# ‘External’ tutorial

Version 5

Dezyne version 2.3.3

June 16th, 2017

# Introduction

## Learning goals

After following this tutorial, you will be able to:

* Understand when to use ‘external’ and what problems it solves
* Understand the impact of using the ‘external’ keyword in your Dezyne models
* Identify where to implement additional logic in your models to support the usage of ‘external’
* Use the Dezyne verification & simulation tools to correctly implement a solution using the ‘external’ keyword

## Intended audience and prerequisites

As this tutorial will build upon the basic features of Dezyne, it is assumed that you are familiar with the Dezyne syntax and the use of Dezyne verification and simulation tools. It is helpful if you have some knowledge on the Dezyne runtime, which can be found in the previous tutorial <https://www.verum.com/support/code-integration-tutorial/>.

This tutorial will consist of two parts: first we will disclose some information on how simulation and verification by Dezyne works and what the effects are of using ‘external’. The second part of the tutorial will focus on designing a solution for the problems that can be found with ‘external’.

The nature of ‘external’ requires you to be able to think in terms of threads and sequencing of events across multiple threads. You will be assisted in this during the tutorial, but it helps if you are familiar with the concepts.

This tutorial will build further upon the Alarm System models and native implementation from the previous tutorials. With ‘external’, we will be able to discover interesting real-world behavior in the Alarm System that can lead to illegality in its components.

## Platform choice

As most of the work can be done in the Dezyne modeling language and only minimal changes are required on the platform, the platform choice from previous tutorials remains unchanged. Raspbian with g++ 4.9.2 on the Raspberry Pi supports all language requirements for using a thread-safe-shell in C++11.

Table of Contents

[Introduction 1](#_Toc485111053)

[Learning goals 1](#_Toc485111054)

[Intended audience and prerequisites 1](#_Toc485111055)

[Platform choice 1](#_Toc485111056)

[Chapter 1: Why use external? 3](#_Toc485111057)

[What can go wrong? 3](#_Toc485111058)

[How does this translate to runtime execution? 4](#_Toc485111059)

[Why does verification not catch this? 5](#_Toc485111060)

[What does ‘external’ do and when should you use it? 5](#_Toc485111061)

[Chapter 2: How to use ‘external’? 7](#_Toc485111062)

[Responsibilities and using ‘external’ components 7](#_Toc485111063)

[How to start solving the problem you face with ‘external’ 8](#_Toc485111064)

[Implementing the solution as component behaviour 11](#_Toc485111065)

[‘The’ solution: handshake protocol 12](#_Toc485111066)

[Chapter 3: Finishing the Alarm System 16](#_Toc485111067)

[Updating the Controller 16](#_Toc485111068)

[Updating the native Timer 19](#_Toc485111069)

[Chapter 4: Retrospect on learning goals 21](#_Toc485111070)

[Understand when to use ‘external’ and what problems it solves 21](#_Toc485111071)

[Understand the impact of using the ‘external’ keyword in your Dezyne models 21](#_Toc485111072)

[Identify where to implement additional logic in your models to support the usage of ‘external’ 21](#_Toc485111073)

[Use the Dezyne verification & simulation tools to correctly implement a solution using the ‘external’ keyword 21](#_Toc485111074)

# Chapter 1: Why use external?

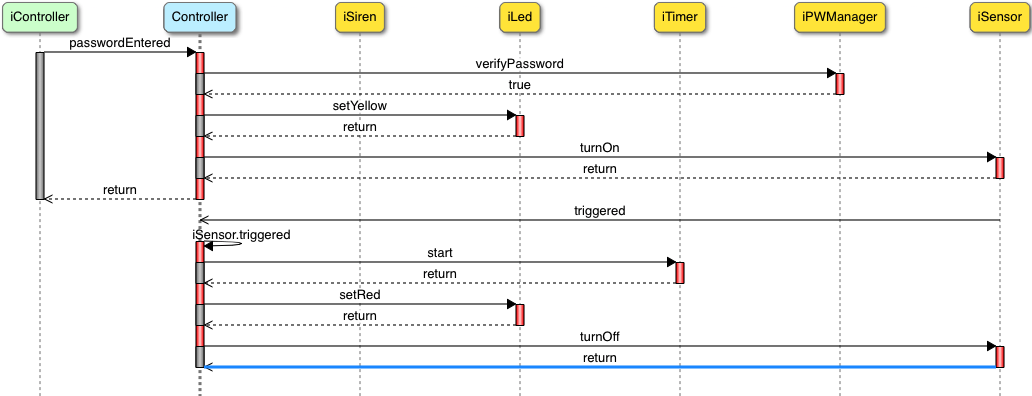
In this chapter, we will discuss a scenario of the Alarm System’s behavior where a delay in communication between two components results in a race condition that ultimately leads to illegal behavior. With this in mind, we will examine the Dezyne verification process and why standard verification does not find the race condition we discovered.

With this information, you should be able to understand when to use ‘external’ and what problems it can solve, as well as the impact of using ‘external’ in your Dezyne models.

## What can go wrong?

For the example in this chapter, a snapshot containing Dezyne models and C++ source code is available on <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/External/Ch1_Alarm_System>.

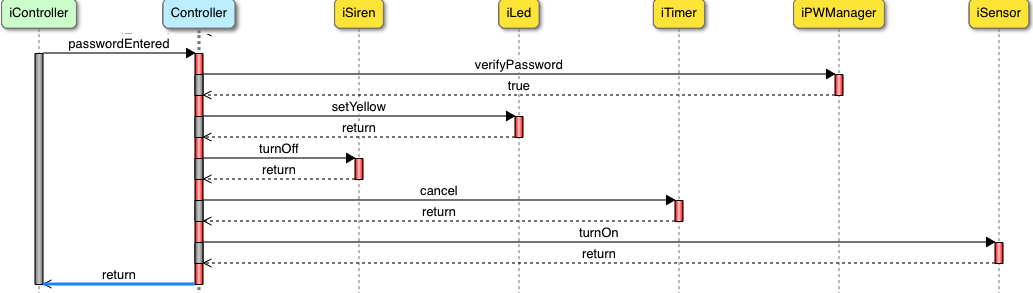
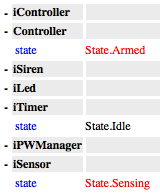
Consider the following situation:



In the above sequence diagram, a valid password has been entered on the IController interface. This resulted in the Controller moving to the Armed state, turning on the Sensor as well.

At this point the Sensor was triggered, resulting in the starting of the Timer as well as the Controller transitioning to the Alarming state. The corresponding Watch window with this sequence of events is included to the right and it confirms that the Controller is in the Alarming state and that the Timer is Running.

Say we were to perform another passwordEntered event, with a valid password. The following sequence will occur:



So far so good, right? The Dezyne model encounters no illegal behavior and the Watch window tells us that the Controller has moved back to the Armed state, ITimer is Idle again and the Sensor is Sensing.

An **important remark** to make at this point is that the passwordEntered event from iController led to a series of events that took an arbitrary amount of time, *x*. This is important to keep in mind for the next paragraph.

## How does this translate to runtime execution?

In order to fully understand what can happen, it’s important to replay the above scenario in real-world context where time does matter. Recall that the code generated from the System required a **thread-safe-shell** to guarantee the single-thread-active property of the application (<https://www.verum.com/supportitem/code-integration-extra-materials/>). This was done by using a dzn::pump to schedule events on a First In, First Out (FIFO) basis for a private thread to handle.

After the first valid passwordEntered event, the Controller is Armed. If Armed, the triggered event from the Sensor leads to the Controller starting the timer. Let’s call this point **t = 0**. The timer is configured to timeout after 30 seconds as specified in the behavior, so the timeout event will be scheduled by the native Timer thread at **t = 30**.

Earlier it was noted that the execution time of the second passwordEntered event is *x*. Say a valid password is entered *just* before **t = 30**. It is now a possibility that during the execution *x,* **t = 30** occurs which results in a timeout event being scheduled. However, after *x*, the Controller is in the Armed state. If you look at the behavior of the Controller component in the Armed state, a timeout event cannot be handled and as such is implicitly illegal:

[state.Armed] {

  on iController.passwordEntered(pw): {

    bool valid = iPWManager.verifyPassword(pw);

    if(valid) {

state = State.Unarmed;

iLed.setGreen();

iSensor.turnOff();

  }

  }

  on iSensor.triggered(): {

    state = State.Alarming;

    iTimer.start($30000$);

    iLed.setRed();

  iSensor.turnOff();

}

}

## Why does verification not catch this?

So now that we have considered a sequence of events in context of runtime behavior, we have found a trace that leads to a race condition triggering illegal behavior. Why doesn’t Dezyne show us this illegal behavior? If you verify the Controller component, it passes every test just fine. If you simulate the Controller component, you will not be able to send a timeout event during the handling of the passwordEntered event.

The reason behind this is that Dezyne verification does not account for the time it takes to execute a sequence of events. During verification, it is assumed that the execution of events occurs instantaneously. If events occur instantaneously, there can be no concurrency of events- there is no timespan in which multiple processes are active. As such, there is no time period *x* in the verification during which the timeout event can be scheduled. We have found a discrepancy between the behavior verified by Dezyne and the behavior of the generated code in a real-world environment: **concurrency**.

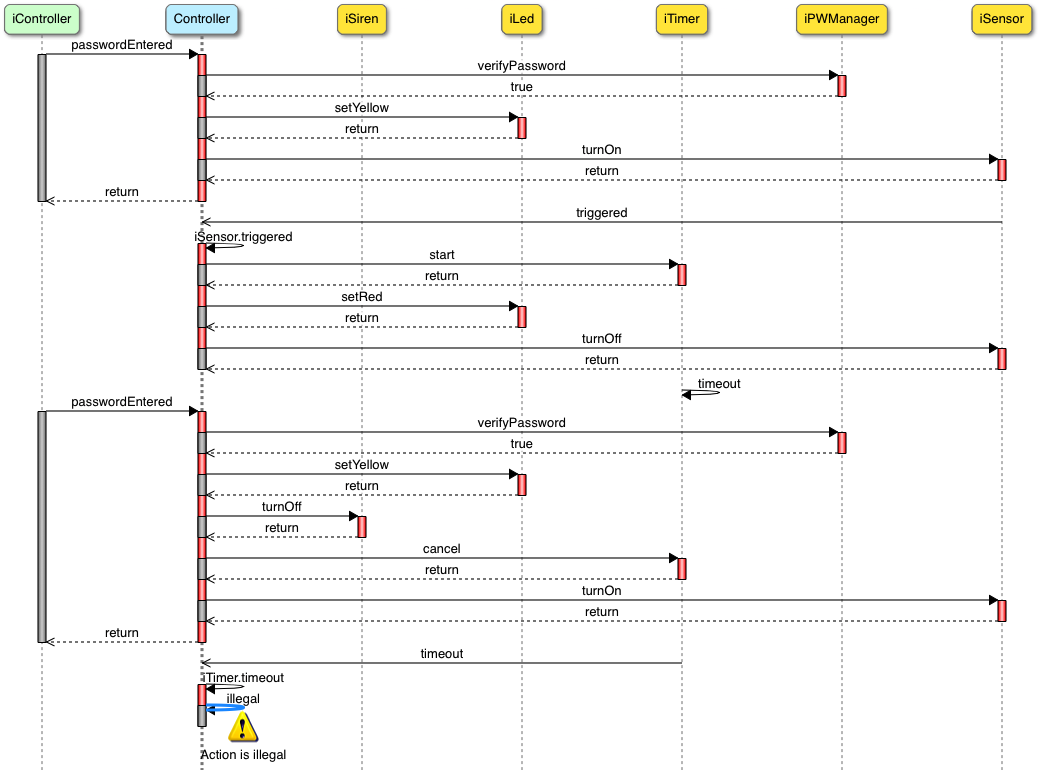
The absence of concurrency in the verified scenario means that only synchronous traces of events are considered. However, the scenario we found relies on asynchronous behavior of one of the native components. This asynchronous behavior can only come from a component that runs on its own execution thread. Using the ‘external’ keyword, you can indicate that the implementing component is active outside of the Dezyne private thread and therefore, extra behaviors must be considered by the Dezyne verification and simulation tools.

## What does ‘external’ do and when should you use it?

The ‘external’ keyword is a modifier for requires interfaces of a component. Marking an interface as ‘external’ will increase its set of possible event sequences in the verification and simulation tools of Dezyne. More specifically, indicating that a requires interface is ‘external’ tells Dezyne that it should account for delays that can occur in communication through the interface. If the interface is stateful, this means that the two components on either end of the interface can get out of sync regarding the state of the protocol they use to communicate.

In the example with ITimer in the previous chapter, the Controller considers the state of Timer is Running when the cancel event is sent. However, the Timer has already timed out and has queued the timeout event accordingly. This is a textbook example of two components, Controller and Timer, being out of sync in regard to their perceived state of the protocol they use to communicate (ITimer). The result is that an unsuspecting Controller still receives a timeout it must handle even though the Controller is under the assumption that the Timer had been canceled.

If you modify the ITimer port of the Controller to requires external ITimer iTimer; and verify the Controller, you will actually find the illegal assert:



So, when should you use ‘external’? Right now, it might seem tempting to make every required port that sends out events ‘external’. However, its usage is not always warranted and as such you might be doing a lot of work for no benefit at all. The general rule of thumb:

**Mark as ‘external’ any requires interface containing out-events that is delegated to the boundary of a system.** Take note that the ‘external’ marking must be done in a behavior component, not a system component.

If a requires interface with out-events is implemented in a Dezyne component contained in the same system, the scheduling of said out-events will be done on the private Dezyne thread during runtime. The semantics enforced by the included Dezyne runtime simply prevent the component from scheduling any out-events while another component is

executing its behaviour. Therefore, no race conditions between state and asynchronous events **within** the generated Dezyne framework can exist.

For more information on the Dezyne runtime semantics, please refer to <https://www.verum.com/supportitem/execution-semantics/>.

With proper inclusion of ‘external’ for components implemented outside of Dezyne, verification and simulation will once again consider the full scope of the possible behaviours of your application. As you saw in the previous sequence view, though, you have some work to do in the Controller component so that it can function correctly with an ‘external’ ITimer port. In the next chapter, we will explore some changes you can make to the implementation of a component to allow for ‘external’ behaviour.

# Chapter 2: How to use ‘external’?

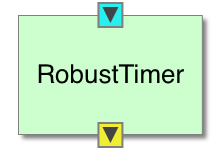
In this chapter, we will explore some possible changes you can make in component behaviour to support the usage of ‘external’ for required interfaces. In a few steps, you will discover:

* Where to build your solution
* How to start solving the problem you face with ‘external’
* Implementing the solution as component behaviour

## Responsibilities and using ‘external’ components

To make it easier to accommodate for concurrent behaviours found in ‘external’ ports, it is recommended to define **another component** that **provides** the ITimer interface and **requires** an external ITimer interface. Then, in the System component, bind the provided ITimer port of the new component to the required ITimer port of the Controller. From the Controller point of view, the ITimer port is no longer ‘external’. Instead, the new component (let’s call it RobustTimer) becomes responsible for translating the ‘external’ ITimer port to a regular ITimer port.

If you’ve performed the steps above, the situation should now look like this:



component RobustTimer {

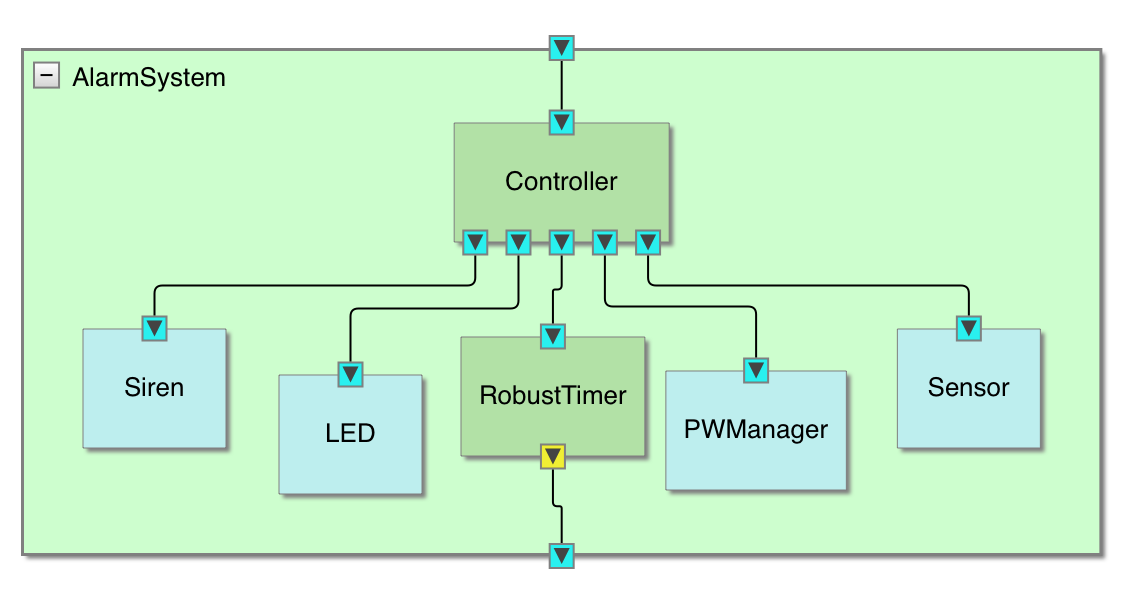
provides ITimer iTimer;

requires external ITimer ext\_iTimer;

behaviour {

}

}



Note that the required ‘external’ ITimer port on RobustTimer is coloured yellow; this is the case for all ports that are marked ‘external’. It is an easy way to identify these ports when looking at the System view. Note that the required ITimer port in Controller is no longer marked ‘external’.

## How to start solving the problem you face with ‘external’

The RobustTimer component will be responsible for the mapping of an ‘external’ ITimer port to a regular ITimer port. The intent is to define the behaviour of the component in such a way that even if ‘external’ delays occur, communication over the provided ITimer port does not suffer from this.

The most straight-forward behaviour specification would be to map in-events on the provided ITimer port to in-events on the required ITimer port and to do the same for out-events of the required port. As a first exercise, implement the behaviour of RobustTimer this way.

Hint: make sure the states defined in the ITimer interface are followed in the RobustTimer component behaviour as well.

Solution:

component RobustTimer {

provides ITimer iTimer;

requires external ITimer ext\_iTimer;

behaviour {

enum State { Idle, Running };

State state = State.Idle;

on iTimer.start(ms): {

[state.Idle] {

ext\_iTimer.start(ms);

state = State.Running;

}

}

on iTimer.cancel(): {

[state.Running] {

ext\_iTimer.cancel();

state = State.Idle;

}

[state.Idle] { }

}

on ext\_iTimer.timeout(): {

[state.Running] {

iTimer.timeout();

state = State.Idle;

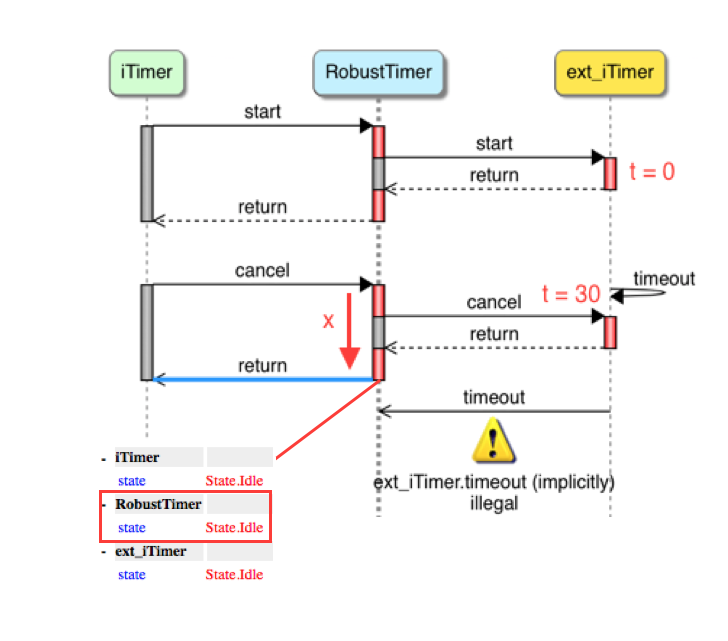
}

}

}

}

The above solution is a first rudimentary step towards a working solution. If you verify the RobustTimer component, you will see the exact same illegal assertion as we found earlier in the previous chapter. However, this time it is extracted from all interleaving with other required and provided ports of the Controller component and you can focus solely on the behaviour of the required ‘external’ ITimer port. Let’s reconsider the timeline described in How does this translate to runtime execution?:



Recall that *x* is the time it takes for the cancel event from iTimer to be processed. During *x*, the timeout occurs on ext\_iTimer and is queued accordingly. After *x*, RobustTimer is in the Idle state but still has to handle the queued timeout. This scenario is not described in the behaviour of RobustTimer and as such, it is implicitly illegal.

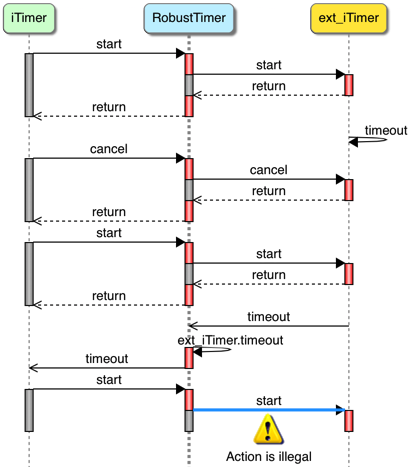
In the next chapter, we will have a look at how to implement behaviour such that these possible concurrencies discovered by ‘external’ can be handled.

## Implementing the solution as component behaviour

Now that the RobustTimer component has been created and the rudimentary communication between the provided ITimer and required ‘external’ ITimer has been established, it is time to start working towards a functional solution that is verifiably correct.

A logical first step could be to specify that when ext\_iTimer sends a timeout event to the RobustTimer while RobustTimer is not in a Running state, the timeout event is ignored (so no operation occurs). This ensures that within the RobustTimer component, possibly delayed timeouts do not trigger illegal behaviour if the cancel event had already occurred.

As an exercise, make the aforementioned change to the handling of ext\_iTimer.timeout() and verify the RobustTimer again. Try to figure out what has happened in the scenario that the verifier shows you before moving on.

Solution:

component RobustTimer {

provides ITimer iTimer;

requires external ITimer ext\_iTimer;

behaviour {

enum State { Idle, Running };

State state = State.Idle;

on iTimer.start(ms): {

[state.Idle] {

ext\_iTimer.start(ms);

state = State.Running;

}

}

on iTimer.cancel(): {

[state.Running] {

ext\_iTimer.cancel();

state = State.Idle;

}

[state.Idle] { }

}

on ext\_iTimer.timeout(): {

[state.Running] {

iTimer.timeout();

state = State.Idle;

}

[state.Idle] {}

}

}

}

The scenario shown to you by the verifier is one where a ‘timeout’ event sent in response to the first ‘start’ is misinterpreted by RobustTimer as if it was sent in response to the second ‘start’ event; however, ext\_iTimer is Running while the timeout event is processed. After propagating the timeout event to iTimer, iTimer is Idle once again, allowing for a start event to occur. While ext\_iTimer is Running, start events are illegal.

Nonetheless, the length of the trace has increased, indicating that we have passed the first hurdle of illegal timeouts after the timer had already been cancelled. Progress has been made, but we’re not quite there yet.

We need to implement some way to distinguish between outdated timeout events and timeout events from the timer that was currently started. Let’s refer to the time window between a timer start and a timeout or cancel as a *session*. There are two ways to approach this situation: keep track of the current session and ignore others, or implement a handshake to ensure that a session is closed before starting a new one.

## ‘The’ solution: handshake protocol

The recommended option is to implement a handshake mechanism to ensure that the current session is closed. If you recall, the original problem we discovered was that it was possible to queue a timeout event while the timer was being cancelled. The description of the problem already hints at part of the solution: apparently there exists a state where the Timer is *being cancelled*. This state starts when the cancel event is sent and ends at some point in time.

When the *being cancelled* state has ended, this means that the Timer has fully stopped and as such, no timeout events can occur anymore. If you queue a notification that this condition is fulfilled, this notification can be used to denote the end of a Timer session.

As an exercise, extend the ITimer interface with a Stopping state that starts when a Running timer is cancelled. Within the Stopping state, in-events should be illegal and inevitably, an out-event cancelled should be sent.

Solution:

interface ITimer {

extern long\_integer $long$;

enum State { Idle, Running, Stopping };

in void start(long\_integer milliseconds);

in void cancel();

out void timeout();

out void cancelled();

behaviour {

State state = State.Idle;

[state.Idle] {

on start: state = State.Running;

on cancel: { }

}

[state.Running] {

on start: illegal;

on cancel: state = State.Stopping;

on inevitable: {

state = State.Idle;

timeout;

}

}

[state.Stopping] {

on start: illegal;

on cancel: illegal;

on inevitable: {

state = State.Idle;

cancelled;

}

}

}

}

With the above interface modification, a handshake protocol can be supported where the end of a Timer session is marked with a cancelled event. You may find yourself wondering why this is so useful.

**Here’s the trick:** the event that marks the end of a session travels through the same communication channel as the timeout events. The interface prohibits sending timeouts during the Stopping state and the Stopping state transitions to the Idle state where again, no timeouts can be sent. Therefore, it is guaranteed that there will be no more timeout events from the previous session after the cancelled event has been placed into the queue.

So, the modified ITimer interface complies with the handshake we want to implement to solve the ‘external’ delay problem. What remains to be done is update the RobustTimer component behaviour to make use of the newly added handshake.

The behaviour we want to add to the RobustTimer is the addition of the Stopping state. When a cancel event is sent to its provided ITimer port, this is propagated to the ‘external’ ITimer port and the RobustTimer transitions to the Stopping state. Until the ‘external’ ITimer port has replied with a cancelled event, any timeouts coming through the port while Stopping should be silently discarded. When the cancelled event is handled, the RobustTimer transitions back to the Idle state and the cancelled event is propagated to the provided ITimer.

As an exercise, add the changes listed above to the behaviour of the RobustTimer component.

Solution:

component RobustTimer {

provides ITimer iTimer;

requires external ITimer ext\_iTimer;

behaviour {

enum State { Idle, Running, Stopping };

State state = State.Idle;

on iTimer.start(ms): {

[state.Idle] {

ext\_iTimer.start(ms);

state = State.Running;

}

}

on iTimer.cancel(): {

[state.Running] {

ext\_iTimer.cancel();

state = State.Stopping;

}

[state.Idle] {  }

}

on ext\_iTimer.timeout(): {

[state.Running] {

iTimer.timeout();

state = State.Idle;

}

[state.Stopping] { /\* discard \*/ }

}

on ext\_iTimer.cancelled(): {

[state.Stopping] {

iTimer.cancelled();

state = State.Idle;

}

}

}

}

If you verify the behaviour in the above solution, you will still encounter one last verification error. While we were focussing on making sure possibly delayed timeouts are properly handled, we didn’t implement the handshake we designed for that purpose in the scenario where an Idle Timer receives a cancel event. Luckily, the fix for this is a pretty simple one:

interface ITimer {

// variable and event declarations

behaviour {

[state.Idle] {

on start: state = State.Running;

on cancel: { cancelled; }

}

// rest of the behaviour specification

component RobustTimer {

provides ITimer iTimer;

requires external ITimer ext\_iTimer;

behaviour {

// variable declarations

on iTimer.cancel(): {

[state.Running] {

ext\_iTimer.cancel();

state = State.Stopping;

}

[state.Idle] { iTimer.cancelled(); }

}

// rest of the component implementation

With these additions in place, the RobustTimer component will pass verification and you have successfully implemented a protocol that is robust against possibly delayed asynchronous events from a native component. By doing this in a Dezyne component, any other component that is connected to the ITimer port provided by RobustTimer can use it as if it were not ‘external’. Only the RobustTimer component is responsible for handling the ‘external’ behaviour.

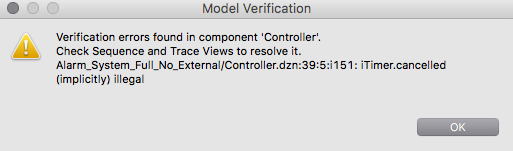
In the next and final chapter, we will have a look at what needs to be updated in the rest of the Alarm System and afterwards we will take a step back and summarise what we’ve discovered throughout this tutorial.

# Chapter 3: Finishing the Alarm System

In the previous chapter, you successfully implemented the behaviour of a component such that it still behaves correctly if communication over a port is potentially delayed. In this chapter, we will have a look at what needs to be done to make the rest of the Alarm System up to date and if any changes need to be made to native code.

## Updating the Controller

A snapshot containing Dezyne models and native code for the current state of the Alarm System can be found on <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/External/Ch3_Current_State>.

Although the Controller hasn’t been changed at all, the ITimer interface that it requires has changed. If you verify the Controller, you will find that the iTimer.cancelled event is (implicitly) illegal:

What has happened is that an out-event was added to the ITimer interface that is not handled in the Controller behaviour. Therefore, it is implicitly illegal. To avoid this, the behaviour of the Controller component should be modified such that it can handle the cancelled event from its required ITimer port.

Just like we added an extra state to the behaviour of the RobustTimer to indicate that it was in between the Running and Idle states, it makes sense to add a state to the Controller that indicates it is “waiting” for its Timer port to be fully cancelled. As the cancellation of an active Timer occurs during the transition from Alarming to Armed, the suggested state to be added is Rearming. As an exercise, add this Rearming state to the behaviour of the Controller component.

Solution:

component Controller {

  provides IController iController;

  requires ISiren iSiren;

  requires ILED iLed;

  requires ITimer iTimer;

  requires IPWManager iPWManager;

  requires ISensor iSensor;

  behaviour {

    enum State { Unarmed, Armed, Alarming, Rearming };

    State state = State.Unarmed;

    [state.Unarmed] {

      on iController.passwordEntered(pw): {

      bool valid = iPWManager.verifyPassword(pw);

      if(valid) {

        state = State.Armed;

        iLed.setYellow();

        iSensor.turnOn();

        }

      }

      on iSensor.triggered(): {}

      on iTimer.timeout(): illegal;

    }

    [state.Armed] {

      on iController.passwordEntered(pw): {

      bool valid = iPWManager.verifyPassword(pw);

      if(valid) {

        state = State.Unarmed;

        iLed.setGreen();

        iSensor.turnOff();

      }

      }

      on iSensor.triggered(): {

      state = State.Alarming;

      iTimer.start($30000$);

      iLed.setRed();

      iSensor.turnOff();

      }

      on iTimer.timeout(): illegal;

    }

    [state.Rearming] {

    }

    [state.Alarming] {

      on iController.passwordEntered(pw): {

      bool valid = iPWManager.verifyPassword(pw);

      if(valid) {

        state = State.Rearming;

        iLed.setYellow();

        iSiren.turnOff();

        iTimer.cancel();

        iSensor.turnOn();

        }

      }

      on iSensor.triggered(): {}

      on iTimer.timeout(): {

      iSiren.turnOn();

      }

    }

  }

}

Of course, simply adding a state is not enough, you will also need to handle events if the Controller is in the Rearming state. If a cancelled event comes in from the ITimer port, the Controller should transition to the Armed state. While the Controller is Rearming, incoming passwords and Sensor triggers are silently discarded. As an exercise, implement the described behaviour in the Rearming state.

Solution:

[state.Rearming] {

    on iController.passwordEntered(pw): {}

    on iSensor.triggered(): {}

    on iTimer.cancelled(): state = State.Armed;

}

With the above solution in place, the Controller component will successfully pass verification when using the new ITimer interface. The last thing that then needs to be modified is the native implementation of the Timer, so that accurately reflects the ITimer changes as well.

As a bonus exercise, you may want to consider checking the validity of a password that was entered during the Rearming state. If the password is valid, receiving the cancelled event could make the Controller transition straight to the Unarmed state.

Solution: (take note that some unchanged pieces of code are left out for the purpose of this document)

component Controller {

  /\* provides/requires interfaces \*/

  behaviour {

    enum State { Unarmed, Armed, Alarming, Rearming };

    State state = State.Unarmed;

    bool correctPasswordQueued = false;

    [state.Unarmed] {

      /\* behaviour unchanged \*/

    }

    [state.Armed] {

      /\* behaviour unchanged \*/

    }

    [state.Rearming] {

      on iController.passwordEntered(pw): correctPasswordQueued = iPWManager.verifyPassword(pw);

      on iSensor.triggered(): {}

      on iTimer.cancelled(): {

    if(correctPasswordQueued) {

      state = State.Unarmed;

      iLed.setGreen();

correctPasswordQueued = false;

    }

    else {

      state = State.Armed;

      iLed.setYellow();

  iSensor.turnOn();

}

      }

    }

    [state.Alarming] {

      on iController.passwordEntered(pw): {

      bool valid = iPWManager.verifyPassword(pw);

      if(valid) {

  state = State.Rearming;

  iTimer.cancel();

      iSiren.turnOff();

        }

      }

      on iSensor.triggered(): {}

      on iTimer.timeout(): {

      iSiren.turnOn();

      }

    }

  }

}

## Updating the native Timer

By adding the cancelled out-event to the ITimer interface, we added a function to the System boundary (the System still requires an ITimer port) that needs to be called from native code. As we are generating a thread-safe-shell for the System, it is permitted to call the event directly from native code; queueing it in the System dzn::pump is done by default.

As an exercise, add the calling of the cancelled event to the native Timer implementation (found in main.cc).

Solution:

int main(int argc, char\* argv[])

{

  dzn::locator loc;

  dzn::runtime rt;

  std::ofstream logfile("log.txt");

  std::ostream outstream(nullptr);

  outstream.rdbuf(logfile.rdbuf());

  loc.set(rt).set(outstream);

  AlarmSystem as(loc);

  as.dzn\_meta.name = "AlarmSystem";

  as.iTimer.in.start = [] (int ms) { alarm(ms/1000); };

  as.iTimer.in.cancel = [&] () { alarm(0); as.iTimer.out.cancelled(); };

  timeout = as.iTimer.out.timeout;

  signal(SIGALRM, sigalrm\_handler);

  as.check\_bindings();

  std::string input;

  while(std::cin >> input) {

    as.iController.in.passwordEntered(input);

  }

}

With this final addition in place, you will have a fully correct implementation of the ITimer interface that is verified to always correctly deal with possible race conditions that are caused by dzn::pump queue delay. If such delays occur, they are handled by the RobustTimer Dezyne component which is verifiably correct.

The final Dezyne models and Alarm System native source code can be found on GitHub: <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/External/Ch4_Final_State>

A future tutorial on the ‘blocking’ keyword will show you a way to make use of ‘external’ with even less modifications to surrounding components, if your execution model allows it. For now, let’s reflect on what we’ve learned in this tutorial in the next and final chapter.

# Chapter 4: Retrospect on learning goals

In the introduction of this tutorial, the following learning goals were mentioned:

* Understand when to use ‘external’ and what problems it solves
* Understand the impact of using the ‘external’ keyword in your Dezyne models
* Identify where to implement additional logic in your models to support the usage of ‘external’
* Use the Dezyne verification & simulation tools to correctly implement a solution using the ‘external’ keyword

In this chapter, you will find a quick summary on each of the learning goals and where they were discussed in the tutorial.

## Understand when to use ‘external’ and what problems it solves

**Mark as ‘external’ any requires interface containing out-events that is delegated to the boundary of a system.** This was discussed at length in What does ‘external’ do and when should you use it? after an analysis of the Alarm System implementation we made in earlier tutorials showed that illegal behaviour could occur.

## Understand the impact of using the ‘external’ keyword in your Dezyne models

Coincidentally, this was discussed before the answer to the previous learning goal was given. An analysis of why Dezyne did not warn you about this illegal behaviour in Why does verification not catch this? can also be used to explain the impact of using ‘external’: **an extra set of behaviours is considered during verification so that possible delays due to external communications are included.**

## Identify where to implement additional logic in your models to support the usage of ‘external’

In Chapter 2: How to use ‘external’?, two important considerations were given as to how you should implement ‘external’ in your models. Firstly, in Responsibilities and using ‘external’ components it is recommended to create a new component thatmaps an **external requires** port to a **provided** port of the same type. This is so that you don’t pollute other components with logic only concerning ‘external’ behaviour. Then, in ‘The’ solution: handshake protocol a handshake was added to the interface so that the state of the two ports can be synchronized.

## Use the Dezyne verification & simulation tools to correctly implement a solution using the ‘external’ keyword

Information regarding this learning goal was present all throughout the tutorial. However, by implementing the solution in a Dezyne component as opposed to doing it in native code, you ensure that you are making full use of the Dezyne toolkit and benefit from its verification and simulation capabilities.