Blech Tutorial

Gretz, Friedrich Grosch, Franz-Josef

Bosch Corporate Research

Version: 1.1
May 2020

This document was originally released as supplementary material for the Bosch Conference on Software Engineering tutorial on *Blech*. It may be used on its own as a means to get a first impression of the language and get a *Blech* project up and running. To complete the programming assignments a Bosch XDK device is required.

Contents

1. Embedded Software

Software is deployed in many different contexts. Here we try to give an intuitive description of embedded systems and their software. Embedded systems are hardware/-software systems wherein discrete software interacts with the analogue world. Sensors measure continuous quantities and provide discrete values to the software. The software reacts by producing control points for actuators or information to be forwarded to other software systems. The term *embedded* reflects that the system does not have a purpose in itself. Instead it provides some functionality within a more complex system, for example the engine control is embedded in a car. The software running in such systems is often called “embedded software”. It consists of various different parts, cf. Fig. 1.1. There may be some real-time operating system that schedules different software components. Drivers are used to make use of particular hardware features while the hardware abstraction layer glues driver software together with the hardware of various platforms. We are mostly interested in what is called the “application level” software. This is where the logic and the functionality of the embedded system is programmed.

The description of our domain of interest is not precise, of course. There are systems which are not classically perceived as embedded systems such as a software-implemented network traffic router. It operates purely on logical data and has no sensors or actuators. Yet it shares the state-based, event-triggered behaviour with the embedded systems described here and the considerations of the following chapters may apply to it too. On the other hand, there are applications that need to work at faster rates and with lower power consumption than what is possible with a software programmable micro controller. Such applications will usually be implemented in a hardware description language. The result of this implementation is a custom application-specific integrated circuit (ASIC) or a configuration of a field-programmable gate array (FPGA). Albeit being part of an embedded system these applications are out of scope for us.

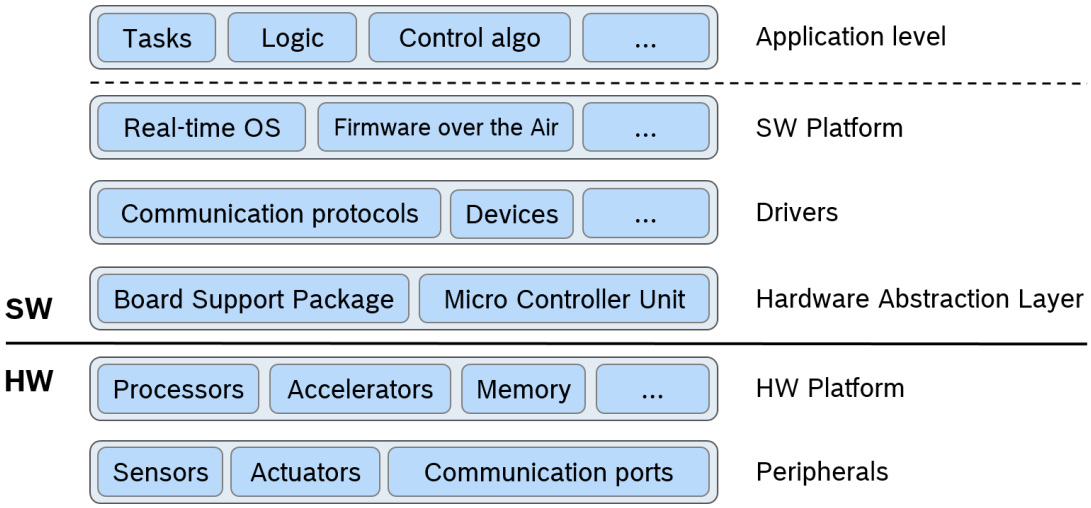

Figure 1.1.: Typical layers of an embedded hardware/software architecture.

2. Challenges

2.1. Resource constraints

Costs per unit and energy efficiency are the two most important requirements for any embedded system. A harsh environment may be another reason that precludes the usage of very fast but sensitive processors. As a consequence the hardware has to be chosen such that it provides just the necessary computation power and memory storage. The software must then be as resource efficient as possible to fulfil its task with the limited resources given. This precludes some of the abstractions and technologies known from desktop or web application development. Examples are functional and logical programming, runtime polymorphism, type introspection or even reflection.

Often an embedded system fulfils a safety critical function. For such applications showing partial correctness[1] is not enough. It needs to be shown that the hardware-software system never crashes and produces its results *on time*. Fortunately, simpler hardware architectures often make the runtime behaviour of any given software more predictable. But also the behaviour of the software needs to be further restricted. For example, memory must not be dynamically allocated at runtime. The reason is that memory leaks are hard to find and may lead to a fatal crash once they occur. Also the timing is affected by the amount of memory used. A typical embedded system has a fixed amount of inputs, outputs and internal state variables. The needed amount of memory does not change over the lifetime of an embedded program. It does not need to generate "garbage" on the heap that can be cleaned up once the result has been computed. Unlike desktop applications, embedded software applications are not loaded and unloaded at runtime—they run continuously.

This leads us to the next section.

2.2. Divide and conquer in embedded systems

Classically, computation problems are decomposed into simpler subproblems that are solved individually. Then, the intermediate results can be processed to obtain the final

[1] A program is partially correct if it produces the correct result *provided that it terminates*. However there is no guarantee that it does terminate on all inputs nor is there a time bound on how long a terminating execution may take. This is the notion of *correctness* which is most commonly used in verification.

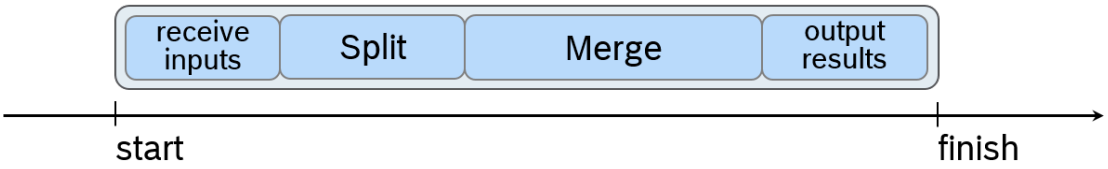

Figure 2.1.: Typical sequential composition of an algorithm.

result. The reasoning is sequential: first do all steps necessary to solve the first subproblem, then all steps to the second, etc... Consider Fig. 2.1, for example. It shows a sequence of operations carried out by the "mergesort" algorithm: it receives an array to be sorted, splits it into smaller chunks which then can be merged while respecting the order between elements. Finally the sorted array is returned. When executed, the algorithm has a point in time where it starts and where it terminates but there is no necessary requirement on the computation duration. Classical computation problems allow performance gains through parallelisation whenever the subproblems to be solved are independent.

This is substantially different for embedded systems. Their software usually runs in *reactions*. It is triggered whenever some specific event occurs, for instance when a timer expires. Then the software is given some inputs and is expected to react with corresponding outputs. However, this single reaction step will involve computations from different software components that are responsible for different functionalities. Conceptually, these software components concurrently perform one reaction step each. For example, imagine a driver assistance system which consists of a collision detector, trajectory planner and controller for steering and acceleration. A classical sequential mindset would dictate: first make all collision detection steps, then plan the whole trajectory and finally emit all control steps and terminate the program. Of course, this does not make any sense due to the reactive nature of the system. Instead, in every time tick the system would perform a reaction step in each of the components. Figure 2.2 visualises the execution of such a reactive system. Ticks along the timeline indicate that a reaction is triggered at specific points in time. In every time tick every software component receives inputs, carries out some computation, returns outputs and waits for the next tick. This is indicated by boxes and the horizontal, dashed arrows between them. During a reaction instance there may be some communication between the individual software components which is indicated by the vertical arrows. The crucial insight is that from the modelling perspective we have several components and they execute concurrently from one reaction to the next. It is true that the individual instructions that need to be performed during one reaction instance possibly have to be scheduled into a sequence to be executed but that is a detail below the programmer's abstract model of computation.

Thus decomposition of the software's complex overall behaviour across reaction steps is obtained by concurrent and hierarchical composition of simpler behaviours. An ap-

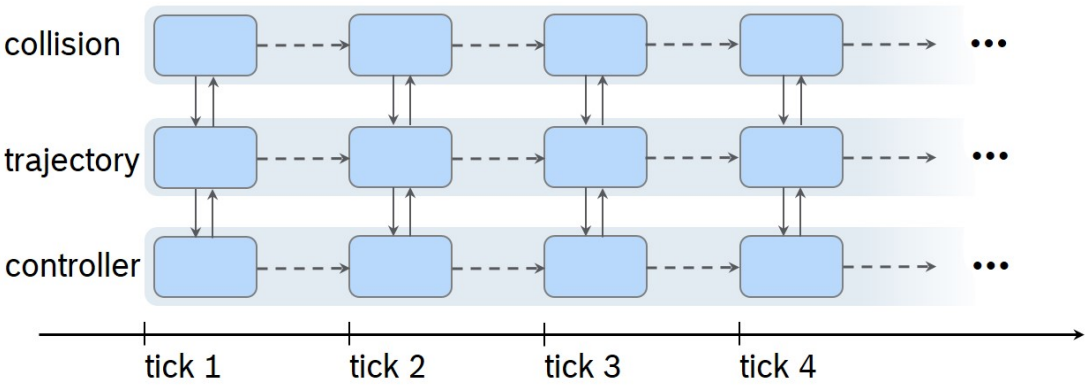

Figure 2.2.: Reactive and concurrent nature of embedded software.

propriate programming language for embedded software should have means to express reaction steps and the concurrent composition of subprograms running in steps.

2.3. Mode switching behaviour

Usually, the application software does not always repeat the same computation in every reaction. Instead it reacts differently depending on the inputs given (user input, sensor readings) and its current state. A program's state is the evaluation of all program variables and program counters. Typically a program will run through many states while executing one reaction step. As programmers we prefer to abstract from concrete states that the program assumes and rather speak of "modes". A mode is a more abstract term to describe a particular behaviour that is currently executed. For example, the software may be in a mode where a PID controller with particular parameters is used to control the system. In a different mode, a different set of parameters or even a different control algorithm may be used. A reaction of a program may result in a mode transition.

We believe it is crucial that modes and transitions between them can be expressed directly in the programming language that is used to develop the application logic. The key issue here is to not encode mode switching behaviour in goto-like jumping spaghetti code but instead allow for a structured imperative control flow.

2.4. Software is *soft*

Often people working in embedded software are so caught up in the aforementioned restrictions of the domain that they seem to forget what writing software is all about: it is flexibility. It is fundamentally wrong to believe that once a product is released its

software does not change. The whole point is that the software can evolve with changing customer needs, be adapted to different variants of customer needs, be ported to other platforms and even be upgraded on devices already in the field. The latter might be due to new features which were developed after the product's release or—in the worst case—due to a bug discovered after release.

All this goes to show that the chosen implementation technology should offer as much flexibility to the developer as possible while meeting the constraints discussed in the previous sections. The program text should be easy to read[2], understand and modify. Local changes must not unexpectedly alter the global behaviour of the software. Clear-cut interfaces are needed so separate individual modules or functions of the software.

[2] "Program text is read more often than written" is a well-known mantra.

3. Synchronous Programming

Here we describe the essential building blocks of an embedded application program by means of our language *Blech*.

Since we are interested in the application level software only, we may assume there is already some basic software—the runtime environment—that takes care of reading sensors and acting upon actuators. We look at the logic in between.

Activities Our software will be built of subprograms called *activities*. For example, here is a declaration of an activity with the name `MyAct`.

```
activity MyAct (in1: bool, in2: float32) (out1: nat8)
    // some code ...
end
```

There are two inputs `in1` and `in2`. The activity may only read its inputs but not modify them. A second parameter list contains one output variable `out1`. It may be both read and written. In every tick the values of the inputs are updated, the code inside `MyAct` is executed and the outputs are written back to the caller.

Reactions We have explained that an application executes in reaction steps. In order to explicitly define a control point where a reaction ends and the next one begins we use the `await` statement. For example, we could write the following code.

```
1  activity MyAct (in1: bool, in2: float32) (out1: nat8)
2      repeat
3          await in1
4          out1 = (out1 + 1) % 100 // count from 0 to 99
5      until in2 < 0.0f end
6  end
```

Initially, `MyAct` is entered and the control flow proceeds to line 3 and stops (regardless of the inputs). In the next tick, `MyAct` resumes its execution at the `await` statement in line 3. It checks the boolean input variable `in1`. If it is false, the reaction ends immediately, the control flow does not advance. Otherwise, if `in1` is true, the calculation is carried out in line 4. This updates `out1` to a new value. Finally, if input `in2` is indeed less than 0 the activity terminates its execution. Otherwise the control flow loops around from line 5 back to line 2 and finally the reaction ends again in line 3. Note that the inputs are not

volatile. *Blech* semantics guarantees that the value of an input does not change while a reaction is running[1].

Activities may terminate after a finite number of reactions but they do not have to. The only requirement is that each reaction takes only a finite amount of time. Therefore there must be an `await` statement on every control flow path in an activity—in particular, every loop must have a pause in its body.

Concurrent composition Several pieces of code that describe stepwise behaviour may be composed concurrently using `cobegin..with..end`.

```
1  activity P ()
2      var x: int32
3      var y: int32
4      var z: int32
5      cobegin
6          run A(x)(z)
7      with
8          run B(y)(x)
9      end
10 end
```

Assume the activities `A` and `B` have already been implemented. In lines 5 – 9 they are composed concurrently. This means the control flow of `P` is forked into two control flow points. One resides in `A` (line 6) and one in `B` (line 8). With every tick both, `A` and `B`, will perform one reaction. When both subprograms terminate, `P` regains control in line 9 and, in this example, terminates too. Of course, more than two branches can be combined using further `with` blocks.

Write-before-read order Notice that in the previous example activity `A` read x and produced z while `B` read y and produced x *concurrently*. To achieve a deterministic behaviour we have to define what value of x is given to `A`. We define that any shared variable must be written before concurrent readers may access the variable. The code generation will automatically take care of this and generate a code which will first execute the step in `B` and then in `A`. We call this the causal execution order. This example shows that the lexicographical order in the source code text is irrelevant to the execution order—only the data flow matters. This makes the programs robust to code refactorings where branches are moved, or code changes where new branches may be introduced.

It is possible to write programs where no causal ordering is possible:

[1] This is a consequence of the synchrony hypothesis, cf. [3]

```
cobegin
    run A(x)(y) // now A writes to y
with
    run B(y)(x)
end
```

In such cases the compiler will raise an error and refuse to translate this program. It is up to the programmer to decide in this case where execution should start using a *previous* value of a variable. For example:

```
cobegin
    run A(x)(y)
with
    run B(prev y)(x) // use value of y from the previous reaction
end
```

Here the execution is clearly determined. Activity `B` starts with the value of y from the previous reaction and computes a new value for x in the current reaction. This current value of x is then used by `A` to produce the new current value of y.

Preemptions In the above `cobegin` examples both branches must terminate before the calling thread regains control and proceeds. Sometimes however we do not want to wait for all branches to terminate and instead want to move on as soon as one of them terminates. Consider this example:

```
1  cobegin weak
2      run WaitForKeyStroke(...)(...) // ... arguments for readability
3  with weak
4      run WaitFor5Seconds()()
5  end
```

This piece of code will halt the control until either we detect a key stroke from the user and `WaitForKeyStroke` terminates or five seconds elapse (or both happens at the same reaction step). The `weak` keyword says that the following block may be aborted at the end of its reaction step. A `cobegin` statement joins in the reaction step in which some branch terminates and all strong branches have terminated. To conclude, consider a last example:

```
1  cobegin weak
2      repeat // infinite loop
3          await true
4          out1 = (out1 + 1) % 100
5      end
6  with
7      run WaitForKeyStroke(...)(...) // no arguments for readability
8  end
```

Here the loop in the first block is intentionally infinite. However the block is weak and hence is aborted at the end of the reaction in which the key stroke was detected and the `WaitForKeyStroke` terminates.

Other variants of preemptions are the `abort` and `reset` statements. They are almost self explanatory.

```
1  activity MyAct (in1: bool) (out1: nat8)
2      // do something ...
3
4      when in1 abort
5          out1 = 1
6          await true
7          out1 = 2
8          await true
9          out1 = 3
10     end
11
12     // do something else ...
13 end
```

The statement in line 4 says that *when* a reaction *starts* in the block lines 5 – 9, it is checked whether `in1` is true and in that case the control flow skips to line 10. Thus when control flow reaches line 4 it will immediately proceed to line 5, set `out1` accordingly and finish this reaction in line 6 (regardless of the value of `in1`). The next reaction starts by checking the abort condition `in1`. If it is true we skip the rest of the block and proceed to line 10. Otherwise, we check the condition of the `await` statement which here is vacuously true and the reaction proceeds to line 7 and finishes in line 8. The same reasoning applies in line 8: the execution is possibly aborted before setting `out1` to 3. In any case, the block is left in line 10.

The `abort` statement is useful whenever we want to skip over a sequence of reactions when we detect some issue at the beginning of a reaction. Sometimes instead of skipping ahead we would like to restart a sequence of reactions. For this we may use the `reset` statement.

```
1  activity MyAct (in1: bool) (out1: nat8)
2      // do something ...
3
4      when in1 reset // reset instead of abort
5          out1 = 1
6          await true
7          out1 = 2
8          await true
9          out1 = 3
10     end
11
12     // do something else ...
13 end
```

It behaves just like the `abort` statement from the previous example except it jumps to line 4 if `in1` is true.

Data structures The previous examples relied on inputs and outputs of activities to carry out computations. It is also possible to declare different kinds of variables.

```
1  const LEN: int8 = 5
2  param lut: [LEN]float32 = {-0.1f, 0.1f, 0.2f} // remaining are 0.0f
3
4  activity MyAct (in1: bool) (out1: nat8)
5      // ...
6      let x: bool = not in1
7      var y = out1 + 12
8      // ...
9  end
```

The keyword `const` declares a compile-time constant. It allows you to parameterise your code with values which need not have a representation after the compilation. The keyword `param` declares run-time read-only constants. Most often this is used for some lookup data structures such as characteristic maps. Note, that `params`s are supposed to be customizable in the compiled binary. Inside an activity you additionally may declare data with `let` and `var`. The former allows read-only access only. The latter declares an ordinary mutable variable. Note that `let` and `param` are not the same. In the example you see that the value of `x` depends on the value of `in1` at run-time. The value of a `param` must be known at compile time already.

In the examples we have used some primitive data types. So far we support `bool`, `nat8`, `nat16`, `nat32`, `nat64`, `int8`, `int16`, `int32`, `int64`, `bits8`, `bits16`, `bits32`, `bits64`, `float32` and `float64`. Line 2 in the example above also demonstrates an array of five 32-bit floats. The initialiser deliberately leaves out the last two values which are automatically filled in with the default value by the compiler. The default value of a boolean is `false`. For all numerical types the default value is zero.

Sometimes types can be deduced by the compiler by looking at the initialiser. For instance, in line 7 the sum of an `nat8` and a constant number is determined to be of type `nat8` again and hence this is the type of `y`.

Structures can be declared using the `struct` keyword.

```
1  struct MyStructure
2    var x: int32 // mutable field
3    let id: nat32 // fixed at initialisation
4  end
```

Functions We use functions to encapsulate complex expressions or algorithms that run entirely within one reaction. Functions cannot contain any statements which we use for reactive concurrent programming: `await`, `run`, `cobegin`, `abort`, `reset`.

```
function getMean (arr: [LEN]int32) (/*no outputs*/) returns int32
    var i: nat32 = 0
    var sum: int32 = 0
    repeat
        sum = sum + arr[i]
        i = i + 1
    until i >= LEN end
    return sum / LEN
end
```

Just like in activities we also distinguish between read-only inputs and read-write outputs in functions. Output parameters are useful when a function needs to update the given data in-place instead of returning a new value. This makes side-effects explicit for the programmer.

4. Hands on *Blech*

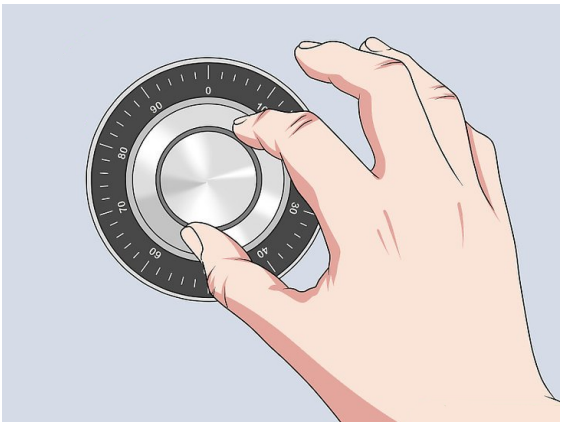

Figure 4.1.: A safe with a dial lock.
Downloaded and adapted from https://www.wikihow.com/Open-a-Safe under CC BY-NC-SA 3.0 license on 09th September 2019.

In this chapter we develop our own *Blech* application. Our toy example uses the analogy of a safe lock. A safe is usually unlocked using a secret code which comprises a sequence of dial rotations, cf. Fig. 4.1. Here we want to mimic the dialling process using a Bosch XDK. The user interacts with the "virtual safe lock" by turning it and pressing buttons. The XDK signals success or failure using its LEDs . Of course, this is not a sensible product but just a toy. However, it demonstrates nearly all aspects of embedded software and its challenges, as well as language features to meet those.

4.1. Preparation

If not already done, install

1. the *Blech* compiler `https://www.blech-lang.org/docs/getting-started/blechc/`
   This compiles *Blech* code to C code
2. Visual Studio Code `https://code.visualstudio.com/`
   We use VSCode to edit *Blech* source files.

3. *Blech* language services for VSCode `https://www.blech-lang.org/docs/getting-started/vsce/`
   This plug-in makes VSCode aware of the *Blech* language and offers editing support such as syntax highlighting or type checking
4. XDK Workbench `https://developer.bosch.com/web/xdk/downloads` *(you need to sign up for free to access the downloads and you need admin rights to install the workbench)*
   This compiles our C code and flashes the result onto the XDK device

We provide code skeletons for this tutorial: `https://www.blech-lang.org/docs/examples/virtuallock`.

4.2. Blinking LEDs

1. Once everything is installed, start the XDK Workbench, select a workspace folder and start a new Mita project ("Use Eclipse Mita"). This will create a folder "EclipseMitaApplication" inside your workspace. It contains a default code skeleton which we do not need however. Instead we copy our own code skeleton into this folder. For this, copy all files from the provided `01_Blinking_LEDs` into `EclipseMitaApplication` (possibly overwriting existing files!).
2. From a command line navigate to `EclipseMitaApplication\src-gen` and run `blechc .\virtualSafeLock.blc`
3. In the XDK Workbench, force the re-compilation of the project simply by making a minimal change to `application.mita`. *For example, add a space in some empty line and save the file.*
4. Connect your XDK to the PC using the USB cable and turn on the XDK (there is a power switch on the XDK itself!) After a short moment, the XDK Workbench should show that an XDK is connected through a COM port. *If it does not, try unplugging the cable from the XDK and connecting it back again.*
5. Hit the `Flash` button in the XDK Workbench. This compiles the generated C code and all necessary drivers, real-time OS, ... into one HEX file and flashes that onto the XDK. It takes a while when running the first time. Subsequent runs will be much faster because only the parts which have changed are recompiled. *If there is an "Invalid Application" error at the very end, hit "Flash" again.*

We have run through the complete build process now, cf. Fig. 4.2. For the rest of this tutorial all you need to do to try out your *Blech* code on the XDK is: edit virtualSafeLock.blc, run `blechc`, click `Flash`.

Now, that the toolchain is tested and we know how to compile our code, let us finally write some in *Blech*.

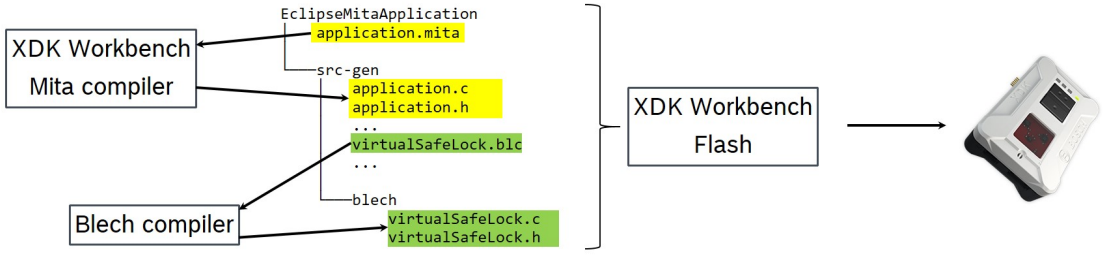

Figure 4.2.: The build process of the entire project: the triggering environment is specified in Mita and is translated to C by the Mita compiler (once); the *Blech* sources are translated to C by the *Blech* compiler; finally, the XDK Workbench compiles these C sources together with drivers and libraries into one executable HEX file and flashes that onto the XDK device.

Assignment Fill in the necessary *Blech* code to make the device blink at a rate of 1 Hz (change its state every 500 ms). You may assume that our Blech program will be called every 100 ms.

In case you are curious how this is achieved you can look it up in the application.mita file. There you see that the Blech tick function is called from an "every 100 milliseconds" block.

4.3. Baseline Specification – Unlocking the virtual lock

1. In the z-plane we discern four positions: top, left, right, bottom
2. LEDs blinking at 1 Hz (every 500 ms) indicate the locked mode
3. Permanently lit LEDs indicate the unlocked mode
4. While entering a sequence of positions the device indicates whether one of the four predetermined positions is recognised (middle LED lit) or whether the device is just about left of or right of a determined position (left, resp. right LED lit)
5. Initially, the device is locked
6. From the locked mode pressing button one starts the unlocking procedure
7. While unlocking, to enter a position the device must be in a determined position and the user must press button one
8. If a wrong position is entered, the device returns to the locked mode
9. Once the last position has been successfully entered the device transitions to the unlocked mode

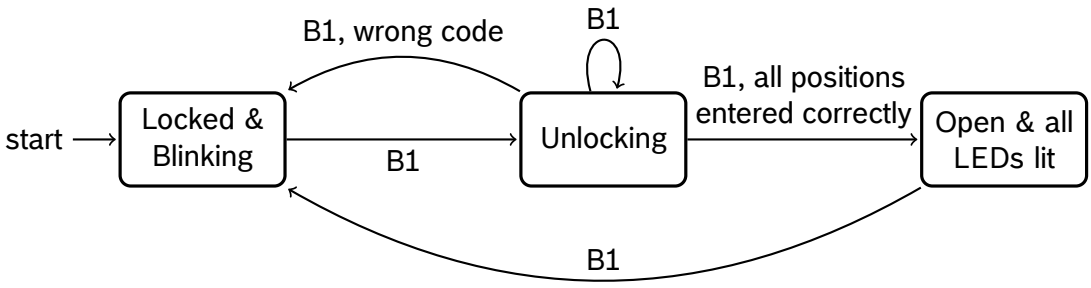

Figure 4.3.: Visualisation of base line specification. B1 – button one is pressed.

Figure 4.3 visualises the behaviour in a diagram.

Assignment

4.3 a) We provide all necessary code to determine the pose of the device given the accelerometer sensor readings. Copy the skeleton from `02_Unlocking`.

4.3 b) Where and how do you store the secret sequence of poses?

4.3 c) How do you model the three modes? How do you transition between them?

4.3 d) Implement the unlocking procedure as described. *Make use of the given activity DisplayOrientation to process sensor readings.*

4.4. Cancellation

We add an additional behaviour to our device. The user may abort any operation by putting the device face down on the table.

Assignment Implement the additional requirement:

- If the XDK lies face down ($z < -900$) on the table (*mlux* < 18000) the program should jump back to its initial state.

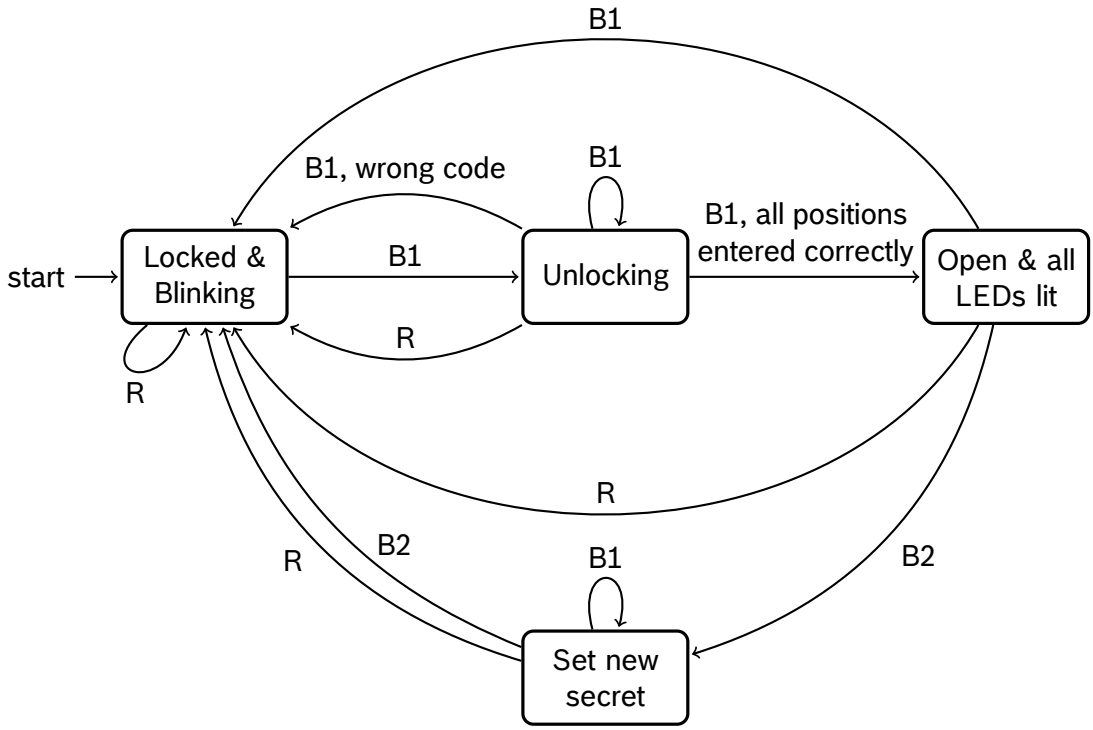

Figure 4.4.: Visualisation of the full specification. B1 – button one is pressed, B2 – button two is pressed, R – reset by placing device face down on the table.

4.5. Full Specification – User Defined Secret Code

Once the device is unlocked the user may set her own secret. The secret sequence may contain up to 8 poses (or less). Entering the poses works analogously to the unlocking phase. Pressing button two after entering some poses saves the sequence and returns to the locked mode. The user may also choose to simply lock the safe again without changing the code. Make sure that the current secret remains unchanged if the user aborts the process of setting a new secret. Figure 4.4 shows a diagram with all possible transitions between modes.

Assignment Finish the implementation of the virtual lock according to the given specification.

4.6. Bonus: a Change Request Comes in...

Now that we have implemented the full specification we are done. So it seems. An expert evaluates our lock and reports that it reveals too much information to an attacker. Since it fails on the first incorrect input and immediately goes to locked mode the secret may be guessed with at most three attempts per position. In order to prevent this information leak the unlocking phase is redesigned as follows:

- In the unlocking phase, keep reading poses entered with button one.
- Only when button two is pressed make a transition to one of the two possible successor modes, depending on whether the input was correct.

Assignment Correct your implementation accordingly.

4.7. Retrospect

Our example demonstrated the **integration** of a *Blech* application on a HW/SW platform. We have used the Bosch XDK as the hardware platform. Mita was used to specify the software runtime. The runtime periodically provides inputs to our *Blech* program and interprets its outputs.

Blech was used to implement the logical steps of our application. **Stepwise behaviour** is implemented inside subprograms that we call *activities*. An activity may pause its control flow with the `await` statement. Executing an activity from one pause to the next is what we call a *reaction*. In Sect. 4.2 we have started out with the simplest stepwise behaviour: counting steps and inverting the LEDs upon every fifth step.

In the next section we started to introduce different **modes of operation**[1]. Our virtual lock could progress from an initial mode to a mode where the user is entering the secret key and upon success a final mode was reached that indicated that the lock is open. Towards the end of the tutorial we added more modes, refined their behaviour and added more transitions between these modes.

Our main tool to build complex behaviours from simple ones is **composition**. The most obvious form of composition is sequential composition. Using synchronous **preemptions** we may skip part of a sequence or jump back to the beginning of a sequence. This was particularly useful in Sect. 4.4. Finally, we have used the `cobegin` statement to express **concurrent composition**. It allows to combine behaviours that happen in the same reaction. For example, in the unlocking mode the device had to indicate its

[1] The SCODE method [1] may help you to identify what are the modes of operation in a given system. Once these are known, *Blech* offers the means to implement the modes and transitions between them in a straightforward way.

pose through its LEDs while also accepting user input and checking its correctness. The `cobegin` statement gives us the flexibility to decide which of its branches must finish execution (strong branches) and which may be aborted at the end of a tick (weak branches).

We use activities to **encapsulate** stepwise behaviour and we use function to encapsulate complex expressions or sequential algorithms. Structuring our program using manageable subprograms facilitates the separation of concerns, allows for modular composition of software and increases maintainability. For large projects this means different teams can develop different parts of a system concurrently and independently as long as the interfaces of these parts are clearly defined.

By design *Blech* **does not have global mutable variables**. Instead the programmer uses parameter lists to pass data in and out of subprograms. In particular, parameters are grouped in an *input* (read-only) and an *output* (read-write) list. This clearly documents what a caller may expect the callee to do with the data. The automated *causality analysis* leverages this information to ensure **consistency of data** between concurrent branches. This makes handling shared variables trivial. Write-write conflicts between concurrent branches are automatically flagged by our compiler. The same is true for read-write cycles (causality cycles). These can be resolved using previous values of input arguments. Since the concurrent composition is **deterministic**, *Blech* programs can be tested with **reproducible** outcomes.

Using this small example we hope to have demonstrated how *Blech* addresses typical challenges in embedded software development, cf. Chapter 2. In the following, we discuss further potential applications of *Blech* and give an outlook of what may be added to the language to leverage its full potential in the future.

5. Scope of *Blech*

Our example in the previous chapter illustrated how the *Blech* language can beneficially used in the embedded systems context. The virtual safe lock, of course, is not a real product. It illustrates how a software product on the Bosch XDK might be developed. However, it is important to stress, that the *Blech* language is in no way tied or limited to a specific platform or product area. Here we give a few pointers into different product areas where *Blech* could substantially improve software development.

5.1. In scope

Today, embedded software is an integral part of all sorts of products. Prominent examples from the automobile industry include electronic control units for all sorts of car components such as the engine, brakes, steering, bodywork electronics or multimedia. In the same way, software is used to control smart home appliances or power tools. Building technology comprises hazard recognition, public address systems, heating or air-conditioning which all are controlled by a software system. Digitalisation in factories, commonly referred to as Industry 4.0, is largely based on adding software systems to machines or produced items in order to monitor or control them. In the same way all the things and gateways in an Internet-of-things architecture rely on software embedded into them.

In all these examples we find commonalities: the software reacts to external inputs in the form of sensor samples or other triggering events. The result of a computation step is some sort of information or command for the environment. Often decisions made during the reaction step are based on the internal state or mode of operation of the software. In such situations *Blech* helps to introduce abstractions and increase maintainability and flexibility of the software.

Note that it does not matter whether the implemented functionality is “close to the hardware” such as an operation on memory buffers triggered by an interrupt, or whether it is a “high level” functionality such as a controller of an infotainment system which decides which menu to display next to the user. It is the *reactive behaviour* of the functionality that is relevant for *Blech*.

The language is designed to meet tight memory and execution time constraints which often are imposed by safety critical real-time systems.

5.2. Out of scope

By design *Blech* precludes asynchronous communication between its activities. *Blech* programs thus may be part of a “globally asynchronous locally synchronous” (GALS) architecture but they cannot replace the asynchronous middleware which is needed to connect the synchronous “islands”.

Another key design choice in *Blech* is to preclude dynamic memory management. This provides us with guarantees which are needed to build safety-critical systems. However it prevents us from building data intensive applications which require data structures *of arbitrary size* natively in *Blech*.

Finally, our concurrency mechanisms allow only a static branch structure as well. This means we cannot spawn worker threads whose number depends on the given input. A typical scenario where this is needed is graphics processing where a program running on a GPU may start as many workers as there are rows in a given matrix.

Note, that these software systems are excluded as a consequence of *Blech*’s design. This is because they are examples taken from domains which are not addressed by *Blech*. We strongly believe that there cannot be a unique language for all sorts of software domains which provides all the necessary abstractions and performance and at the same time guarantees a safe code generation and deterministic execution. Today’s complex software products often cover a wide range of domains and for each domain the appropriate technology should be used. This is reality in every modern desktop computer: its chipset firmware is written in the appropriate assembly language, operating system primitives are written in C, native applications often use C++ while additional desktop apps may be written in C# or Java; finally web applications will use JavaScript or TypeScript, for example. The same is true for smart phones or tablets. Also we see this trend in complex cyber-physical systems which cross a spectrum of domains: from controlling individual valves and electric drives to complex layered control algorithms, planning and reasoning algorithms and finally communication to other machines or the internet. Each task needs a suitable technology. *Blech* is tailored to be one of them.

6. Available Technology

The chapter deals with the most obvious question about *Blech*: why do you invent a new language instead of simply using XYZ? Of course, embedded systems are developed for decades and there is a large number of languages already in place. We briefly mention the most popular ones and highlight where the difference to *Blech* is.

6.1. The Classics

Hopefully, we have convinced the reader by now that *Blech* offers constructs and guarantees not easily achievable directly in C. It should be clear that our code generation (that emits C code) does more than just expanding a few extra macros. However C++ is often called for when C is considered insufficient. We argue that C++ does not achieve the desired benefits in *embedded software*. While it offers namespaces and classes that allow to better separate individual parts of the software, there are no mechanisms that support the reactive and concurrent nature of the embedded applications considered here.

Ada explicitly addresses embedded programming by providing custom sized data types and a strict type system. Furthermore it offers concurrency as a native language mechanism. However its concurrent entities called *tasks* need to be manually orchestrated. Synchronisation mechanisms have to be correctly used by the programmer to ensure a dead-lock free and race-condition free execution.

Finally the Rust language offers mechanisms to guarantee a deterministic, race and dead-lock free execution of concurrent programs. Yet the purpose of concurrency in Rust is not to decompose a software into concurrent, reactive components as we described in Sect. 2.2. It is used to allow passing information (shared state) among asynchronously running threads. Rust accomplishes its goal well but it is a different goal.

6.2. Model-based engineering

Engineers often use models to develop a solution to a problem. In control oriented domains this model includes the physical behaviour of the plant to be controlled and the controller itself. The idea of model-based engineering is to reuse that model which was

used in the design phase for the subsequent programming phase. Various tools exist to support this approach.

Simulink is a tool that originally was developed to simulate the behaviour of continuous systems described by differential equations. Today it also allows to model discrete-time systems. Special extensions allow to generate executable C code from discrete-time controller models. Additional extensions provide an automata modelling mechanism. The automaton outputs its current state which is used by other blocks to enable or disable some of their functionality. Although Simulink allows to specify the behaviour of blocks using MATLAB script, it is not an IDE in the classical sense: typical tasks of software development such as refactoring are not directly supported by Simulink.

ASCET is another tool for embedded software development, especially for safety-critical software in the automotive domain. Originally, ASCET tried to abstract from C source code by presenting graphical representations of the code to the programmer. Thus in general it is geared more towards programming and less to modelling and simulating of physical processes.

Scade is a tool based on the synchronous language Lustre. It provides a graphical frontend for this data flow oriented language. Automata models may be used similarly to Simulink. Scade has a certified code generator which makes it a popular tool in particularly safety critical domains such as avionics.

All three tools are useful in specific areas of development but we feel they do not fit our goals well. Our goal was to have an imperative programming language—the tool of an embedded developer—enriched with synchronous control flow statements. This allows the programmer to describe behaviour in terms of sequences and concurrent compositions thereof. To better understand this, simply imagine programming our virtual lock in terms of Moore machines with nodes that solve equations on data streams.

Further more, it is important to stress that not every embedded system is developed according to a model-based engineering approach. Thus these tools are not applicable or simply too expensive in many product areas.

In areas where they are applied we see a tendency to use “one tool to rule them all”: for example we often find Simulink models that not only include the actual controller but all sorts of software components around it that do not benefit from a physics simulation. Instead those parts complicate the software development process in terms of reusability, portability, versioning, maintainability and control over the generated code. We believe a more viable approach is to write software in an actual programming language such as *Blech*. This software can then (automatically) be wrapped inside an Functional Mock-up Unit to simulate its interaction with a plant model in Simulink or Modelica, for example.

6.3. Academic Languages

Synchronous programming as we advocate it has a long history already [2]. However, it was rooted in engineering disciplines and initially focused on hardware design. From the software perspective this brought severe limitations: only recently [4] a synchronous language was enabled to update a mutable variable more than once within a reaction.

SCCharts is an actively developed language. Its programs are represented by a state chart notation and by a textual description. The key difference with *Blech* is that in SCCharts the programmer has to model control flow using states and transitions. Every operation is expressed as either a side-effect of a transition between two states or as an action upon entry, exit or reactivation of a state. *Blech* instead uses the usual control flow statements known from standard imperative programming and extends the set of statements to express reactions, synchronous concurrency and preemptions. Thus the languages feel very different from a practical software engineering perspective.

Lustre is the textual language upon which the Scade tool was built, see above.

Céu has been designed in the tradition of Esterel as an imperative synchronous language. But although it allows the concurrent composition of blocks as we do with `cobegin`, Céu carries out no causality analysis and simply executes a step in every block in lexicographical order. While this seems like a little difference it has great impacts on the design of the language and the programs written in it. Céu does not care whether two branches overwrite each others shared variables. It also has no notion of a read-write cycle. Moving the branches (in a refactoring step) may change the behaviour of the Céu program—this cannot happen in *Blech*.

Despite many crucial differences, clearly the development of the *Blech* language has been influenced by all these languages.

7. Outlook

Here we briefly give a list of features that we have conceptually developed but not yet fully implemented in our compiler or other tooling. Our intention is to give an impression on what other benefits *Blech* may bring besides what was discussed specifically in this tutorial.

We invite everyone who is interested to collaborate with us and profit from all of these possible features! Remember that in open source projects naturally those features emerge fast which most people are working on.

Further details on the individual topics may be found in the language evolution (`https://www.blech-lang.org/docs/language-evolution/`) or blog (`https://www.blech-lang.org/blog/`) sections of the *Blech* website. Feel free to get involved in the discussions on Github issues, Slack, the mailing list, or simply drop us an e-mail.

1. Multi-clock *Blech* and parallel programming
2. Completing synchronous control flow
    - Error handling
    - Preemption handling and clean-up code
    - Communicating events with signals
3. Completing data types for synchronous programming
    - Enumeration types
    - References and reference types
    - Physical dimensions
    - Generic data types
    - Strings
4. Module system
5. Testing
    - Host versus target development
    - Unit testing

    - Regression tests
    - Software in the loop
6. Tooling
    - Debugging
    - Build process

Bibliography

[1] The SCODE method. https://www.etas.com/en/products/scode-analyzer-scode-methode.php. Accessed: 2019-09-06.

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[3] Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. The synchronous hypothesis and synchronous languages. In *Embedded Systems Handbook*. 2005.

[4] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha S. Roop. Sequentially constructive concurrency - A conservative extension of the synchronous model of computation. *ACM Trans. Embedded Comput. Syst.*, 13(4s):144:1–144:26, 2014.

A. Sample solutions

Solution for Section 4.2 The runtime environment is implemented in Mita as follows.

```
1  package main;
2  import platforms.xdk110;
3
4  native unchecked fn blc_blech_virtualSafeLock_tick(blc_entry_x: int32,
5                           blc_entry_y: int32,
6                           blc_entry_z: int32,
7                           blc_entry_pressedOne: bool,
8                           blc_entry_pressedTwo: bool,
9                           blc_entry_mlux: int32,
10                          blc_entry_ledLeft: &bool,
11                          blc_entry_ledMiddle: &bool,
12                          blc_entry_ledRight: &bool) : void
13   header "blech/virtualSafeLock.c";
14
15 native unchecked fn blc_blech_virtualSafeLock_init() : void
16   header "blech/virtualSafeLock.c";
17
18 var globONEhasBeenPressed = false;
19 var globTWOhasBeenPressed = false;
20
21 every button_one.pressed {
22   globONEhasBeenPressed = true;
23 }
24
25 every button_two.pressed {
26   globTWOhasBeenPressed = true;
27 }
28
29 setup led : LED {
30     var right = light_up(color = Red);
31     var middle = light_up(color = Orange);
32     var left = light_up(color = Yellow);
33 }
34
35 every XDK110.startup {
36   blc_blech_virtualSafeLock_init();
37 }
38
39 every 100 milliseconds {
40   var x = accelerometer.x_axis.read();
41   var y = accelerometer.y_axis.read();
```

```
  var z = accelerometer.z_axis.read();

  let mlux = light.intensity.read();

  var ledR: bool = led.right.read();
  var ledM: bool = led.middle.read();
  var ledL: bool = led.left.read();

  blc_blech_virtualSafeLock_tick(x, y, z,
    globONEhasBeenPressed, globTWOhasBeenPressed,
    (mlux as int32),
    &ledL, &ledM, &ledR
  );

  globONEhasBeenPressed = false;
  globTWOhasBeenPressed = false;

  led.right.write(ledR);
  led.middle.write(ledM);
  led.left.write(ledL);
}
```

The *Blech* program calls a `Blink` activity which inverts the status of the LEDs every 0.5 s. The delay is achieved by the `CountDown` activity which terminates after the given number of ticks.

```
/// invert LEDs' status values
function invertLEDs () (ledLeft: bool, ledMiddle: bool, ledRight: bool)
    ledRight = not ledMiddle
    ledLeft = not ledMiddle
    ledMiddle = not ledMiddle
end

/// When called, delays execution for a given number of ticks
activity CountDown (ticks: nat32)
    var steps = ticks
    repeat
        await true
        steps = steps - 1
    until steps <= 0 end
end

/// Invert the status of all LEDs every half a second
activity Blink () (ledLeft: bool, ledMiddle: bool, ledRight: bool)
    repeat
        invertLEDs()(ledLeft, ledMiddle, ledRight)
        run CountDown(5) // do nothing for 5 ticks = 0.5s
    end
end

@[EntryPoint]
activity XDKBlinking_LEDs (x: int32, y: int32, z: int32, pressedOne: bool,
                           pressedTwo: bool, mlux: int32)
                          (ledLeft: bool, ledMiddle: bool, ledRight: bool)
```

```
29      run Blink()(ledLeft, ledMiddle, ledRight)
30  end
```

Solution for Section 4.3 Here is the complete listing, including helper functions provided as parts of the code skeleton. The crucial activities to be implemented here were `entry`, `Locked`, `Unlock`, and `Success`.

```
1   /*****************************************************************************
2    * Global constants
3    *****************************************************************************/
4   const OneG: int32 = 4095 // acceleration value from sensor which we
5                            // consider to be at least 1g (gravitation force)
6   const PositionEpsilon: int32 = 400  // 10% epsilon
7   const Cos45xG: int32 = 2895 // cos(45°) * OneG = sin(45°) * OneG
8
9   /// We encode pose information by a prime number encoding
10  /// For example: we represent the XDK standing upright (12 o'clock) as
11  /// NORTH * EXACT = 2 * 11 = 22
12  const UNDEFPOS: nat32 = 1
13
14  const NORTH: nat32 = 2
15  const EAST: nat32 = 3
16  const SOUTH: nat32 = 5
17  const WEST: nat32 = 7
18
19  const EXACT: nat32 = 11
20  const RIGHTOF: nat32 = 13
21  const LEFTOF: nat32 = 17
22
23  /// The maximum length of the secret
24  const MAXLEN: nat32 = 8
25
26
27  /*****************************************************************************
28   * Helpers
29   *****************************************************************************/
30  /// invert LEDs' status values
31  function invertLEDs () (ledLeft: bool, ledMiddle: bool, ledRight: bool)
32      ledRight = not ledMiddle
33      ledLeft = not ledMiddle
34      ledMiddle = not ledMiddle
35  end
36
37  /// indicate succesfully entered secret
38  function successToLEDs ()  (ledLeft: bool, ledMiddle: bool, ledRight: bool)
39      ledRight = true
40      ledLeft = true
41      ledMiddle = true
42  end
43
44  /// true if the device is put face down on the table
45  function faceDownOnTheTable(z: int32, mlux: int32) returns bool
```

```
    return mlux < 18000 and z < -900
end

/// value is +- PositionEpsilon around point
function around (v: int32, p: int32) returns bool
    // abs(p-v) <= epsilon
    if v <= p then
        return p-v <= PositionEpsilon
    else
        return v-p <= PositionEpsilon
    end
end

function isExact (nearG: int32, nearZero: int32) returns bool
    return nearG >= OneG - PositionEpsilon
        and around(nearZero, 0)
end

function isRightOf (opposite: int32, adjacent: int32) returns bool
    return OneG > opposite
        and opposite > Cos45xG
        and PositionEpsilon < adjacent
        and adjacent < Cos45xG
end

function isLeftOf (opposite: int32, adjacent: int32) returns bool
    return isRightOf(opposite, -adjacent)
end

/// Determine proximity of given vector to South direction
function isSouthAligned (x: int32, y: int32) returns nat32
    if isExact(x, y) then
        return EXACT
    elseif isRightOf(x, y) then
        return RIGHTOF
    elseif isLeftOf(x, y) then
        return LEFTOF
    else
        return UNDEFPOS
    end
end

/// point symmetric to isSouthAligned
function isNorthAligned (x: int32, y: int32) returns nat32
    return isSouthAligned(-x, -y)
end

/// map to isSouthAligned by rotation
function isEastAligned (x: int32, y: int32) returns nat32
    return isSouthAligned(-y, x)
end

/// point symmetric to isEastAligned
function isWestAligned (x: int32, y: int32) returns nat32
```

```
100      return isEastAligned(-x, -y)
101  end
102
103  /// Determines the XDK's pose given the x and y values of the accelerometer
104  function determineOrientation (x: int32, y: int32) returns nat32
105      // check every direction and take the first that gives a defined
         alignment
106      var alignment = isNorthAligned(x, y)
107      if UNDEFPOS != alignment then return alignment * NORTH end
108      alignment = isEastAligned(x, y)
109      if UNDEFPOS != alignment then return alignment * EAST end
110      alignment = isSouthAligned(x, y)
111      if UNDEFPOS != alignment then return alignment * SOUTH end
112      alignment = isWestAligned(x, y)
113      if UNDEFPOS != alignment then return alignment * WEST end
114      return UNDEFPOS
115  end
116
117  /// Given a pose sets LED to reflect the alignment
118  function poseToLED (pose: nat32)
119                      (ledLeft: bool, ledMiddle: bool, ledRight: bool)
120      ledMiddle = false
121      ledRight = false
122      ledLeft = false
123      if pose % EXACT == 0 then
124          ledMiddle = true
125      elseif pose % RIGHTOF == 0 then
126          ledRight = true
127      elseif pose % LEFTOF == 0 then
128          ledLeft = true
129      end
130  end
131
132  /// Given a pose, tells if it is exactly aligned
133  function poseIsExact (pose: nat32) returns bool
134      return pose % EXACT == 0
135  end
136
137  /*****************************************************************************
138   * Misc helper activities
139   *****************************************************************************/
140
141  /// In every tick: given accelerometer sensor readings
142  /// sets LED to reflect the pose
143  activity DisplayOrientation (x: int32, y: int32)
144                              (ledLeft: bool, ledMiddle: bool, ledRight: bool,
145                               pose: nat32)
146      repeat
147          pose = determineOrientation(x, y)
148          poseToLED(pose)(ledLeft, ledMiddle, ledRight)
149          await true
150      end
151  end
152
```

```
153 /// When called, delays execution for a given number of ticks
154 activity CountDown (ticks: nat32)
155     var steps = ticks
156     repeat
157         await true
158         steps = steps - 1
159     until steps <= 0 end
160 end
161
162
163 /// Invert the status of all LEDs every half a second
164 activity Blink () (ledLeft: bool, ledMiddle: bool, ledRight: bool)
165     repeat
166         invertLEDs()(ledLeft, ledMiddle, ledRight)
167         run CountDown(5) // do nothing for 5 ticks = 0.5s
168     end
169 end
170
171 /*****************************************************************************
172  * Activities (representing modes)
173  *****************************************************************************/
174
175 activity EnterSecret (secret: [MAXLEN]nat32, pose: nat32, pressedOne: bool)
        returns bool
176     var idx: nat32 = 0
177     var isOk = true
178     repeat
179         await pressedOne
180         if poseIsExact(pose) then
181             if pose == secret[idx] then
182                 idx = idx + 1
183                 if idx < MAXLEN and secret[idx] == UNDEFPOS then // guard
        array access
184                     idx = MAXLEN // skip the rest
185                 end
186             else
187                 isOk = false
188             end
189         end
190         // else inexact position, do not evaluate
191     until not isOk or idx == MAXLEN end
192     return isOk
193 end
194
195 /// Contains the process of unlocking the virtual lock
196 /// Returns true iff lock has been opened successfully
197 activity Unlock (x: int32, y: int32, pressedOne: bool)
198                 (ledLeft: bool, ledMiddle: bool, ledRight: bool)
199                 returns bool
200     var secret: [MAXLEN]nat32 = { EXACT * NORTH, EXACT * EAST, EXACT * WEST,
201                                   EXACT * SOUTH, UNDEFPOS, UNDEFPOS,
202                                   UNDEFPOS, UNDEFPOS }
203     var pose: nat32
204     var isOk = false
```

```
205      cobegin weak
206          run DisplayOrientation(x, y)(ledLeft, ledMiddle, ledRight, pose)
207      with
208          isOk = run EnterSecret(secret, pose, pressedOne)
209      end
210      return isOk
211  end
212
213  /// In the locked mode, keep blinking until the user presses button 1
214  activity Locked (pressedOne: bool)
215                  (ledLeft: bool, ledMiddle: bool, ledRight: bool)
216      when pressedOne abort // Button 1: start unlocking
217          run Blink()(ledLeft, ledMiddle, ledRight)
218      end
219  end
220
221  /// Lock has been successfully opened
222  /// Determine exclusively pressed button
223  activity Success (pressedOne: bool)
224                   (ledLeft: bool, ledMiddle: bool, ledRight: bool)
225      successToLEDs()(ledLeft, ledMiddle, ledRight)
226
227      await pressedOne
228  end
229
230  /*****************************************************************************
231   * Program starts here
232   *****************************************************************************/
233  @[EntryPoint]
234  activity XDKUnlocking (x: int32, y: int32, z: int32, pressedOne: bool,
235                         pressedTwo: bool, mlux: int32)
236                        (ledLeft: bool, ledMiddle: bool, ledRight: bool)
237      repeat
238          // I.
239          run Locked(pressedOne)(ledLeft, ledMiddle, ledRight)
240
241          // II.
242          var successful = false
243          successful = run Unlock(x, y, pressedOne)
244                                 (ledLeft, ledMiddle, ledRight)
245
246          if successful then
247              // III.
248              run Success(pressedOne)(ledLeft, ledMiddle, ledRight)
249          end
250      end
251  end
```

Solution for Section 4.4 In the main activity we wrap the entire contents of the loop inside an `abort` statement. Thus no matter where the program resumes to perform a reaction, if the user made the “reset gesture” (`faceDownToTable`) then the control flow

jumps to the end of the abort in the main activity, and the loop brings us back to the initial mode.

```
/// true if the device is put face down on the table
function faceDownOnTheTable(z: int32, mlux: int32) returns bool
    return mlux < 18000 and z < -900
end

@[EntryPoint]
activity XDKCancellation(x: int32, y: int32, z: int32, pressedOne: bool,
                         pressedTwo: bool, mlux: int32)
                        (ledLeft: bool, ledMiddle: bool, ledRight: bool)
    repeat
        // abort when the device is put face down on the table
        when faceDownOnTheTable(z, mlux) abort
            // I.
            run Locked(pressedOne)(ledLeft, ledMiddle, ledRight)

            // II.
            var successful = false
            successful = run Unlock(x, y, pressedOne)
                                   (ledLeft, ledMiddle, ledRight)

            if successful then
                // III.
                run Success(pressedOne)(ledLeft, ledMiddle, ledRight)
            end
        end
    end
end
```

Solution for Section 4.5 We add a new activity `Programming` that lets the user enter a new secret. It uses `EnterNewSecret` as a helper.

```
activity EnterNewSecret (pose: nat32, pressedOne: bool, pressedTwo: bool)
                        (newSecret: [MAXLEN]nat32) returns bool
    var idx: nat32 = 0
    cobegin weak
        repeat
            await pressedOne and not pressedTwo
            if poseIsExact(pose) then
                newSecret[idx] = pose
                idx = idx + 1
            end
            // else inexact position, do not evaluate
        until idx == MAXLEN end
    with weak
        await pressedTwo and not pressedOne // finish programming
    end
    return idx > 0 // at least one position has been entered
end

```

```
19  /// The process of setting a new secret in the virtual lock
20  activity Programming (x: int32, y: int32, pressedOne: bool, pressedTwo: bool)
21                       (secret: [MAXLEN]nat32, ledLeft: bool, ledMiddle: bool,
        ledRight: bool)
22                       returns bool
23      var pose: nat32
24      var newSecret: [MAXLEN]nat32 = { UNDEFPOS, UNDEFPOS, UNDEFPOS, UNDEFPOS,
25                                       UNDEFPOS, UNDEFPOS, UNDEFPOS, UNDEFPOS }
26      var isOk = false
27
28      cobegin weak
29          run DisplayOrientation(x, y)(ledLeft, ledMiddle, ledRight, pose)
30      with
31          isOk = run EnterNewSecret(pose, pressedOne, pressedTwo)(newSecret)
32      end
33      if isOk then
34          secret = newSecret
35      end
36      return isOk
37  end
```

The `Success` activity is extended to distinguish whether the user wants to simply close the lock or to set a new secret.

```
1   /// Lock has been successfully opened
2   /// Determine exclusively pressed button
3   activity Success (pressedOne: bool, pressedTwo: bool)
4                    (ledLeft: bool, ledMiddle: bool, ledRight: bool)
5                    returns bool
6       successToLEDs()(ledLeft, ledMiddle, ledRight)
7
8       await pressedOne and not pressedTwo
9             or pressedTwo and not pressedOne //exactly one button is pressed
10
11      if pressedTwo then
12          return true // indicate that we want to reprogram the secret
13      else
14          return false // Button 1 leads back to start
15      end
16  end
```

The `Unlock` activity is changed to take the secret as input parameter instead of defining it as a local variable.

```
1   activity Unlock (secret: [MAXLEN]nat32, x: int32, y: int32, pressedOne:bool)
2                   (ledLeft: bool, ledMiddle: bool, ledRight: bool)
3                   returns bool
4       var pose: nat32
5       var isOk = false
6       cobegin weak
7           run DisplayOrientation(x, y)(ledLeft, ledMiddle, ledRight, pose)
8       with
9           isOk = run EnterSecret(secret, pose, pressedOne)
10      end
```

```
    return isOk
end
```

We adapt the main activity accordingly.

```
@[EntryPoint]
activity XDKUserDefinedCode (x: int32, y: int32, z: int32, pressedOne: bool,
                             pressedTwo: bool, mlux: int32)
                            (ledLeft: bool, ledMiddle: bool, ledRight: bool)
    var secret: [MAXLEN]nat32 = { EXACT * NORTH, EXACT * EAST, EXACT * WEST,
                                  EXACT * SOUTH, UNDEFPOS, UNDEFPOS,
                                  UNDEFPOS, UNDEFPOS }
    repeat
        // abort when the device is put face down on the table
        when faceDownOnTheTable(z, mlux) abort
            // I.
            run Locked(pressedOne)(ledLeft, ledMiddle, ledRight)

            // II.
            var successful = false
            successful = run Unlock(secret, x, y, pressedOne)
                                  (ledLeft, ledMiddle, ledRight)
            if successful then

                // III.
                var wantReprogramming = false
                wantReprogramming = run Success(pressedOne, pressedTwo)
                                                (ledLeft, ledMiddle, ledRight)
                if wantReprogramming then

                    // IV.
                    _ = run Programming(x, y, pressedOne, pressedTwo)
                                       (secret, ledLeft, ledMiddle, ledRight)
                end
            end
        end
    end
end
```

Solution for Section 4.6 All we need to do is prevent early termination. Thus keep reading up to 8 positions. Return the value of `hasUnlockingSucceeded` only when button two is pressed. However this boolean variable is true only when the last correct pose has been entered and nothing else.

```
activity EnterFullSecret (secret: [MAXLEN]nat32, pose: nat32,
                          pressedOne: bool, pressedTwo: bool) returns bool
    var idx: nat32 = 0
    var codeOk = true
    cobegin weak
        repeat
            await pressedOne and not pressedTwo
            if poseIsExact(pose) then
```

```
 9                  if idx >= MAXLEN then  // entered code to long
10                      codeOk = false
11                  elseif secret[idx] == UNDEFPOS then // entered code too long
12                      codeOk = false
13                  elseif pose != secret[idx] then  // entered pose incorrect
14                      codeOk = false
15                  end
16                  idx = idx + 1
17                  // else inexact position, do not evaluate
18              end
19          end
20      with
21          await pressedTwo and not pressedOne
22      end
23      if idx >= MAXLEN or secret[idx] == UNDEFPOS then  // enough poses entered
24          return codeOk
25      else
26          return false
27      end
28  end
29
30  /// Contains the process of unlocking the virtual lock
31  /// Returns true iff lock has been opened successfully
32  activity Unlock (secret: [MAXLEN]nat32, x: int32, y: int32,
33                   pressedOne: bool, pressedTwo: bool)
34                  (ledLeft: bool, ledMiddle: bool, ledRight: bool)
35                  returns bool
36
37      var pose: nat32
38      var isOk = false
39      cobegin weak
40          run DisplayOrientation(x, y)(ledLeft, ledMiddle, ledRight, pose)
41      with
42          isOk = run EnterFullSecret(secret, pose, pressedOne, pressedTwo)
43      end
44      return isOk
45  end
```