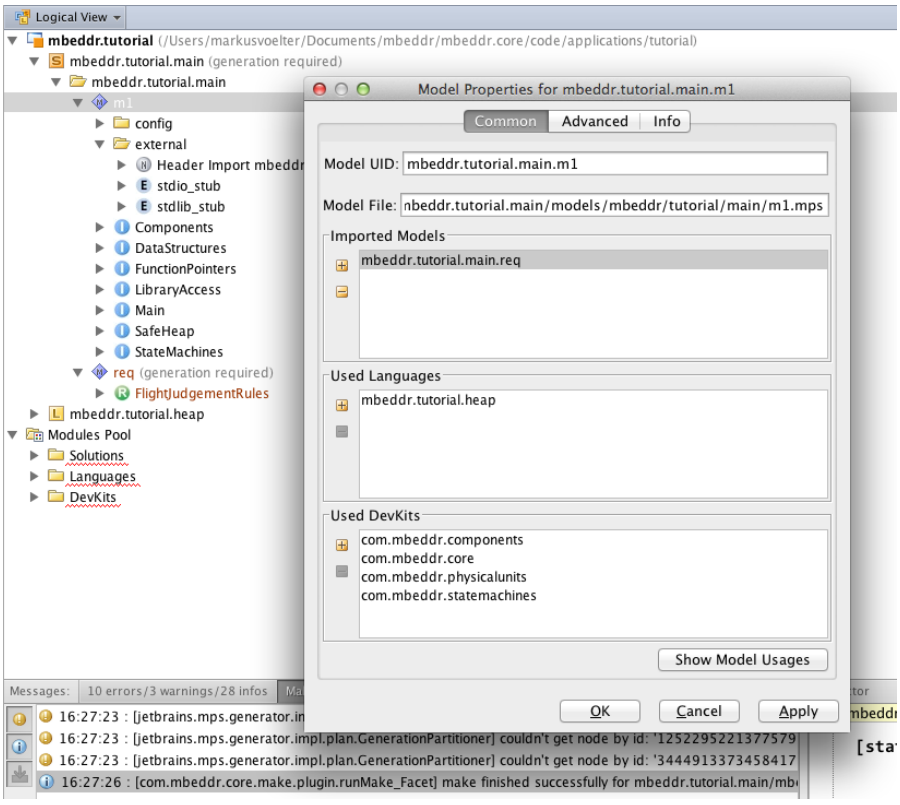


You can now enter the requirements from above. Make up a useful name for each of them and copy the text above into the summary or into the detail description text box.

To build hierarchies you can add child requirements with an intention. Here is how the result looks for me:

```
functional PointsForTakeoff (0): Once a flight lifts off, you get 100 points
functional InFlight (0): Points you get in flight
  functional FasterThan100 (0): For each trackpoint where you go more than 100 mps, y
  functional FasterThan200 (0): For each trackpoint where you go more than 100 mps, y
functional Landing (0): Stuff Relating to Landing
  functional ShortLandingRoll (0): You should land as short as possible
  functional FullStop (0): Once you land successfully, you get another 100 points.
```

Of course, now that we have captured the requirements we want to relate them to the implementation code, which is, in this case, a state machine. So let's go back to the state machine we built in the previous chapter. To be able to trace from the state machine to these requirements we first have to make the **req** model visible to the **m1** model that contains the state machine. Open **m1**'s properties, go to the **Imported Models** section and add the **req** model[1]:



[1]Note that this import is a *physical* import between the models (XML files essentially), whereas the **import** clause on modules is a *logical* namespace import.

While you are in the model properties dialog of **m1**, you can also add the **com.mbeddr.cc.reqtrace** devkit to that model; we need it for tracing. We want to establish traces between implementation code and the requirements we just captured. To make this possible we first have to specify that the **StateMachines** implementation module contains code that traces to the **FlightJudgementRules** requirements. We can do this by attaching a **Reference to a Requirements Module** to the implementation module by using an intention:

```
requirements modules: FlightJudgementRules
module StateMachines imports DataStructures, stdlib_stub, stdio_stub {
```

We can now add traces in the state machine. Remember the contents of the **beforeFlight** state:

```
state beforeFlight {
  entry { points = 0; }
  on next [tp->alt > 0 m] -> airborne
  exit { points += TAKEOFF; }
}
```

When the state is exited, we add 100 points (represented by the **TAKEOFF** constant). This is an implementation of the first requirement/rule. Select the exit action and execute the **Add Trace** action. Doing this on the other relevant program elements leads to a program with traces. The color of the trace actually depends on the kind of trace; the **implements** kind defaults to green:

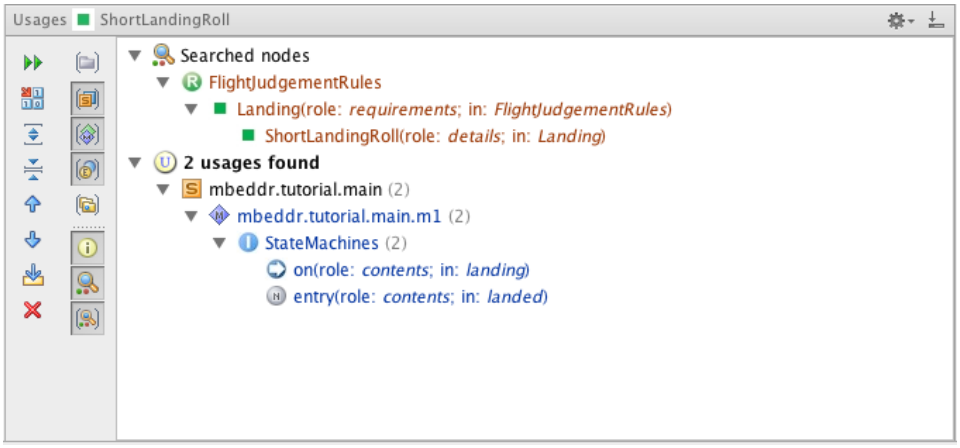
```
[#define TAKEOFF = 100;]-> implements PointsForTakeoff
[#define HIGH_SPEED = 10;]-> implements FasterThan100
[#define VERY_HIGH_SPEED = 20;]-> implements FasterThan200
[#define LANDING = 100;]-> implements Landing

statemachine FlightAnalyzer initial = beforeFlight {
  in next(Trackpoint* tp) <no binding>
  in reset() <no binding>
  out crashNotification() ⇒ raiseAlarm
  readable var int16 points = 0
  state beforeFlight {
    entry { points = 0; } entry
    on next [tp->alt > 0 m] ➔ airborne
    [exit { points += TAKEOFF; } exit]-> implements PointsForTakeoff
  } state beforeFlight
  state airborne {
    on next [tp->alt == 0 m && tp->speed == 0 mps] ➔ crashed
    on next [tp->alt == 0 m && tp->speed > 0 mps] ➔ landing
    [on next [tp->speed > 200 mps] ➔ airborne { points += 20; } on next]-> implements FasterThan200
    [on next [tp->speed > 100 mps] ➔ airborne { points += 10; } on next]-> implements FasterThan100
    on reset [ ] ➔ beforeFlight
  } state airborne
  state landing {
    on next [tp->speed == 0 mps] ➔ landed
    [on next [ ] ➔ landing { points--; } on next]-> implements ShortLandingRoll
    on reset [ ] ➔ beforeFlight
  } state landing
  state landed {
    [entry { points += LANDING; } entry]-> implements ShortLandingRoll
    on reset [ ] ➔ beforeFlight
  } state landed
  state crashed {
    entry { send crashNotification(); } entry
  } state crashed
}
```

Note how we attach traces to constants, transitions and actions: the trace facility is orthogonal with regards to the language constructs that are annotated. It works for all.

Returning back to the **FlightJudgementRules** requirements, we can select the heading and run the **Update Trace Status** intention. Requirements that are traced from program code are marked blue:

```
⊞ ● functional PointsForTakeoff (0): Once a flight lifts off, you get 100 points implemented
⊞ ● functional InFlight (0): Points you get in flight
  ⊞ ● functional FasterThan100 (0): For each trackpoint where you go more than 100 mps, you get 10 points implemented
  ⊞ ● functional FasterThan200 (0): For each trackpoint where you go more than 100 mps, you get 20 points implemented
⊞ ● functional Landing (0): Stuff Relating to Landing implemented
  ⊞ ● functional ShortLandingRoll (0): You should land as short as possible implemented
  ⊞ ● functional FullStop (0): Once you land successfully, you get another 100 points.
```

We notice that the **FullStop** requirement is still red, so it seems we haven't trace it correctly. Maybe we mixed something up with the landing stuff. Let's find out from where we trace to the **ShortLandingRoll** requirement. To do this, select the **Find Usages** entry from the context menu on the **ShortLandingRoll** requirement. In the dialog that opens uncheck all Finders except **Traces** and press **OK**. Here is the result you get:



The dialog reveals that we have attached **`ShortLandingRoll`** to two elements. A transitin and an entry action. That's wrong: the second one should have been the **`FullStop`** requirement. Change it, rerun **`Update Trace Status`** on the requirements module, and everything should be blue.

Finally, if you are annoyed by your beautiful state machine being polluted by all these requirements traces, you can hide them: select **`false`** for **`show traces`** in the top right corner of the **`FlightJudgementRules`** requirements module. The traces are still there, of course (and remain attached as you copy, paste or move program elements), but they are not shown.

Bibliography