Dezyne Reference Manual

Component based, formally verified.

The Dezyne developers

Edition 2.10.0
28 September 2020

Table of Contents

1 Introduction

Dezyne is a component based language, as well as a method for the development of event-driven systems. The language has formal semantics, which is coherently expressed in: a textual representation, a graphical representation, a mathematical representation, a source code representation, and the observable behaviour of a machine executing the resulting program. The concepts available in the language denote the different properties[1] that can be observed and have meaning in one or more of the reprentations: textual, graphical, mathematical, program and execution.

1.1 Purpose

The purpose of Dezyne is to eliminate the need for testing certain aspects or qualities of a software program that could instead be proven to be correct by applying formal verification methods.

1.2 Principles

Dezyne aims to support known software engineering principles:

- clarity and ease of discovery
  - visual overview of architecture (layers of decomposition)
  - visual overiew of behaviour (event trace scenarios)
- rigor and formality
  - Dezyne allows enforcing completeness and correctness
  - implementations consistent with specifications
  - specification consistency
- separation of concerns (divide and conquer, low coupling – high cohesion)
  - data is separated from control
  - components allow for functional decomposition
  - interfaces specify the conversations between each peer in a point to point relation
- composition is separated from behaviour
- modularity (low coupling, high cohesion; dependency inversion)
  - at the file level, Dezyne allows separating type, interface, component and system definitions
  - a component only depends on its interfaces
  - a system only depends on it components
  - a system is a component
- abstraction (Liskov substitution)
  - data independence
  - an event represents the transition from one (system) state to another

[1] Structural: events and their direction in an interface, ports on a component, components in a system, bindings between the ports; behavioural: guarded trigger => action | assignment | if-else | function

    - its name and place in the system structure strongly indicates its meaning
    - an interface is the complete abstraction of an implementation from the perspective of any client
    - and interfaces completely isolate the implementation from its environment
- anticipation of change (open closed)
    - modularity and composition allows changing implementations whithout changing interfaces, dependencies and dependents
    - a system in Dezyne is inherently protected from changes propagating[2]
- generality
    - Dezyne allows trivially substituting components offering a port of a compatible interface type
    - Dezyne's data independence[3] allows components to be more general
- incremental development
    - increments in Dezyne can obviously follow component scope
    - Dezyne supports use case driven development or cross-cutting features equally well
- testability
    - trace generation allows for 100% test coverage of code paths

1.3 Typical use of Dezyne

- Build working software systems

  A software system built with Dezyne will consist of stateful Dezyne components and non-stateful, functional code also called algorithms or lambda functions.
- Formally specify communication protocols
- Formally specify hardware constraints/behaviour

If you are new to Dezyne and wonder how or where it could work for you, consider the following table. It shows where and how Dezyne is more likely to be used

better fit		**less likely fit**
stateful systems	over	functional systems
mixed state/functional software	over	database software
machine control	over	compilers
event-driven	over	signal/clock-driven
large systems (> 50.000LOC)	over	small systems (< 10.000)
device drivers	over	spread sheets
failure => $$$	over	failure => refresh (F5)
distributed systems	over	single-SOC systems
concurrent systems	over	single threaded
mature systems	over	throwaway prototypes

[2] Interfaces are like a change firewall, they stop a change propagating across modules.

[3] Data types in Dezyne are similar to typedefs in C and allow a component to be repurposed without changes to the model itself.

ASP backend	over	REST server
Graphical UI	over	command-line UI
asynchronous programming	over	blocking calls

1.4 The Dezyne Application Domain

Metaphorically speaking, when software is like an orchestra, sound is produced by the instruments, but it requires the intricate coordination of each individual musician, the ensemble itself and possibly the conductor to make the music. Dezyne is aimed at producing such software.

Both the conductor and the musician are examples of entities coordinating concurrent activities with external actors. The musician must coordinate with the conductor, their fellow musicians, and the instrument, which in actuality might require the use of both hands, the lungs, etc. The conductor monitors the music, monitors the score, sets the tempo, coordinates the entries of musicians and shapes the phrasing (see `https://en.wikipedia.org/wiki/Conducting`).

Dezyne is purposfully designed for software systems where by nature it is highly desirable to build up explicit contextual awareness to drive program execution, i.e. when the software system to be built rises above a certain complexity level where simpler approaches are no longer adequate and abstraction is paramount. The table below is intended as a guideline, not a rule. Dezyne can be applied at every level, the converse is not necessarily true.

type	**level**	**description**	**example**
dezyne[4]	sophisticated	distributed and coupled/supervisory	machine controller, autonomous robot
bus or black board	intermediate	independent and separated	automobile ECUs, automated tunnel/bridge
"god" controller	simple	monolithic	elevator PLC, motion controller

Encapsulation, i.e. hiding implementation (interaction) details, leads to protocol and state abstraction.

In its current state, Dezyne is capable of describing a system with a static structure of components which exchanges events with different external actors in order to coordinate towards achieving an overarching goal.

Dezyne allows carefully composing the controlling structure together with appropriate data transformation, whilst maintaining the ability to reason about each facet separately.

[4] Hierarchy (tree/network) of components

2 Installation

In order to install Dezyne on your system, you can use a binary installation that we prepared especially for you. In case you are interested in building Dezyne from source yourself, watch this space!

2.1 Binary Installation

This section describes how to install Dezyne on an arbitrary system from a self-contained tarball providing binaries for Dezyne and for all its dependencies. This is often quicker than installing from source, which is described in the next sections. The only requirement is to have GNU tar and gzip.

2.1.1 Generic GNU/Linux Binary

Installing goes along these lines:

1. Download the binary tarball from ‘`https://dezyne.org/download/dezyne-2.10.0-x86_64-linux.tar.`
   e.g.:

   ```
   $ wget https://dezyne.org/download/dezyne-2.10.0-x86_64-linux.tar.gz
   ```

   Make sure to download the associated `.sig` file and to verify the authenticity of the tarball against it, do something like:

   ```
   $ wget https://dezyne.org/download/dezyne-2.10.0-x86_64-linux.tar.gz.sig
   $ gpg --verify dezyne-2.10.0-x86_64-linux.tar.gz.sig
   ```

   If that command fails because you do not have the required public key, then run this command to import it:

   ```
   $ wget https://savannah.gnu.org/people/viewgpg.php?user_id=4348 \
         -qO - | gpg --import -
   ```

   and rerun the `gpg --verify` command.

   Take note that a warning like “This key is not certified with a trusted signature!” is normal.

   Now, you can unpack the tarball; do something like:

   ```
   $ tar --warning=no-timestamp -xf /path/to/dezyne-2.10.0-x86-64.tar.gz
   ```

   Then try:

   ```
   $ cd dezyne-2.10.0
   $ ./dzn --help
   ```

2. Make the `dzn` command available from other locations or to other users on the machine, for instance with:

   ```
   $ ln -s $PWD/dzn ~/bin/dzn
   or
   # ln -s $PWD/dzn /usr/local/bin/dzn
   ```

3. And optionally, make the `dzn-env` prefix-command[1] available:

   ```
   $ ln -s $PWD/dzn-env ~/bin/dzn-env
   ```

[1] `dzn-env` can be used as a prefix for using programs from your operating system, such as `info`, `man`, or `emacs`; so that they may find and use the documentation and extensions that are provided in the binary release.

```
or
# ln -s $PWD/dzn-env /usr/local/bin/dzn-env
```

4. Test your installation

```
$ dzn hello
$ dzn-env info dezyne
```

and get busy Dezyne'ing, see Chapter 7 [The Dezyne command-line tools], page 82! You can skip the following sections about building from source.

2.1.2 Generic Microsoft Windows Binary

Installing goes along these lines:

1. Download the binary zip archive from '`https://dezyne.org/download/dezyne-2.10.0-i686-windows.z`

...

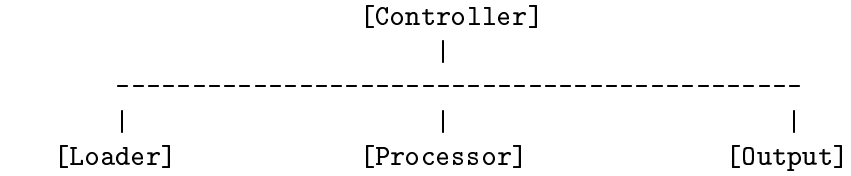
and get busy Dezyne'ing, see Chapter 7 [The Dezyne command-line tools], page 82!

3 The Dezyne Modeling Language

When a software system is built, it is good practice to split it up (decompose) in parts that have high cohesion and low coupling. High cohesion means that the system part performs a dedicated task or responsibility. Low coupling means that the system parts are able to perform their responsibility without much interaction with other parts, system parts thus have simple interfaces to interact with each other.

3.1 System Decomposition

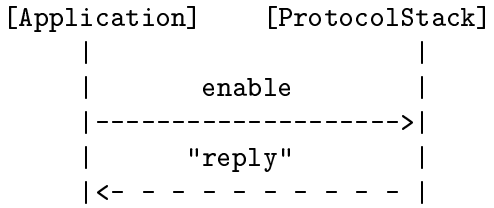
As an illustration, below a picture of a system with four parts where the loader loads raw input into a machine, the processor transforms the raw input into the final product and the output part moves the product out of the machine and the complete process is controlled by the controller.



Software development using Dezyne is component based, meaning that the system parts are **components**

- A component implements a certain functionality.
- A component implementation may use the functionality offered by other components.
- The functionality that is offered by a component to others is defined in the components **interface**.
- The functionality of a component can be used by sending and receiving **events** to and from the component over the **interface**, which is similar to calling one or more methods on the component interface.

To illustrate this, in the interaction diagram below the component called "Application" requests the component "ProtocolStack" to activate itself, by sending the enable event to the "ProtocolStack" component. The ProtocolStack responds synchronously with a "reply", either a void reply or a value returned as result of the enable.



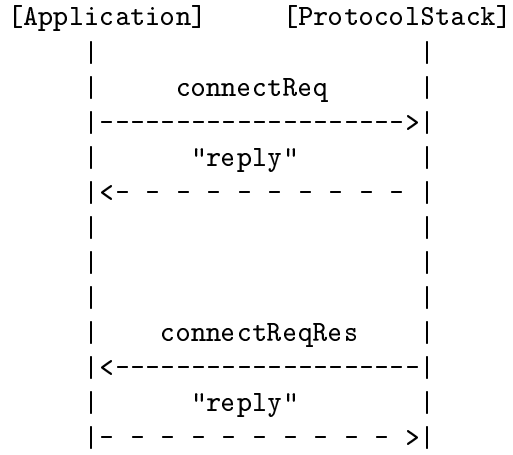
Events have a direction. The direction is determined as seen from the component providing the service. For the Server component in this example the enable event is an **in** event.

3.1.1 Synchronous and asynchronous communication

Between components there are two fundamental type of interactions: 1) synchronous and 2) asynchronous.

The example above depicts a synchronous action, after sending the event enable to the ProtocolStack, the Application execution will be blocked until the "reply" (either a void reply or a value returned as result) is received.

When the ProtocolStack is activated, it can be requested to setup a connection by sending the event "connectReq" to it. After giving a "reply" on this event the ProtocolStack component will start setting up a connection and the two components continue to run concurrently, i.e., are able to perform other tasks while the connection request is being processed by the ProtocolStack component. Once the request is completed the ProtocolStack will send a notification connectReqRes to the Application. This illustrates an asynchronous action.



Remember that events have a direction. For the ProtocolStack component in this example the connectReqRes event is an **out** event. The Application, however, receives this event as an in event that has a void return or some other return type (currently Dezyne only supports void out events).

3.1.2 Interfaces as abstraction of a component's behaviour

In the Dezyne Modelling language, an **interface** contains the definition of the events, their direction and optionally their parameters, and a specification of the externally visible behaviour of the component that will provide an implementation of the interface.

With behaviour we mean which events (or method calls) can be received and can be sent at which stages in the execution process, or in other words the protocol a user of the interface should obey. In the case of the Application and ProtocolStack the interface defines that the ProtocolStack should first be enabled before anything else can be done with it, that a connectReq must be followed by a connectReqRes and that it is not allowed to send a second connectReq before a connectReqRes has been received.

As described above, components use or provide functionality via interfaces and not by interacting directly with other components. Therefore, interface specifications can not contain component implementation details.

An interface specification that defines part of the ProtocolStack interface (the enabling and disabling) in the example above could look like:

```
interface IProtocolStack
{
```

```
    enum Result                                  // enumeration type used for the reply val
    {                                            // the event enable
      Ok,
      Failed
    };
    in Result enable();                          // in event to enable the ProtocolStack
                                                 //   the event will return if enabling was
    in void disable();                           // in event to disable the ProtocolStack
    behaviour                                    // the specification of the externally vis
    {
        enum State {OFF, ON};
        State state = State.OFF;
        [state.OFF]
        {
            on enable:                           // One possible response to enable is Ok
            {
               reply(Result.Ok);
               state = State.ON;
            }
            on enable: reply(Result.Failed);     // The other possible response to enable i
            on disable: illegal;                 // Disabling the Protocol Stack is not all
        }
        [state.ON]
        {
            on enable: illegal;                  // Enabling the Protocol Stack is not allo
            on disable: state = State.OFF;
        }
    }
}
```

In the interface specification three things stand-out: 1) actions for all possible events have to be specified, 2) events can be marked as 'illegal' and 3) all possible replies are specified.

1) In the interface specification above a state machine has been introduced to define what is allowed at each stage: the protocol specified by this state machine must be obeyed. In order to have an unambiguously defined interface in each state, the actions for all possible events have to be specified. That is why you see for each state the "on enable" and "on disable" statement. A Dezyne model is complete when it specifies for each possible state and each possible incoming event what happens in terms of actions when this event is received while the interface is in this particular state. Dezyne will check that the specification is complete and thereby ensure that there will be no undefined actions for all possible events. In Dezyne this is referred to as a **completeness check**.

2) For completely specifying the behaviour of an interface, one must specify also all actions that should not happen or are not allowed at certain stages, i.e., will not be serviced when the component implementing the interface is in a particular state. This is done by marking actions that are not allowed as **illegal**. The Dezyne verification will verify that all components using the interface will adhere to it and in case a component is sending an

event that is illegal at that stage Dezyne will point this out as a interface incompliancy. Note that marking behaviour as illegal is a requirement or design decision, in the example above the ProtocolStack could also be such that it is allowed to send a disable when the component is not yet enabled.

3) To unambiguously define all possible behaviour, an interface must specify all replies that are possible. Definition of all possible replies is described in way that is abstract from implementation details. What you see specified in the OFF state when the enable event is received is an example of how this is done in the Dezyne Modeling language. What is described here is that as a result of the enable event two things can happen: a reply plus a state transition in case enabling is successful and a reply when enabling fails. In Dezyne this is referred to as a **non deterministic choice**. With this definition it is clear that for a user of this interface that it should be able to deal with both replies at this stage.

In the interface above, there is no definition yet the connectionReq and especially how the connectReqRes will be returned in case processing of the connection request is ready. How to specify this is outlined next.

3.1.3 Modelling events not visible to the outside world but leading to external visible behaviour

The following is crucial for understanding how to develop models in Dezyne.

As mentioned, components use or provide functionality via interfaces and not by interacting directly with other components. Therefore, interface specifications can not contain component implementation details.

However, when this leads to externally visible behaviour, interfaces necessarily capture decisions that are internal to the implementation. As an example consider in the case above the completion of the connection request that will trigger the server component to return a notification connectReqRes. But how is that captured in an interface specification that should not contain implementation details?

There are two options to specify internal events (like the completion of the request initiated by connectReq), that lead to external events in interfaces:

- by using **on inevitable**; meaning that when nothing else happens and the system is not triggered externally, the event will eventually always occur. For example a timeout event of a timer will eventually always take place if the timer is not cancelled by the client.
- by using **on optional**; meaning that it may or may not occur. For example, an error message may not be generated when the system is working properly.

With this knowledge the interface specification for the ProtocolStack in the example we have been using so-far could look like:

```
interface IProtocolStack
{
  enum Result                                  // enumeration type used for the reply val
  {                                            // the event enable
    Ok,
    Failed
  };
  in Result enable();                          // in event to enable the ProtocolStack
```

```
                                                //   the event will return if enabling was
    in void disable();                          // in event to disable the ProtocolStack
    in void connectReq();
    out void connectReqRes();
    behaviour                                   // the specification of the externally vis
    {
        enum State {OFF, ON, CONNECTING, CONNECTED};
        State state = State.OFF;
        [state.OFF]
        {
            on enable:                          // One possible response to enable is Ok
            {
               reply(Result.Ok);
               state = State.ON;
            }
            on enable: reply(Result.Failed);  // The other possible response to enable i
            on disable,connectReq: illegal;
        }
        [state.ON]
        {
            on enable: illegal;
            on disable: state = State.OFF;
            on connectReq: state = State.CONNECTING;
        }
        [state.CONNECTING]
        {
            on enable, connectReq: illegal;
            on disable: state = State.OFF;
            on inevitable:                  // inevitably the connectReq request will b
            {
                state = State.CONNECTED;
                connectReqRes;
            }
        }
        [state.CONNECTED]
        {
            on enable, connectReq: illegal;
            on disable: state = State.OFF;
        }
    }
}
```

3.1.4 Dezyne modelling language versus programming languages

The Dezyne modelling language syntax closely resembles C, C++ and Java, making it easy to work with in terms of data types, scope, and so forth. But you must keep in mind always that the Dezyne Modelling language is truly a modelling language, not a language that

directly compiles at the same level of abstraction as C/C++/Java, even though it looks and feels like these familiar languages. Instead, verified Dezyne models ultimately generate real C/C++/Java source code, which in turn goes through standard C/C++/Java compilers or interpreters to become actual machine-executable code.

Models are compiled by Dezyne into formal language which is then model checked (formally verified) by checking every possible path through the communication and state structures of the modelled system, seeking problems such as interface incompliancies, deadlocks, live-locks, race conditions and unhandled events. Each path has steps like "component A presents information x via interface I to component B, and component B responds through interface I, perhaps now or perhaps later (or perhaps never), with information y intended for A, then A presents y or some transformation of y via interface J to component C, . . . " Sequences matter, but timing does not matter except for the problem case of "never returns". Dezyne will either find no problems, or it will find one and tell you, the model developer, the exact sequence of communications and actions that leads to the problem.

From this viewpoint, Dezyne and its modelling language are both a specification tool and an exploration tool. You do not need to concern yourself up front about, say, infinite loops or deadlocks; just design what seem to be correct interfaces and component states, and then verify the model. Dezyne will tell you if your conceptual model has a flaw. You might be surprised by what Dezyne tells you. In fact, the power of Dezyne is precisely here: Dezyne uncovers complex, subtle, hard-to-find bugs in your communications and behaviour logic – before you write or generate even one line of C, C++ or Java!

3.1.5 How to use the language help topics

The language help topics are setup in such a way that they provide support on how to use the Dezyne Modelling Language (DML) constructs.

For each topic there is a general intent, followed by the syntax and examples.

The topics can be read in any order, however, there is a certain sequence that can guide you in setting up your first DML models:

First it is outlined how an interface and events should be constructed, next how a component is constructed and how to specify that a component provides or requires an interface. Then you can find how to define variables and how to specify interface and component behaviour with all the possible constructs (functions, conditionals, etc). Finally it is shown how you can decompose components into others / group components into systems.

3.2 Creating an Interface

Interfaces define how components interact: the events that can be communicated and the interaction protocol.

Each event has a direction specified by the 'in' or 'out' keywords.

The interface protocol is specified in a 'behaviour' section. For each in event it specifies the actions performed.

An interface does not lead to any code being generated. It serves as a specification only.

3.2.1 Interface Syntax

The 'interface' keyword is used to declare an interface, followed by the interface name.

Within the interface declaration, a list of in and out events is defined followed by a behaviour declaration.

```
interface IntName
{
   in type inEvent1();   // list of in events
   in type inEvent2();
   /* ...  */
   out void outEvent1(); // list of out events
   out void outEvent2();
   /* ...  */
   behaviour             // contract; protocol description
   { /* ...  */ }
}
```

3.2.2 Interface Examples

3.2.2.1 An interface with three events and a simple behaviour

The following interface has three events defined: two in events and one out event. The direction is as seen from the component implementing this interface.

```
interface IMotion
{
  in void start();
  in void stop();
  out void ready();
  behaviour
  {
    on start, stop: {}
  }
}
```

3.2.2.2 An interface specifying an interaction protocol

This example shows an interface with a more complex interaction protocol, where the start event is only allowed in case the interface is in state Off and the stop event is only allowed in case the interface is in state Active.

```
interface IMotion
{
  in void start();
  in void stop();
  out void started();
  out void ready();
  behaviour
  {
      enum State {Off, Active};
      State state = State.Off;
      [state.Off]
      {
```

```
            on start:
            {
                state = State.Active;
                started;
            }
            on stop: illegal;
        }
        [state.Active]
        {
            on start: illegal;
            on stop:
            {
                state = State.Off;
                ready;
            }
        }
    }
}
```

See also:

- Section 3.2.3 [Specifying Events], page 13,
- Section 3.6 [Specifying Behaviour], page 28,

3.2.3 Specifying Events

Component interaction is performed via so-called *events* through instances of interfaces (that are called ports).

In each interface therefore the events that can be received (in events) and sent (out events) are specified.

3.2.3.1 Event Syntax

An event is declared by its direction (**in** or **out**), type and name.

The direction indicates if the event is received by (in) or sent (out) from a component that will be implementing the interface

Out events are only allowed to have **void** type.

```
in return-type event-name();      // an in event declaration
out void event-name();            // an out event declaration
```

Events can also have a parameter list to pass data variables, see Section 3.5 [Using Data Variables and Parameters], page 25,.

```
[in/out] return-type event-name(parameter-list)
```

The parameter list is a set of the following: [in/out/inout] indicating whether the datatype is input for or output from the event, the parameter type and parameter name. See also Section 3.5 [Using Data Variables and Parameters], page 25,.

3.2.3.2 Event Examples

3.2.3.3 An interface with three void events (and a simple behaviour)

The following interface has three events defined: two in events and one out event.

The direction is as seen from the component implementing this interface.

```
interface IMotion
{
  in void start();
  in void stop();
  out void ready();
  behaviour
  {
    on start, stop: {}
  }
}
```

3.2.3.4 An interface with an enum type event

This interface has three events defined: two incoming events and one outgoing event. One of the incoming events is of an enum type, that is defined in the first line of the interface declaration.

```
interface iSimpleProtocol
{
  enum Result {Ok, Failed, Error};
  in Result connect();
  in Result disconnect();
  behaviour
  {
    on connect: reply(Result.Ok);    // Action = set return value to 'Result.Ok'
    on disconnect: reply(Result.Ok); // Action = set return value to 'Result.Ok'
  }
}
```

See also:

- Section 3.2 [Creating an Interface], page 11,
- Section 3.5 [Using Data Variables and Parameters], page 25,

3.3 Creating a Component

Three types of component can be defined in Dezyne:

- The first type is a component that has only a name defined and has no implementation. In Dezyne this is a placeholder for a component that is implemented in another programming language.
- The second type is a component that has an implementation marked by the 'behaviour' keyword.
- The third type is a component decomposed into others that has an implementation marked by the 'system' keyword. The system specifies contained component instances, see further Section 3.7 [Decomposing a Component], page 50,.

3.3.1 Component Syntax

A component declaration is named, and introduces a list of provides and requires interfaces and optionally behaviour or a system.

```
1) component CompName
   // placeholder for component implemented in another programming language
   {
     provides CompInterface compinterface; // list of provided interfaces
     requires UsedInterface usedinterface; // list of required interfaces
    }
2) component CompName
   {
   // component with behaviour
     provides CompInterface compinterface; // list of provided interfaces
     requires UsedInterface usedinterface; // list of required interfaces
     behaviour   // behaviour
     { ...  }
   }
3) component CompName
   {
   // system component
     provides CompInterface compinterface; // list of provided interfaces
     requires UsedInterface usedinterface; // list of required interfaces
     system   // system
     { ...  }
   }
```

3.3.1.1 Examples

3.3.1.2 A component that will be implemented in another programming language

This component has no implementation. It represents a component implemented in another programming language.

```
component CompName
{
  provides CompInterface compinterface;
}
```

3.3.1.3 A Component Implementing One Interface and a Simple Behaviour

The following component implements one interface and a straightforward behaviour section.

```
component Leaf
{
  provides ISensor ctrl;
    behaviour
    {
```

```
            on ctrl.event(): { }
        }
    }
```

3.3.1.4 A Component Decomposed Into Two Components

A component decomposed into two components where these components are connected via their ports.

```
component Composition
{
  provides CompInterface compinterface;
  system
  {
    Comp1 comp1instance;
    Comp2 comp2instance;
    comp1instance.port1 <=> comp2instance.port2;
  }
}
```

See also:

- Section 3.6 [Specifying Behaviour], page 28,
- Section 3.7 [Decomposing a Component], page 50,
- Section 3.3.3 [Ports], page 17,

3.3.2 Importing Models

When a component refers to other components or interfaces and their details, the declaration of these has to be known.

An import clause is available to load the declaration from separate files holding interface or component declarations.

3.3.2.1 Syntax

The keyword **import** is used to import interface or component definition files.

```
import ModelName.dzn;
```

Each import clause will look on the file system for a file with the required name, and include the content.

For the time being the file should be located in the same directory as the current model. So 'import Console.dzn' expects a file './Console.dzn'

3.3.2.2 Examples

3.3.2.3 A component importing an interface

The component Leaf implements the ISensor interface for which the declaration is imported using the import statement.

```
import ISensor.dzn
component Leaf
```

```
{
  provides ISensor sensor;
    behaviour
    {
        on sensor.event(): { }
    }
}
```

See also:

- Section 3.3 [Creating a Component], page 14,

3.3.3 Ports

A port is an instance of an interface. A component has ports to communicate with other components.

The keyword **provides** is used to specify that a component provides an implementation of an interface.

The keyword **requires** is used to specify that a component uses or requires an interface.

A component must provide an implementation for all in-events of provided and all out events of required interfaces.

3.3.3.1 Syntax

```
provides Interface port;

requires Interface port;

requires external Interface port; (1)(Since 2.0.0)

requires injected Interface port; (2)
```

1) port to a component with possibly delaying communication channel (see Section 3.3.4 [External], page 19)

2) port to a shared resource (see Section 3.3.5 [Injected], page 20)

3.3.3.2 Examples

3.3.3.3 A component implementing one interface

The component Leaf implements the ISensor interface that has one in-event called event.

```
component Leaf
{
  provides ISensor sensor;
    behaviour
    {
        on sensor.event(): { }
    }
}
```

3.3.3.4 A component implementing one interface and requiring another interface

The component MessageHandler below is providing an implementation of the iMessageHandler interface and requires the iProtocolStack interface.

```
component MessageHandler
// MessageHandler component
// implementing the iMessageHandler interface
// and using the iProtocolStack interface
{
    provides iMessageHandler mh;
    requires iProtocolStack ps;
    behaviour
    {
    enum State {Idle, Busy};
    State state = State.Idle;
    [state.Idle]
    // initial or idle state
      {
      on mh.sendMessage():
        {
          ps.sendMessage(); //forward the message to ProtocolStack
          state = State.Busy;
        }
      on ps.messageSent(): illegal;
      }
    [state.Busy]
    // State to capture that a message is being send
      {
      on mh.sendMessage():
        {
          illegal;
        }
      on ps.messageSent():
      // the ProtocolStack has send the message, now we need to inform
      // our client and go to the Idle state so another message can
      // be processed
        {
          mh.messageSent();
          state = State.Idle;
        }
      }
    }
}
```

See also:

- Section 3.3 [Creating a Component], page 14,
- Section 3.3.4 [External], page 19,

3.3.4 External

(since 2.0.0) The 'external' keyword specifies that communication over a port may be delayed. (The delay may be caused by inter-process communication.) Model verification will consider this delay for out events on an external port. The delay may cause race conditions leading to illegal behaviour or interface compliance errors.

Note: 'external' delay is taken into account for required port out events. Marking a provided port as external has no effect on verification.

3.3.4.1 Syntax

```
requires external Interface port; (1)(Since 2.0.0)
provides external Interface port; (2)(Since 2.0.0)
```

Explanation:

1, 2) Communication over 'port' is possibly delayed.

3.3.4.2 Examples

3.3.4.3 Race condition due to external delay

Component RemoteTimerProxy illustrates how a delayed communication channel may cause a race condition leading to illegal behaviour.

The implementation of component RemoteTimerProxy is correct for 'requires ITimer rp' but incorrect for 'requires external ITimer rp' due to race between pp.cancel and rp.timeout.

```
extern double $double$;
interface ITimer {
  in void create(double seconds);
  in void cancel();
  out void timeout();
  behaviour {
    bool is_armed = false;
    [!is_armed] on create: is_armed = true;
    [is_armed] on create: illegal;
    on cancel: is_armed = false;
    [is_armed] on inevitable: {timeout; is_armed = false;}
  }
}
component RemoteTimerProxy {
  provides ITimer pp;
  requires external ITimer rp;
  behaviour
  {
    bool is_armed = false;
    on pp.create(s): {
      [!is_armed] {rp.create(s); is_armed = true;}
      [is_armed] illegal;
    }
    on pp.cancel(): {rp.cancel(); is_armed = false;}
```

```
      on rp.timeout(): {
        [is_armed] {pp.timeout(); is_armed = false;}
        [!is_armed] illegal;
      }
    }
  }
```

See also:

- Section 3.6.8 [Reply], page 40,
- Section 3.6.3 [Actions], page 30,

3.3.5 Injected

A single provided port can be bound to multiple requires ports. The 'injected' keyword must be used to specify that a port can be bound to multiple times. This section explains

- Constraints imposed on interface protocol to allow injection.
- How to specify a system in which a single component is bound multiple times.

Regular Dezyne port bindings are peer-to-peer.

TODO: Explain meaning, implications and justification.
Multiple requires ports can be bound to a single provided port. TODO: Explain constraints

3.3.6 Behaviour

A behaviour section is used to define the actions an interface or component will perform based on the events received.

3.3.6.1 Interface Versus Component Behaviour

Behaviour in an interface is a specification. It is an agreement between components providing and requiring this interface about the messages and protocol exchanged. Both components must adhere to this specification. Interface conformance is an imprtant check in the verifier. The code generator does not generate any code based on the behaviour specification of an interface.

Behaviour in a component is a definition. It defines the actual behaviour expressed by the object modeled. Whereas an interface might still allow some freedom, sometimes multiple messages are allowed, inside a component the behaviour must be fully deterministic. The code generator uses this component behaviour specification to generate target code.

3.3.6.2 Syntax

The 'behaviour' keyword is used to declare a behaviour section. Optionally it may have a name.

```
behaviour [OptionalName]
{
  behaviour-body
}
```

The behaviour body contains a collection of statements that define the actions that will be performed based on events received.

The following elements can be used in the behaviour body:

- Local variables: see Section 3.4 [Using Local Variables], page 22,
- Functions: see Section 3.6.7 [Using functions], page 38,
- see Section 3.6.3 [Actions], page 30,
- State declarations and initialisations: see Section 3.6.4 [Specifying Stateful Behaviour], page 33,
- see Section 3.6.8 [Reply], page 40,
- Conditions: see Section 3.6.6 [Using Conditional Statements], page 37,
- Guards: see Section 3.6.5 [Using Guards], page 36,

3.3.6.3 Examples

3.3.6.4 A basic behaviour section

This example shows an interface with a basic behaviour section.

```
interface IMotion
{
  in void start();
  in void stop();
  behaviour
  {
      bool started = false;
      on start: started = true;
      on stop: started = false;
  }
}
```

3.3.6.5 A behaviour section with a protocol definition

This example shows an interface with a more complex behaviour section, where the start event is only allowed in case the interface is in state Off and the stop event is only allowed in case the interface is in state Active.

```
interface IMotion
{
  in void start();
  in void stop();
  behaviour
  {
      enum State {Off, Active};
      State state = State.Off;
      [state.Off]
      {
          on start: state = State.Active;
          on stop:  illegal;
      }
      [state.Active]
      {
```

```
            on start: illegal;
            on stop:  state = State.Off;
        }
    }
}
```

3.3.6.6 A behaviour section with a protocol definition, alternative form

This example shows an interface with the same behaviour as the one above, but an alternative form is used to define the behaviour.

In the previous example based on the state value (using guarded statements) the actions for the events per state were defined. In the alternative below based on the events the actions in each state are defined.

```
interface IMotion
{
  in void start();
  in void stop();
  behaviour
  {
      enum State {Off, Active};
      State state = State.Off;
      on start:
      {
          [state.Off] state = State.Active;
          [state.Active] illegal;
      }
      on stop:
      {
          [state.Off] illegal;
          [state.Active]  state = State.Off;
      }
    }
}
```

See also:

- Section 3.4 [Using Local Variables], page 22,
- Section 3.6.7 [Using functions], page 38,
- Section 3.6.3 [Actions], page 30,
- Section 3.6.4 [Specifying Stateful Behaviour], page 33,
- Section 3.6.8 [Reply], page 40,
- Section 3.6.6 [Using Conditional Statements], page 37,
- Section 3.6.5 [Using Guards], page 36,

3.4 Using Local Variables

Variables can be defined and used.

3.4.1 Variable Syntax

Variables are named, have a type, and an initial value, which is an expression:

```
variableType variableName = initialValue;
```

Supported types are:

- **enum**: enumerated type.
- **bool**: boolean type (true, false)
- **subint**: integer type.

To ensure that an integer variable falls within a finite set (in order to have a limited range of possible paths / states in the system), use a type definition, e.g. subint myInt {0..10}.

3.4.2 Variable Examples

3.4.2.1 Simple state machine where the state is captured in a variable of enum type

This example shows a simple state machine for a siren where the state is captured in an enumeration.

```
interface Siren
{
  in void turnOn();
  in void turnOff();
  behaviour
  {
    enum State { SirenOff, SirenOn }; // type declaration
    State state = State.SirenOff;     // variable declaration
    [state.SirenOff] // if the Siren is off, only allow to turn it on
    {
      on turnOn:  state = State.SirenOn;
      on turnOff: illegal;
    }
    [state.SirenOn] // if the Siren is on, only allow to turn it off
    {
      on turnOff: state = State.SirenOff;
      on turnOn:  illegal;
    }
  }
}
```

3.4.2.2 Simple state machine where the state is captured in a boolean type

This example shows a simple state machine for a siren where the state is captured in a variable of boolean type

```
interface Siren
{
```

```
    in void turnOn();
    in void turnOff();
    behaviour
    {
      bool SirenOn = false;
      [SirenOn] // if the Siren is on, only allow to turn it off
      {
        on turnOff: SirenOn = false;
        on turnOn:  illegal;
      }
      [otherwise]
      {
        on turnOn:  SirenOn = true;
        on turnOff: illegal;
      }
    }
}
```

3.4.2.3 Limited retrying activation of a device using a variable of integer type

This example shows how a counter based on an integer type can be used to limit retrying activation of a device.

```
component AlarmSystemComp
{
  provides AlarmSystem alarmSystem;
  requires Sensor sensor;
  requires Siren siren;
  behaviour
  {
    enum State
    {
      NotActivated,
      Activated_Idle,
      Activated_AlarmMode,
      Deactivating
    };
    subint myInt {0..10};                 // myInt is defined as integer within the ra
    myInt counter = 0;                    // myInt is initialised with value 0.
    State state = State.NotActivated;
    void activating(Sensor.Values value)  // recursive function to activate the sensor
    {
      if (value == Sensor.Values.OK)
      {
        reply(AlarmSystem.Values.Ok);
        state = State.Activated_Idle;
      }
```

```
        else
        {
          if (counter <10)
            {
            counter = counter + 1;
            Sensor.Values val = Sensor.Activate();
            activating(val);                 // try again
            }
          else
            {
            reply(AlarmSystem.Values.Failed);
            }
         }
      }
      [state.NotActivated]
      {
        on alarmSystem.SwitchOn():
        {
          Sensor.Values val = Sensor.Activate();
          activating(val);                    // call the activation function
        }
        on alarmSystem.SwitchOff(), sensor.DetectedMovement(), sensor.Deactivated():
        {
          illegal;
        }
      }
      /* ...  */
      /* ...  */
    }
  }
```

See also:

- Section 3.6.3 [Actions], page 30,
- Section 3.6.7 [Using functions], page 38,
- Section 8.3 [Types and Expressions], page 85,

3.5 Using Data Variables and Parameters

Data variables are used as input for or output from external components.

Apart from assignment, data operation / manipulation is done in handwritten code.

3.5.1 Data Syntax

Use the keyword extern to define a datatype in Dezyne and the external datatype it will be transformed to when code is generated:

```
extern datatypeInternal $datatypeExternal$;
```

After the definition variables of this type can be used to further define event (or method) signatures.

See also:

- Section 3.2.3 [Specifying Events], page 13,

3.5.3 Data Examples

3.5.3.1 Using an character type in an interface definition

This example shows an interface definition for an handwritten component that will evaluate input based on a character type.

Note that in interface definitions there is no data handling at all and therefore no parentheses and parameters are needed in e.g. on event statements.

```
interface IAlarmConsole
{
  extern char $char$;
  in void KeyPressed(char key);
  in void Initialize();
  behaviour
  {
    on KeyPressed: {}
    on Initialize: {}
  }
}
```

3.5.3.2 Using character and integer type in an interface definition

This example shows an interface definition for a handwritten component that will accept characters for a PIN code and will also return the complete code.

```
interface ICodeBuilder
{
  extern char $char$;
  extern xint $int$;          // as int is a Dezyne keyword, xint is used
  enum Result
  {
    CharAdded,
    InvalidChar,
    CodeComplete,
    BufferCleared
  };
  in Result AddDigit(in char digit);
  in void GetCode(out xint code);
  behaviour
  {
    enum State {Building, Complete};
    State state = State.Building;
    [state.Building]
```

```
    {
      on AddDigit: reply(Result.CharAdded);
      on AddDigit: reply(Result.InvalidChar);
      on AddDigit: {reply(Result.CodeComplete); state = State.Complete;}
      on AddDigit: reply(Result.BufferCleared);
      on GetCode: {}
    }
    [state.Complete]
    {
      on AddDigit: illegal;
      on GetCode: state = State.Building;
    }
  }
}
```

3.5.3.3 Passing data parameters between components

This example shows how to pass parameters between components, using the interfaces of the examples above.

```
component AlarmConsole
{
  provides IAlarmConsole alarmConsole;
  requires ICodeBuilder codestore;
  requires IAlarmSystem alarmSystem;
  /* ...  */
  /* ...  */
  behaviour
  {
    /* ...  */
    /* ...  */
    [state.Unarmed]
    {
      /* ...  */
      /* ...  */
      on alarmConsole.KeyPressed(key):                          // parameter key contai
      {
        ICodeBuilder.Result val = codestore.AddDigit(key);      // add the value to the
        if (val == ICodeBuilder.Result.CodeComplete)            // if the pincode is co
        {
          xint pin;
          codestore.GetCode(pin);                               // get the value of the
          IAlarmSystem.Result val = alarmSystem.SwitchOn(pin); // turn on the alarm sy
          /* ...  */
          /* ...  */
        }
      }
    /* ...  */
```

```
      /* ...  */
      }
    /* ...  */
    /* ...  */
    }
}
```

See also:

- Section 3.6.3 [Actions], page 30,
- Section 3.6.7 [Using functions], page 38,

3.6 Specifying Behaviour

A behaviour section is used to define the actions an interface or component will perform based on the events received.

3.6.1 Interface Versus Component Behaviour

Behaviour in an interface is a specification. It is an agreement between components providing and requiring this interface about the messages and protocol exchanged. Both components must adhere to this specification. Interface conformance is an imprtant check in the verifier. The code generator does not generate any code based on the behaviour specification of an interface.

Behaviour in a component is a definition. It defines the actual behaviour expressed by the object modeled. Whereas an interface might still allow some freedom, sometimes multiple messages are allowed, inside a component the behaviour must be fully deterministic. The code generator uses this component behaviour specification to generate target code.

3.6.2 Behaviour Syntax

The 'behaviour' keyword is used to declare a behaviour section. Optionally it may have a name.

```
behaviour [OptionalName]
{
  behaviour-body
}
```

The behaviour body contains a collection of statements that define the actions that will be performed based on events received.

The following elements can be used in the behaviour body:

- Local variables: See Section 3.4 [Using Local Variables], page 22,
- Functions: See Section 3.6.7 [Using functions], page 38,
- See Section 3.6.3 [Actions], page 30,
- State declarations and initialisations: See Section 3.6.4 [Specifying Stateful Behaviour], page 33,
- See Section 3.6.8 [Reply], page 40,
- Conditions: See Section 3.6.6 [Using Conditional Statements], page 37,
- Guards: See Section 3.6.5 [Using Guards], page 36,

3.6.2.1 Examples

3.6.2.2 A basic behaviour section

This example shows an interface with a basic behaviour section.

```
interface IMotion
{
  in void start();
  in void stop();
  behaviour
  {
      bool started = false;
      on start: started = true;
      on stop: started = false;
  }
}
```

3.6.2.3 A behaviour section with a protocol definition

This example shows an interface with a more complex behaviour section, where the start event is only allowed in case the interface is in state Off and the stop event is only allowed in case the interface is in state Active.

```
interface IMotion
{
  in void start();
  in void stop();
  behaviour
  {
      enum State {Off, Active};
      State state = State.Off;
      [state.Off]
      {
          on start: state = State.Active;
          on stop:  illegal;
      }
      [state.Active]
      {
          on start: illegal;
          on stop:  state = State.Off;
      }
   }
}
```

3.6.2.4 A behaviour section with a protocol definition, alternative form

This example shows an interface with the same behaviour as the one above, but an alternative form is used to define the behaviour.

In the previous example based on the state value (using guarded statements) the actions for the events per state were defined. In the alternative below based on the events the actions in each state are defined.

```
interface IMotion
{
  in void start();
  in void stop();
  behaviour
  {
      enum State {Off, Active};
      State state = State.Off;
      on start:
      {
          [state.Off] state = State.Active;
          [state.Active] illegal;
      }
      on stop:
      {
          [state.Off] illegal;
          [state.Active]  state = State.Off;
      }
   }
}
```

See also:

- Section 3.4 [Using Local Variables], page 22,
- Section 3.6.7 [Using functions], page 38,
- Section 3.6.3 [Actions], page 30,
- Section 3.6.4 [Specifying Stateful Behaviour], page 33,
- Section 3.6.8 [Reply], page 40,
- Section 3.6.6 [Using Conditional Statements], page 37,
- Section 3.6.5 [Using Guards], page 36,

3.6.3 Actions

The actions an interface or component should perform when an event is received are specified within a Behaviour (see Section 3.6 [Specifying Behaviour], page 28).

Actions can be changing the value of a variable, invoking a function, generating another event or changing to another state and any combination.

In response to one event trigger, there can be different actions to be taken as long as they have non-overlapping guards. Guards are boolean expressions. The statement behind the guard is selected when the expression evaluates to true. The keyword 'otherwise' defines a catch-all guard which is true only when none of the other guard expressions in a list of guarded statements evaluates to true.

Note that the use of guards is reserved for action selection. The guards in combination with the trigger event determine which action is activated. Thus, the action is a (compound) statement inside an on event statement (optionally) behind an innermost guard.

3.6.3.1 Syntax

Declaring an action statement:

```
on intf1.evt1(), intf2.evt2():
{
  action-body;
}
```

Here 'intf1.evt1' and 'intf2.evt2' are event instances, which refer to interface variables 'intf1' and 'intf2' and to events 'evt1' and 'evt2'.

Intentionally the 'Statement' will be executed when one of the two triggers occurs.

action-body: a collection of conditions / guards and statements (function invocations, changing variable values, changing to another state, setting a reply value, generating another event, marking the event as illegal) that define what actions to perform.

Declaring a guarded action statement:

```
[state.State1]
on intf1.evt1(), intf2.evt2():
{
  action-body;
}
```

Here the action statement will only be selected when guard [state.State1] evaluates to true.

Special action statements can defined that will be performed in case no other events occur, or that optionally may be performed in case no other events occur, see Section 3.6.9 [Using inevitable and optional], page 43.

3.6.3.2 Examples

3.6.3.3 Simple state machine with guarded action statements

This example shows a simple state machine for a siren.

Here the value of a variable representing the state of the state machine is used as a guard to select which action statements to evaluate.

```
interface Siren
{
  in void TurnOn();
  in void TurnOff();
  behaviour
  {
    enum State { SirenOff, SirenOn }; // type declaration
    State state = State.SirenOff;     // variable declaration
    [state.SirenOff] // if the Siren is off, only allow to turn it on
    {
```

```
      on TurnOn:  state = State.SirenOn;
      on TurnOff: illegal;
    }
    [state.SirenOn] // if the Siren is on, only allow to turn it off
    {
      on TurnOff: state = State.SirenOff;
      on TurnOn:  illegal;
    }
  }
}
```

3.6.3.4 Two alternative styles to define actions based on events

This example shows an interface with two alternative styles to define the actions based on incoming events.

```
interface IPsm
{
  in void connect();
  in void disconnect();
  out void connected();
  out void error();
  behaviour
  {
    enum ProtocolState {Disconnected, Connected}; // type declaration
    ProtocolState state = ProtocolState.Disconnected; // variable declaration
    bool isError = false; // variable declaration

    // Alternative 1: using guards based on e.g.  state value inside the action statem
    on connect:
    {
      // Select action to be performed based on state variable values.
      [state.Disconnected && !isError]
      { // Action
        connected; // Output event
        state = ProtocolState.Connected; // Assign to state variable
      }
      [otherwise]
      { // Action
        error; // Output event
        isError = true; // Assign to state variable
      }
    }
    // Alternative 2: using guards based on state variable outside the action statemen
    {
      [state.Connected]
      {
        on disconnect:
        {
```

```
            [isError]
            { // Action
              isError = false;
            }
            [otherwise]
            { // Action
              state = ProtocolState.Disconnected;
            }
          }
        }
        [otherwise]
        {
          on disconnect:
          {
            error;
            isError = true;
          }
        }
      }
    }
  }
```

See also:

- Section 3.6 [Specifying Behaviour], page 28,
- Section 3.6.4 [Specifying Stateful Behaviour], page 33,
- Section 3.6.7 [Using functions], page 38,
- Section 3.6.6 [Using Conditional Statements], page 37,
- Section 3.6.10 [Using illegal], page 45,
- Section 3.6.8 [Reply], page 40,
- Section 3.6.9 [Using inevitable and optional], page 43,

3.6.4 Specifying Stateful Behaviour

State variables and guard expressions enable specification of protocols in a behaviour section. They specify the selection of one of multiple different actions. See Section 3.6.3 [Actions], page 30, for an input event.

The state of an interface is composed of the values of all state variables.

The state diagram of an interface takes only the first variable declared into account.

State variable declarations can refer to public types (e.g. defined for use as event return values) or to private type declarations local to the 'behaviour' section. Types can be boolean, integer with limited contiguous range or enumerated.

Guards are boolean expressions based on state variables. The statement behind the guard is selected when the expression evaluates to true. The keyword 'otherwise' defines a catch-all guard which is true only when none of the other guard expressions in a list of guarded statements evaluates to true.

3.6.4.1 Syntax

Declaring a state variable type based on an enum.

```
enum State {State1, State2};
```

Declaring a state variable based on the above type and initialising it:

```
State state = State.State1;
```

Declaring a guarded action statement

```
[state.State1]
{
  action-statement ;
}
```

Here the action statement will only be selected when guard [state.State1] evaluates to true.

See Section 3.6.3 [Actions], page 30, for how to declare action statements.

3.6.4.2 Examples

3.6.4.3 Simple state machine

This example shows a simple state machine for a siren.

```
interface Siren
{
  in void TurnOn();
  in void TurnOff();
  behaviour
  {
    enum State { SirenOff, SirenOn }; // type declaration
    State state = State.SirenOff;     // variable declaration
    [state.SirenOff] // if the Siren is off, only allow to turn it on
    {
      on TurnOn:  state = State.SirenOn;
      on TurnOff: illegal;
    }
    [state.SirenOn] // if the Siren is on, only allow to turn it off
    {
      on TurnOff: state = State.SirenOff;
      on TurnOn:  illegal;
    }
  }
}
```

3.6.4.4 Two alternative styles to define actions based on events

This example shows an interface with two alternative styles to define the actions based on incoming events.

```
interface IPsm
{
```

```
  in void connect();
  in void disconnect();
  out void connected();
  out void error();
  behaviour
  {
    enum ProtocolState {Disconnected, Connected}; // type declaration
    ProtocolState state = ProtocolState.Disconnected; // variable declaration
    bool isError = false; // variable declaration
    // Alternative 1: using guards based on e.g.  state value inside the action statem
    on connect:
    {
      // Select action to be performed based on state variable values.
      [state.Disconnected && !isError]
      { // Action
        connected; // Output event
        state = ProtocolState.Connected; // Assign to state variable
      }
      [otherwise]
      { // Action
        error; // Output event
        isError = true; // Assign to state variable
      }
    }

interface IPsm
{
  in void connect();
  in void disconnect();
  out void connected();
  out void error();
  behaviour
  {
    enum ProtocolState {Disconnected, Connected}; // type declaration
    ProtocolState state = ProtocolState.Disconnected; // variable declaration
    bool isError = false; // variable declaration
    // Alternative 2: using guards based on state variable outside the action statemen
    {
      [state.Connected]
      {
        on disconnect:
        {
          [isError]
          { // Action
            isError = false;
          }
          [otherwise]
```

```
            { // Action
              state = ProtocolState.Disconnected;
            }
          }
        }
        [otherwise]
        {
          on disconnect:
          {
            error;
            isError = true;
          }
        }
      }
    }
  }
```

See also:

- Section 3.6 [Specifying Behaviour], page 28,
- Section 3.6.3 [Actions], page 30,

3.6.5 Using Guards

Guards are boolean expressions based on variables. The statement behind the guard is selected if the expression evaluates to true. If more than one guard evaluates to true, a non-deterministic choice for one of guards must be made. Guards can be nested.

3.6.5.1 Syntax

An expression between [and] is used to declare a guarded statement

```
[expression] statement;
```

Here the statement will only be executed when expression evaluates to true.

Expression can be a compound expression using the logic operators || (OR) and &&(AND):

```
[expression1 || expression2] statement;  // statement will be executed when expression
[expression1 && expression2] statement;  // statement will be executed when exrpession
```

The keyword 'otherwise' defines a catch-all guard which is true only when none of the other guard expressions in a list of guarded statements evaluates to true.

```
[expression1] statement-true-1;
[expression2] statement-true-2;
[otherwise]   statement-false-1-2;
```

Here statement-true-1 will be executed when expression1 evaluates to true, statement-true-2 will be executed when expression2 evaluates to true and statement-false-1-2 will be executed when both expression1 and expression2 evaluate to false.

3.6.5.2 Examples

3.6.5.3 Simple state machine

This example shows a simple state machine for a siren where guards are used to select the correct actions based on the state.

```
interface Siren
{
  in void TurnOn();
  in void TurnOff();
  behaviour
  {
    enum State { SirenOff, SirenOn }; // type declaration
    State state = State.SirenOff;    // variable declaration
    [state.SirenOff] // if the Siren is off, only allow to turn it on
    {
      on TurnOn:  state = State.SirenOn;
      on TurnOff: illegal;
    }
    [state.SirenOn] // if the Siren is on, only allow to turn it off
    {
      on TurnOff: state = State.SirenOff;
      on TurnOn:  illegal;
    }
  }
}
```

See also:

- Section 3.6.3 [Actions], page 30,
- Section 3.6.6 [Using Conditional Statements], page 37,

3.6.6 Using Conditional Statements

Conditional statements (if-else) can be used to program decision making on which actions to perform

3.6.6.1 Syntax

The keyword if is used for a simple conditional statement:

```
if (expression) statement-true;
```

In this case when the expression is true then statement-true will be executed.

An if-else construction can be used for programming more elaborate decisions.

```
if (expression) statement-true;
else statement-false;
```

if the expression is true then statement-true will be executed, else statement-true is skipped and statement-false is executed.

3.6.6.2 Examples

3.6.6.3 A basic if-else example

This example shows an interface with a behaviour section where if else is used to select the actions to be performed based on the value of a state and a local variable.

```
interface IImperativeAction
{
  in void doIt();
  out void done();
  out void skip();
  behaviour
  {
    enum Toggle {Ping, Pong};
    Toggle state = Toggle.Ping;
    on doIt:
    {
      bool localVariable = false;
      if (state.Ping)
      {
        localVariable = true;
      }
      else
      {
        localVariable = (state == Toggle.Ping);
      }
      if (localVariable)
      {
        done;
        state = Toggle.Pong;
      }
      else
      {
        skip;
        state = Toggle.Ping;
      }
    }
  }
}
```

See also:

- Section 3.6.5 [Using Guards], page 36,

3.6.7 Using functions

Functions are defined in a behaviour section.

These functions can be called as part of an action.

Functions can access global variables.

3.6.7.1 Syntax

Function declaration:

```
return-type function-name (parameter-list)
{
  function-body;
}
```

A function definition consists of a function header and a function body.

Here are all the parts of a function:

- return-type: The return-type is the data type of the value the function returns. Use the keyword void for functions that do not return a value.
- function-name: This is the actual name of the function.
- parameter-list: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter.

  This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- function-body: The function body contains a collection of statements that define what the function does.

3.6.7.2 Examples

3.6.7.3 A basic function

This example shows an interface with a behaviour section where a function is defined and used to determine the proper actions.

```
interface IFunctionCallAction
{
  in void doIt();
  out void done();
  out void skip();
  behaviour
  {
    enum Toggle {On, Off};
    Toggle state = Toggle.On;
    bool shouldWeDoIt()                  // the function returns a Boolean value, is c
    {
      bool result = false;
      result = (state == Toggle.On);
      return result;                     // the result of the evaluation is returned
    }
    on doIt:
    {
      bool localVariable = shouldWeDoIt(); // the function is called and the return va
      if (localVariable)
      {
```

```
          done;
          state = Toggle.Off;
        }
        else
        {
          skip;
          state = Toggle.On;
        }
      }
    }
  }
```

See also:

- ?

3.6.8 Reply

The 'reply' keyword assigns a value to be returned. The 'reply' keyword also specifies that a blocked thread is allowed to return when all statements in the enclosing action block have been performed.

A 'reply' statement may appear anywhere in an 'action'. Execution of an action block continues after a 'reply' statement is performed. Thread returns occur when the current execution thread reaches the end of the current action block. Then all threads return for which a 'reply' has been executed in the current action block.

Void 'reply' must be omitted in interface models and in non-blocking component action statements. In these cases the void reply is implicit.

3.6.8.1 Syntax

1) `reply(variable-name);`

2) `reply(CONSTANT);`

3) `port-name.reply(); (since 2.0.0)`

4) `port-name.reply(variable-name); (since 2.0.0)`

5) `port-name.reply(CONSTANT); (since 2.0.0)`

Explanation:

1) Set the value to be replied to the value currently held by variable-name. (This form applies to interfaces. This form must also be used in component models when the only active thread is associated with the port for the current 'on event' and the current action block is not marked as 'blocking'.)

2) Set the value to be replied to a defined constant (e.g. an enumerated type). (This form applies to interfaces. This form must also be used in component models when the only active thread is associated with the port for the current 'on event' and the current action block is not marked as 'blocking'.)

3) Allow void return on port with name 'port-name'. (This form applies to component models when the current 'action' was specified as 'blocking' or when port 'port-name' is still blocked.)

4) Allow valued return on port with name 'port-name' and assign the value held by 'variable-name' to the reply value for 'port-name'. (This form applies to component models when the 'action' was specified as 'blocking' or when port 'port-name' is still blocked.)

5) Allow valued return on port with name 'port-name' and assign the value held by 'CONSTANT' to the reply value for 'port-name'. (This form applies to component models when the 'action' was specified as 'blocking' or when port 'port-name' is still blocked.)

3.6.8.2 Examples

3.6.8.3 Setting an enum as reply value.

This example shows how to set a reply value using an enumerated type.

```
interface iNetwork
{
// interface to the Network.
// The Network can be requested to send a message by using sendMessage.
// When the message has been sent the Network will inform it's client
// by using messageSent.
  enum MsgRecRes
  {
    OK,
    Failed
  };
  in MsgRecRes sendMessage();
  out void messageSent();
  behaviour
  {
    enum State {Idle, Busy};
    State state = State.Idle;
    [state.Idle]
    // initial or idle state
    {
      on sendMessage: // the interface must cover the case that a message is received
      {
        reply(MsgRecRes.OK);
        state = State.Busy;
      }
      on sendMessage: // the interface must cover the case that a message is not recei
      {
        reply(MsgRecRes.Failed);
      }
    }
    ...
  }
}
```

3.6.8.4 Thread signalling

Component CA illustrates the use of 'reply' to signal blocked thread release. The action statement in 'on pp.request*' are marked as 'blocking' and do not contain a reply statement. A thread calling pp.request_reply or pp.request_notify will block after calling rp.request. That thread returns when the action statement for 'on rp.accept' or 'on rp.reject' have been performed.

```
interface IBlockingRequest
{
  in bool request_reply();
  in void request_notify();
  out void accept();
  out void reject();

  behaviour
  {
    on request_reply: reply(true);
    on request_reply: reply(false);
    on request_notify: accept;
    on request_notify: reject;
  }
}
interface INonBlockingRequest
{
  in void request();
  out void accept();
  out void reject();
  behaviour
  {
    bool is_considering = false;
    on request: is_considering = true;
    [is_considering] on inevitable: {accept; is_considering = false;}
    [is_considering] on inevitable: {reject; is_considering = false;}
  }
}
component CA
{
  provides IBlockingRequest pp;
  requires INonBlockingRequest rp;
  behaviour
  {
    bool reply_requested = true;
    on pp.request_reply(): blocking {reply_requested = true; rp.request();}
    on pp.request_notify(): blocking {reply_requested = false; rp.request();}
    [reply_requested] on rp.accept(): pp.reply(true);
    [!reply_requested] on rp.accept(): {pp.accept(); pp.reply();}
    [reply_requested] on rp.reject(): {bool b = false; pp.reply(b);}
```

```
      [!reply_requested] on rp.reject(): {pp.reject(); pp.reply();}
    }
  }
```

See also:

- Section 3.6.3 [Actions], page 30,
- Section 3.6.12 [Blocking], page 48,

3.6.9 Using inevitable and optional

To specify in an interface model that the implementation behind the interface will at a certain moment perform an action requiring a response to be sent over the interface the keyword inevitable or optional can be used.

Inevitable means that when no other events occur, this event will occur. Optional means that the event may or may not occur.

Note that an Inevitable event is not guaranteed to occur always. If there are other (optional or inevitable) events that can occur, the inevitable event may or may not occur. So it is only inevitable in the absence of other events.

The difference between the two is how the event is handled by the model verification. For inevitable events, the verification only checks the cases where the event does occur, i.e. it assumes that the event will eventually occur if no other events occur. For optional events, the model verification also checks the cases where the event never happens (which could be a cause of deadlock). Note: when both optional and inevitable events are used in the same state, it is only guaranteed that one of the two events will occur, but it is not guaranteed that the inevitable event will always occur.

3.6.9.1 Syntax

Declaration of an inevitable or optional event trigger and the actions that need to be performed as a result of it:

```
on inevitable: action-body ; // collection of optional conditions / guards and stateme
on optional: action-body ;
```

The declaration of inevitable and optional actions follows the declaration of actions (see Section 3.6.3 [Actions], page 30), where action-body: a collection of conditions / guards and statements (function invocations, changing variable values, changing to another state, generating another event) that define what actions to perform.

3.6.9.2 Examples

3.6.9.3 An inevitable time-out

This example shows the interface specification for a timer.

As it is up to the implementation to define exactly how the timer will expire, the (internal or implementation specific) trigger leading to that the time-out event has to be send over the Timer interface can not be specified in the interface definition. However, we define here that that inevitably the timer will expire and the time-out event has to be send.

```
interface iTimer
{
```

```
// interface for a basic timer
    in void createTimer();
    in void cancelTimer();
    out void timeout();
    behaviour
    {
        enum State {Idle, Busy};
        State state = State.Idle;
        [state.Idle]
        {
            on createTimer: {state = State.Busy;}
            on cancelTimer: illegal;
        }
        [state.Busy]
        {
            on createTimer: illegal;
            on cancelTimer: {state = State.Idle;}
            on inevitable:
            {
                timeout;
                state = State.Idle;
            }
        }
    }
}
```

3.6.9.4 An interface for a sensor with an optional event when the sensor is triggered

This example shows the interface specification for a sensor.

As it is up to the implementation to define exactly what happens when the sensor is triggered, the (internal or implementation specific) trigger leading to that the triggered event has to be send over the Sensor interface can not be specified in the interface definition. However, we define here that that optionally the sensor might be triggered and the triggered event has to be send.

```
// Interface to a Sensor
// The Sensor can be enabled via enable
// The Sensor can be disabled via disable
// The Sensor indicates that it is triggered via triggered
interface Sensor
{
  in void enable();
  in void disable();
  out void triggered();
  behaviour
  {
    enum States { Disabled, Enabled, Triggered };
```

```
    States state = States.Disabled;
    [state.Disabled]
    {
      on enable:        state = States.Enabled;
      on disable:       illegal;
    }
    [state.Enabled]
    {
      on enable:        illegal;
      on disable:       state = States.Disabled;
      on optional:      { triggered; state = States.Triggered; }
    }
    [state.Triggered]
    {
      on enable:        illegal;
      on disable:       state = States.Disabled;
    }
  }
}
```

See also:

- Section 3.6.3 [Actions], page 30,

3.6.10 Using illegal

To specify that an event is not allowed at a certain stage, the keyword illegal can be used.

3.6.10.1 Syntax

The keyword illegal is used to indicate that an event is not allowed:

```
on intf1.evt1(), intf2.evt2(): illegal;
```

Here 'intf1.evt1' and 'intf2.evt2' are event instances, which refer to interface variables 'intf1' and 'intf2' and to events 'evt1' and 'evt2'.

Note: **do not use illegal in conditional statements (if / if else) or functions in interface definitions** ! This is because an event can only be declared illegal in a direct way due to the declarative character of interfaces. (Or in other words: in an interface one declares the expected behaviour and an illegal inside conditional statements or functions would require an evaluation of the conditional statement or function for the be expected behaviour.)

3.6.10.2 Examples

3.6.10.3 A timer where depending on the state certain events are allowed and others not

This example shows the interface specification for a timer. Initially a timer can be created and it is not allowed to cancel the timer before creation. When the timer has been created it is not possible to request again creation of the timer.

```
interface iTimer
```

```
{
// interface for a basic timer
    in void createTimer();
    in void cancelTimer();
    out void timeout();
    behaviour
    {
        enum State {Idle, Busy};
        State state = State.Idle;
        [state.Idle]
        {
            on createTimer: state = State.Busy;
            on cancelTimer: illegal;
        }
        [state.Busy]
        {
            on createTimer: illegal;
            on cancelTimer: state = State.Idle;
            on inevitable:
            {
                timeout;
                state = State.Idle;
            }
        }
    }
}
```

3.6.10.4 Incorrect and correct interface definition with a conditional illegal

An interface has to be defined that does not allows a second event (checkModel in this case) before the asynchronous reply (checkModelRes) has been sent to the first event.

The following code snippet shows an incorrect interface definition with an illegal inside an if-else construction.

```
[state.Authenticated]
   {
     on checkModel:
     {
        if (ModelCheckOngoing == false)
        {
           reply(res.Ok);
           ModelCheckOngoing = true;
        }
        else illegal;
     }
     on inevitable:
     {
```

```
            if (ModelCheckOngoing == true)
            {
                ModelCheckOngoing = false;
                checkModelRes;
            }
        }
    }
}
```

The following snippet shows how to correctly define this:.

```
[state.Authenticated]
{
      [!ModelCheckOngoing] on checkModel:
      {
          reply(res.Ok);
          ModelCheckOngoing = true;
      }
      [ModelCheckOngoing] on checkModel: illegal;
      [ModelCheckOngoing] on inevitable:
      {
        ModelCheckOngoing = false;
        checkModelRes;
      }
}
```

See also:

- Section 3.6.3 [Actions], page 30,

3.6.11 Restrictions Multiple Provides

For a component with a behaviour and with multiple provides ports, two restrictions hold. These restrictions relate to the fact that activity cannot be forked. These are the restrictions:

1) Within the handling of an in-event of a provides port, it is not allowed to post an out-event on another provides port.

2) Within the handling of an out-event of a requires port, it is not allowed to post an out-event to more than one provides port.

The violation of any of these restrictions will be reported as a compliance error.

These are examples of the two types of violations:

```
interface Intf
{
    in void e();
    out void c();
    behaviour
    {
        on e: {}
        on optional: c;
    }
```

```
}
component ViolationType1
{
    provides Intf p0;
    provides Intf p1;
    behaviour
    {
        on p0.e(): p1.c(); // compliance error: p1.c not allowed by port p1
        on p1.e(): {}
    }
}
component ViolationType2
{
    provides Intf p0;
    provides Intf p1;
    requires Intf r;
    behaviour
    {
        on p0.e(), p1.e(): {}
        on r.c(): { p0.c(); p1.c(); } // compliance error: p1.c not allowed by port p1
    }
}
```

See also:

- The example project MultipleProvides.

3.6.12 Blocking

(since 2.0.0)

The 'blocking' keyword allows omission of a reply statement from an action statement and supresses the implicit void reply performed after the last statement in an action statement. An action statement in another port in-event should perform the reply statement for the blocked port. Thus, time and value of a blocked port reply depend on another in-event. The keyword 'blocking' is allowed to appear once in the path of statements prefixing an action statement.

Only 'provides' ports are affected by 'blocking'. A call of a provided port in-event will not return before a reply statement is performed for that port.

Guard expressions or 'on event' is commutative with respect to blocking. If 'blocking' appears before a guard or 'on event' it applies to the action statement after the guard or 'on event'.

Note: 'blocking' may only be used in components with a single 'provides' port. This limitation may be lifted in a future release.

Note: Systems containing 'blocking' component instances must be contained in a thread-safe shell (see Section 6.2 [Thread-safe Shell], page 78).

3.6.12.1 Syntax

```
on event: blocking action-statement; (1)
```

```
blocking on event: action-statement; (2)
on event: blocking [guard] action-statement; (3)
on event: {blocking [guard] action-statement1; [guard] action-statement2;} (4)
```

Explanation:

2) The 'blocking' keyword applies to the action-statement following 'on event:'. This form is semantically equivalent to 1).

3) The 'blocking' keyword applies to the action-statement following [guard]. This form is semantically equivalent to "on event: [guard] blocking action-statement;"

4) The 'blocking' keyword applies to action-statement1. It does NOT apply to action-statement2.

3.6.12.2 Examples

3.6.12.3 Thread signalling

Component CA illustrates the use of 'blocking' to postpone thread return. An 'on event' on a different port performs the 'reply' statement for the blocked port.

The action statement in 'on pp.request_value' is marked as 'blocking' and does not contain a reply statement. A thread calling pp.request_reply blocks after action statement 'rp.request()'. The blocked thread returns when the action statement for 'on rp.accept' or 'on rp.reject' have been performed.

```
interface IBlockingRequest
{
  in bool request_value();
  in void request_event();
  out void accept();
  out void reject();

  behaviour
  {
    on request_value: reply(true);
    on request_value: reply(false);
    on request_event: accept;
    on request_event: reject;
  }
}
interface INonBlockingRequest
{
  in void request();
  out void accept();
  out void reject();
  behaviour
  {
    bool is_considering = false;
    on request: is_considering = true;
    [is_considering] on inevitable: {accept; is_considering = false;}
```

```
    [is_considering] on inevitable: {reject; is_considering = false;}
  }
}
component CA
{
  provides IBlockingRequest pp;
  requires INonBlockingRequest rp;
  behaviour
  {
    bool is_valued_event = true;
    on pp.request_value(): blocking {is_valued_event = true; rp.request();}
    blocking on pp.request_event(): {is_valued_event = false; rp.request();}
    [is_valued_event] on rp.accept(): pp.reply(true);
    [!is_valued_event] on rp.accept(): {pp.accept(); pp.reply();}
    [is_valued_event] on rp.reject(): {bool b = false; pp.reply(b);}
    [!is_valued_event] on rp.reject(): {pp.reject(); pp.reply();}
  }
}
```

See also:

- Section 3.6.8 [Reply], page 40,
- Section 3.6.3 [Actions], page 30,
- Section 6.2 [Thread-safe Shell], page 78,

3.7 Decomposing a Component

A component can be decomposed into a group of components. A grouping of components is called a *system*. A component that is decomposed into a system has no own behaviour definition, the behaviour of the system is defined by the component into which it is decomposed.

3.7.1 Systems of Components

The components within a system can also be composite components, so it is possible to define systems of systems.

Composite components can also expose ports of contained components by binding their own provided or required ports to ports of contained components. In that case the binding looks like

```
port <=> component1_instance_name.port
```

3.7.1.1 Syntax

The keyword *system* is used to define a composition. Inside the system definition the instances of the components that are contained in the system are specified.

To define a system inside the composite component use:

```
component Composition
{
```

```
  system
  {
    Component1 component1instance;
    Component2 component2instance;
// binding of the components:
// where Component1 provides or requires port1
// and Component2 requires or provides port1
    component1instance.port1 <=> component2instance.port1;
  }
}
```

Within a system components have to be connected together. A binding statement takes the form of

```
component1_instance_name.port <=> component2_instance_name.port
```

Only ports with an equal interface definition and opposite direction can be bound together.

Composite components can also expose ports of contained components by binding their own provided or required ports to ports of contained components. In that case the binding looks like

```
port <=> component1_instance_name.port
```

```
component Composition
{
  provides Interface1 port1;
  system
  {
    Component1 component1instance;
    Component2 component2instance;
// binding of the components:
// where Component1 provides port1;
// where Component1 provides or requires port2
// and Component2 requires or provides port2
    port1 <=> component1instance1.port1
    component1instance.port2 <=> component2instance.port2;
  }
}
```

3.7.1.2 Examples

3.7.1.3 Decomposition into components and binding their ports.

Here a component is decomposed into two other components C1 and C2.

```
interface I1
{
  in void event();
}
component C1
{
```

```
  provides I1 i1;
}
component C2
{
  requires I1 i1;
}
component Decomposition
{
  system
  {
    C1 c1;
    C2 c2;
    c1.i1 <=> c2.i1;
  }
}
```

3.7.1.4 Exposing ports

Here a component is decomposed into two other components C1 and C2 and the interface provided by component C2 is exposed.

```
interface I1
{
  in void event();
}
interface I2
{
  in void event();
}
component C1
{
  provides I1 i1;
}
component C2
{
  provides I2 i2;
  requires I1 i1;
}
component Decomposition
{
  provides I2 i2Proxy;
  system
  {
    C1 c1;
    C2 c2;
    i2Proxy <=> c2.i2;
    c1.i1 <=> c2.i1;
  }
```

```
}
```

See also:

- Section 3.7.2 [Specifying Ports], page 53,
- Section 3.7.3 [Binding Ports], page 53,

3.7.2 Specifying Ports

See Section 3.3.3 [Ports], page 17,

3.7.3 Binding Ports

A component can be decomposed into a group of components. A grouping of components is called a *system*.

Component instances in the composition are connected by binding their ports. Only ports with an equal interface definition and opposite direction can be bound together.

3.7.3.1 Syntax

A binding statement takes the form of

```
component1_instance_name.port <=> component2_instance_name.port
```

Only ports with an equal interface definition and opposite direction can be bound together.

3.7.3.2 Examples

3.7.3.3 Decomposition into components and binding their ports.

Here a component is decomposed into two other components C1 and C2.

```
interface I1
{
  in void event();
}
component C1
{
  provides I1 i1;
}
component C2
{
  requires I1 i1;
}
component Decomposition
{
  system
  {
    C1 c1;
    C2 c2;
    c1.i1 <=> c2.i1;
  }
}
```

See also:

- Section 3.7 [Decomposing a Component], page 50,

4 Execution Semantics

The execution semantics of Dezyne are illustrated using different model examples and the corresponding sequence diagrams.

When interpreting the models and corresponding sequence diagrams, keep in mind that the body of an event is executed atomically in the context of its behaviour.

For an in-event all action statements are executed depth-first. All out-events are stored in event queues at the receiving components. After the completion of all action statements, just before control is passed back to the caller, a component will flush its own queue of pending out events. Recursively all out-events are handled this way.

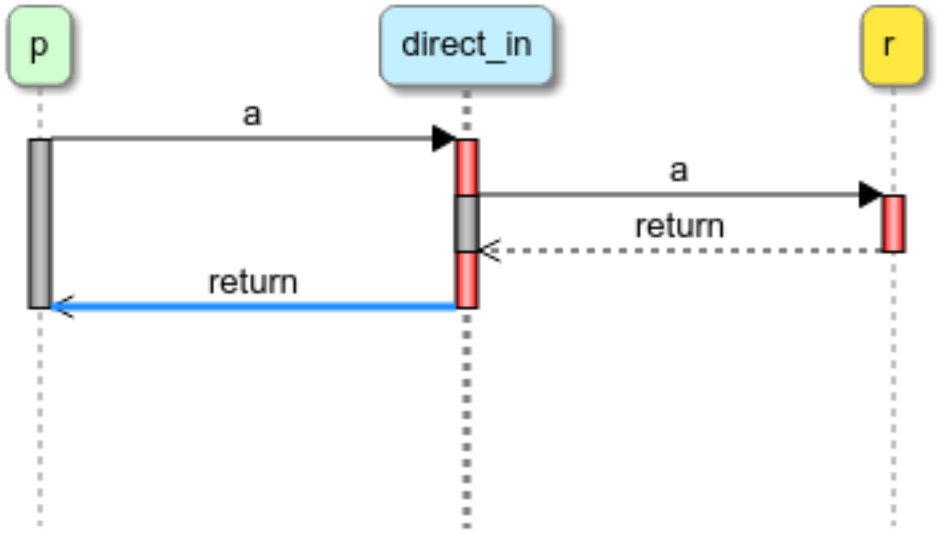
4.1 Direct in event

A provides port in-event (p.a) call resulting in a requires port in-event (r.a) is implemented as a function calling another function.

```
interface I
{
 in void a();
 behaviour
 {
   on a: {}
 }
}

component direct_in
{
  provides I p;
  requires I r;
  behaviour
  {
    on p.a(): r.a();
  }
```

```
}
```



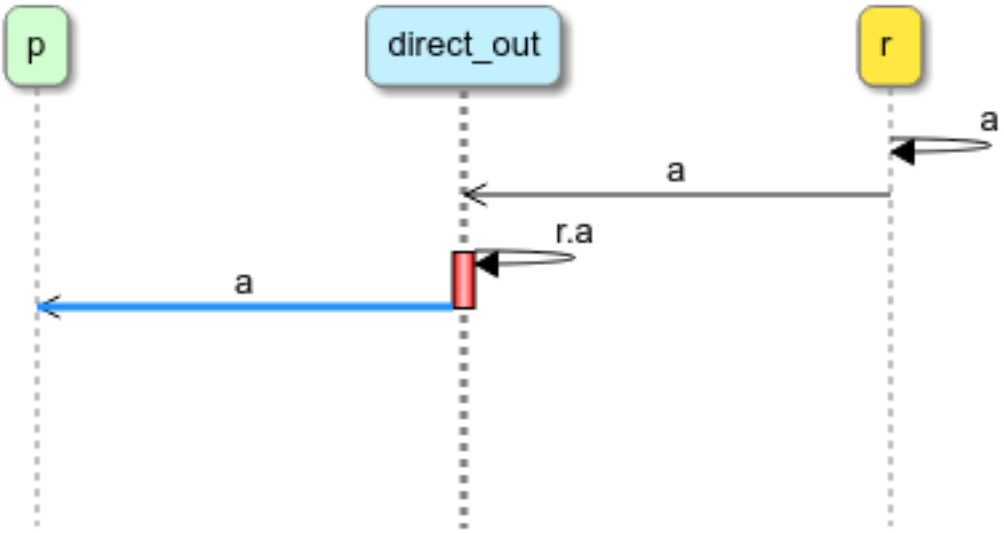
4.2 Direct out event

A requires port out-event (r.a) resulting in a provides port out-event (p.a) is implemented as a function posting an event in the component queue followed by a call to flush the queue.

```
interface I
{
  out void a();
  behaviour
  {
    on inevitable: a;
  }
}

component direct_out
{
  provides I p;
  requires I r;
  behaviour
  {
    on r.a(): p.a();
  }
```

```
}
```



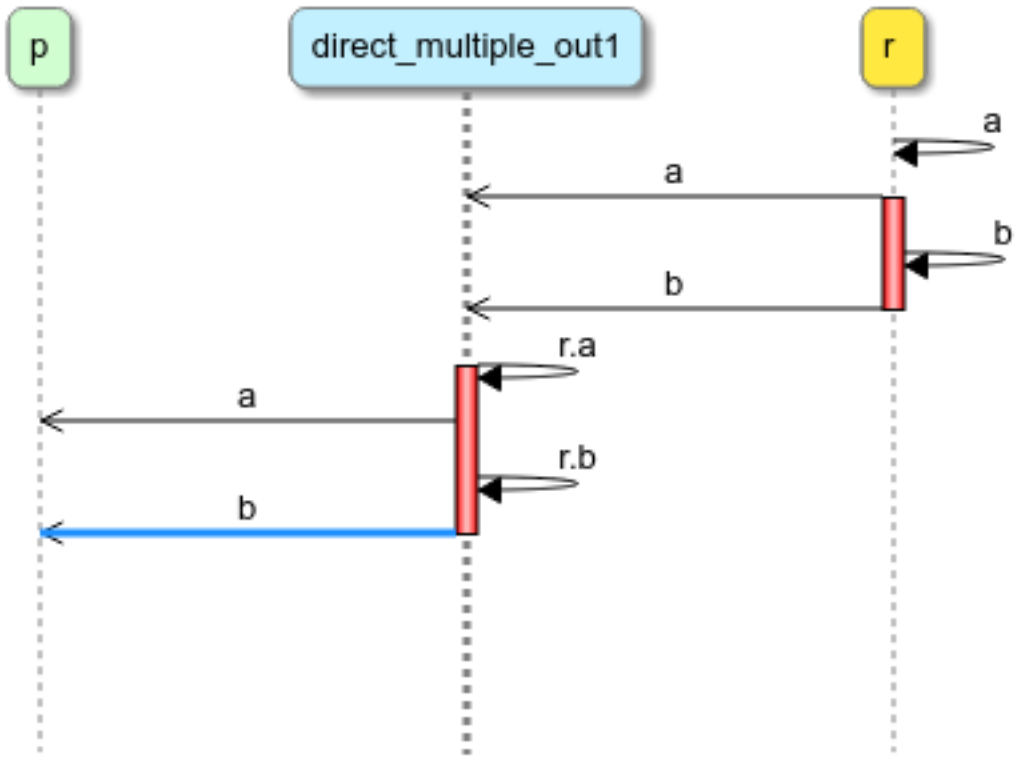
4.3 Direct multiple out events

A requires port inevitably triggering multiple out-events (r.a, r.b) is implemented as one function call for each out-event posting in the component queue, followed by a single flush call to trigger component processing of the events. The below 2 versions of the component are indistinguisable looking from the outside.

```
interface I
{
  out void a();
  out void b();
  behaviour
  {
    on inevitable: {a; b;}
  }
}

component direct_multiple_out1
{
  provides I p;
  requires I r;
  behaviour
  {
    on r.a(): p.a();
    on r.b(): p.b();
  }
```

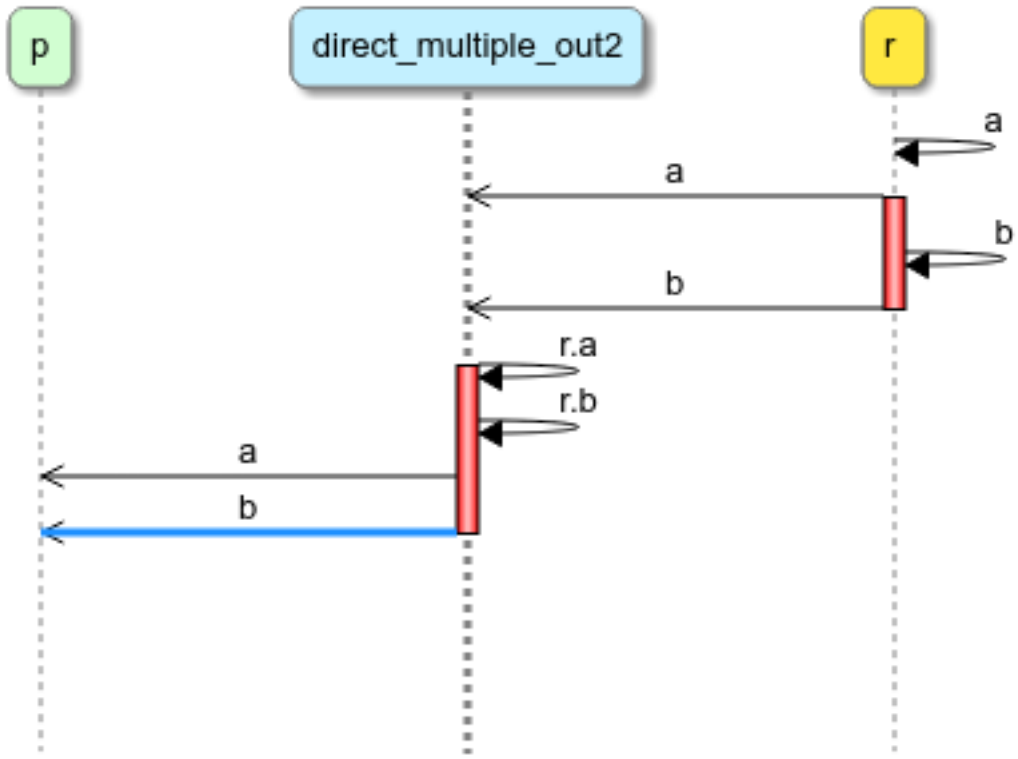
```
}
```




```
import direct_multiple_out.dzn;

component direct_multiple_out2
{
  provides I p;
  requires I r;
  behaviour
  {
    on r.a(): {}
    on r.b(): {p.a(); p.b();}
  }
```

```
}
```



The third variant is left as an exercise to the reader.

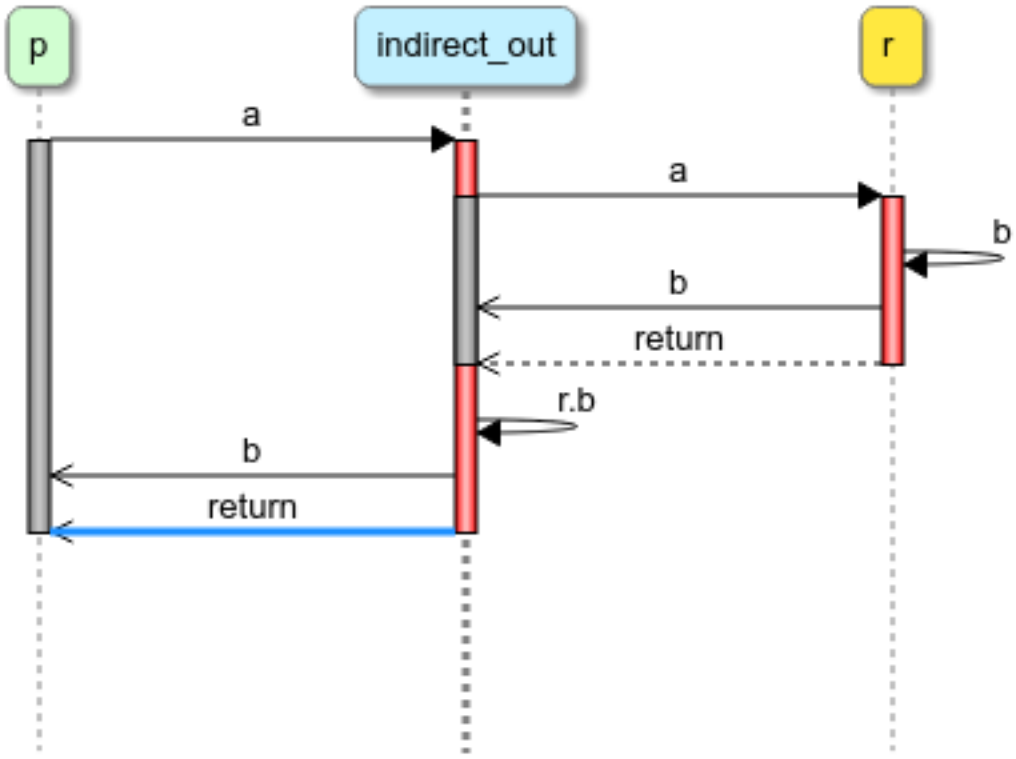
4.4 Indirect out event

A requires port out-event (r.b) posted in the context of a provides port in-event (p.a) call is processed before the provides port in-event (p.a) returns.

```
interface I
{
  in void a();
  out void b();
  behaviour
  {
    on a: b;
  }
}

component indirect_out
{
  provides I p;
```

```
  requires I r;
  behaviour
  {
    on p.a(): r.a();
    on r.b(): p.b();
  }
}
```



4.5 Indirect multiple out events

Since the provided interface is the same in the 3 cases below the externally visible behaviour is identical.

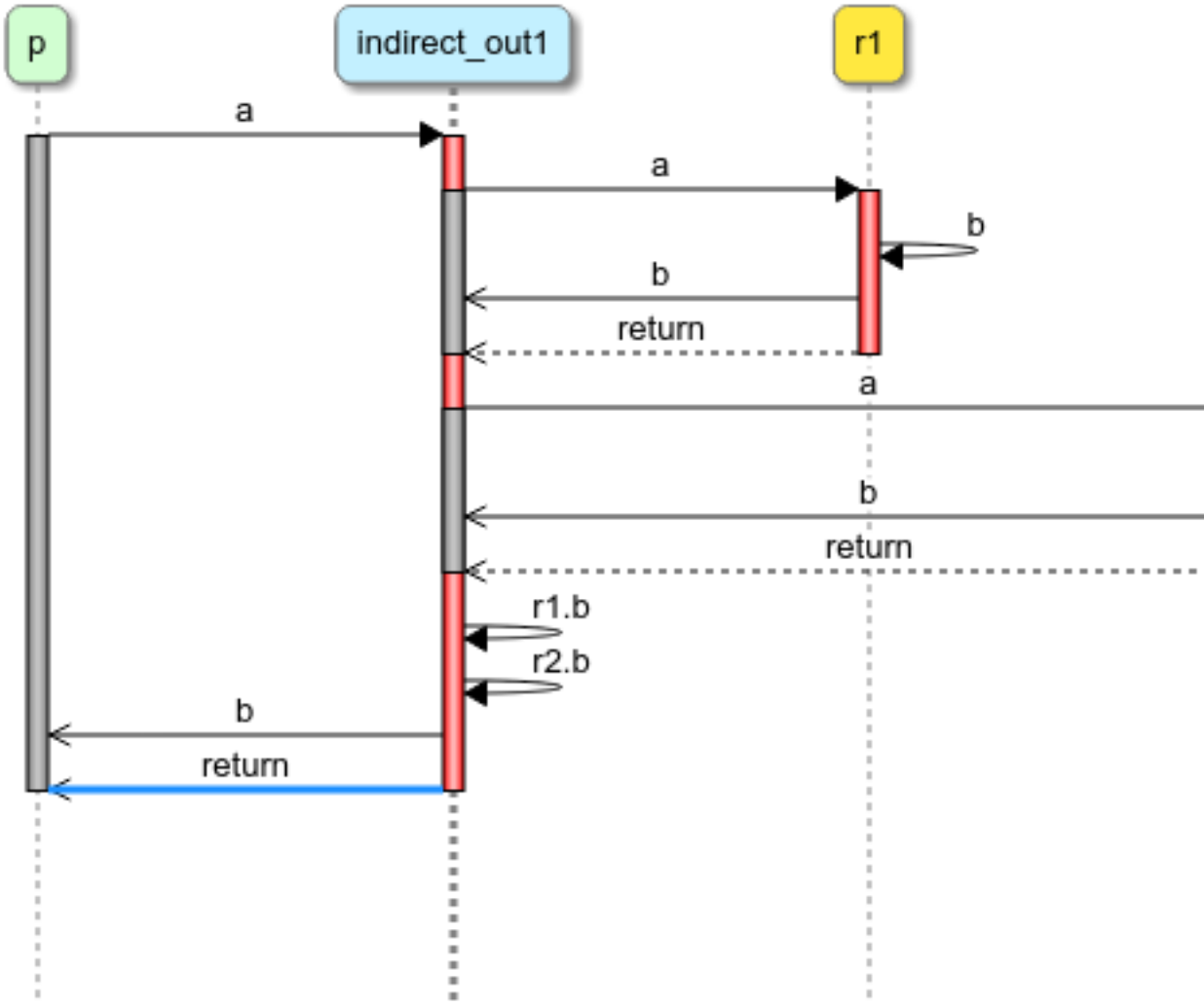
The 3 different behaviour implementations of the component show the subtle differences in the internal handling of messages.

```
interface I
{
  in void a();
  out void b();
  behaviour
  {
```

```
    on a: b;
  }
}

component indirect_multiple_out1
{
  provides I p;
  requires I r1;
  requires I r2;
  behaviour
  {
    on p.a(): {r1.a(); r2.a();}
    on r1.b(): {}
    on r2.b(): p.b();
  }
```

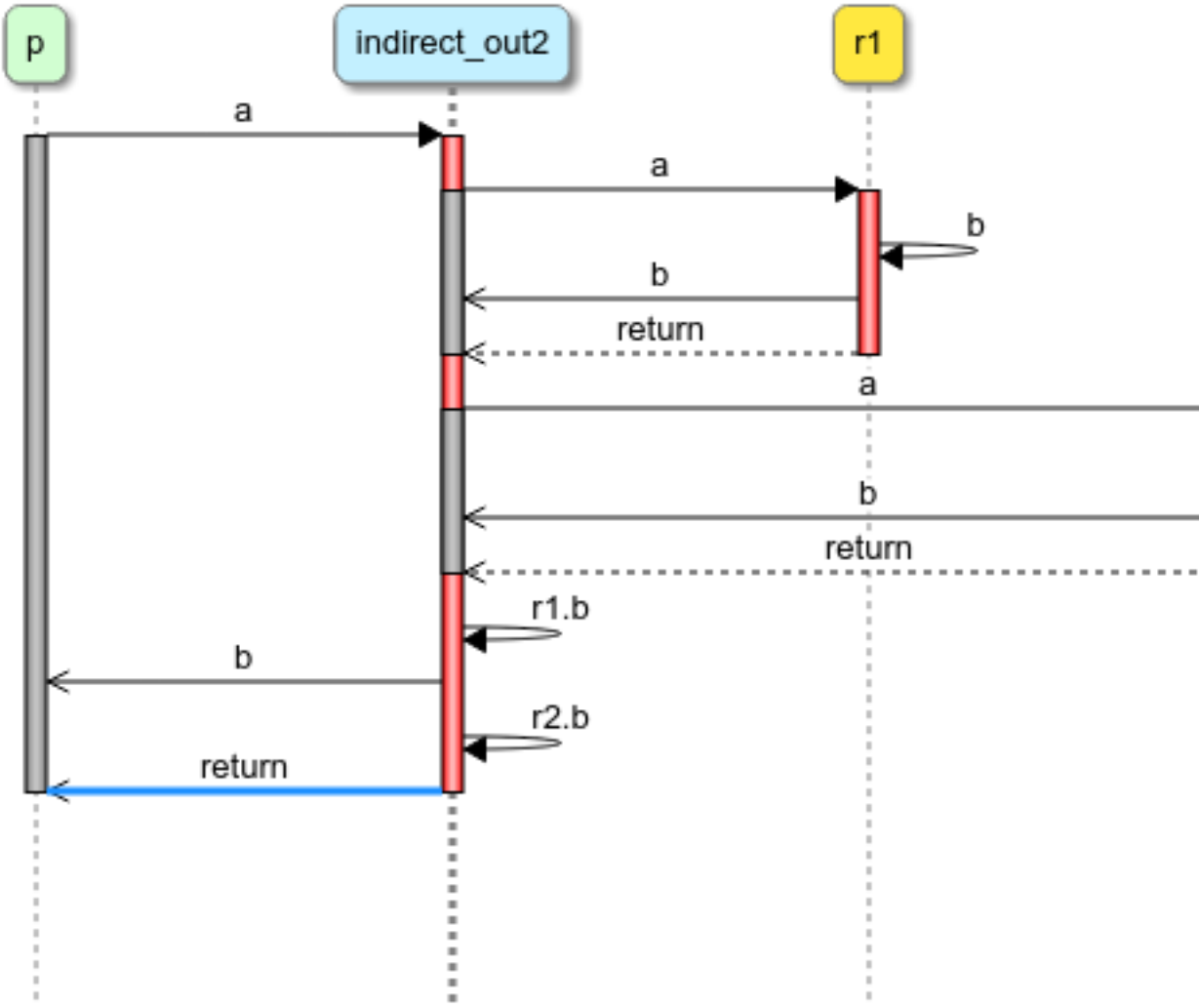
```
}
```




```
import indirect_multiple_out.dzn;

component indirect_multiple_out2
{
  provides I p;
  requires I r1;
  requires I r2;
  behaviour
  {
    on p.a(): {r1.a(); r2.a();}
    on r1.b(): p.b();
    on r2.b(): {}
```

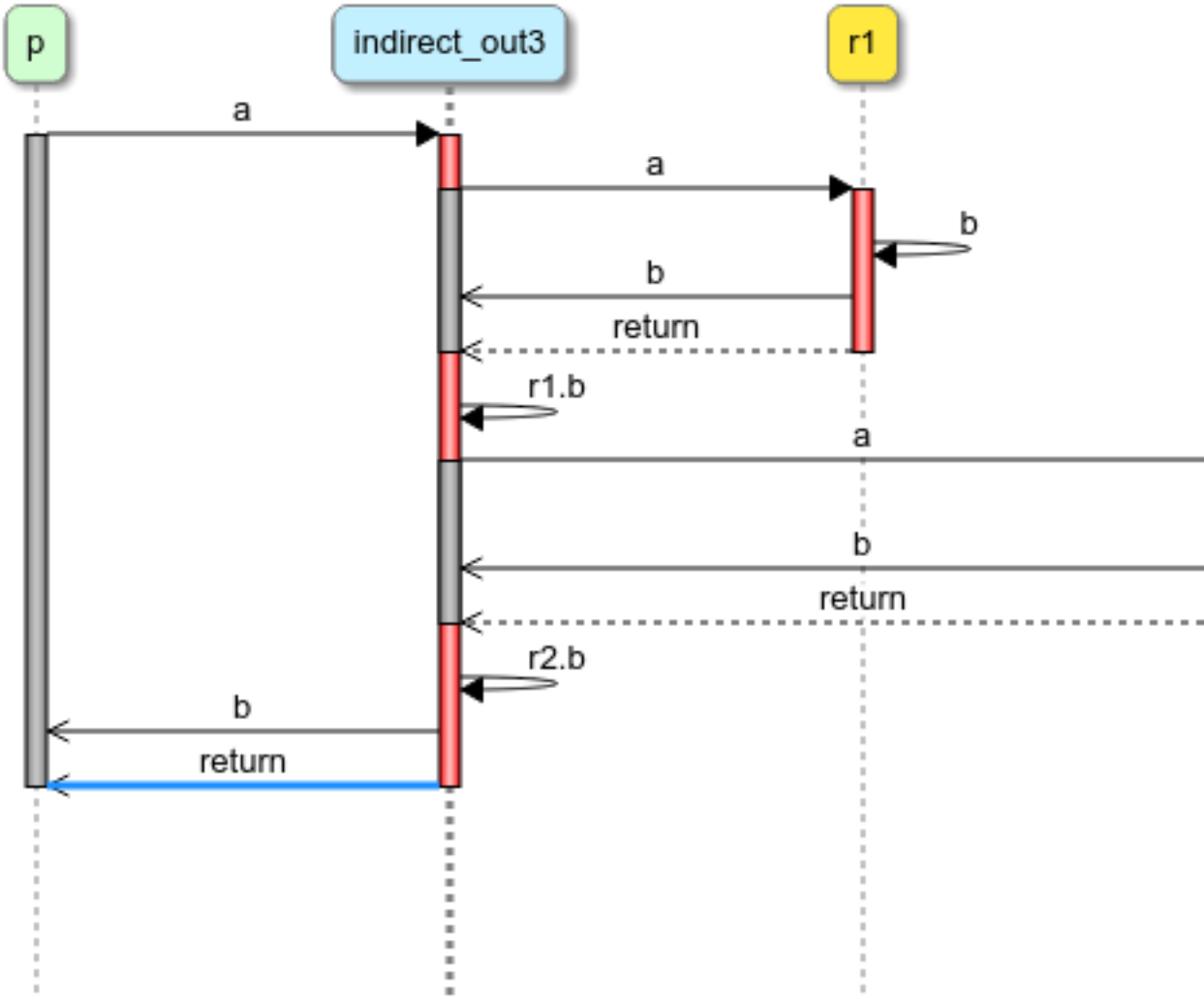
```
  }
}
```




```
import indirect_multiple_out.dzn;

component indirect_multiple_out3
{
  provides I p;
  requires I r1;
  requires I r2;
  behaviour
  {
    on p.a(): r1.a();
```

```
    on r1.b(): r2.a();
    on r2.b(): p.b();
  }
}
```



4.6 Indirect blocking out event

Also see Section 3.6.12 [Blocking], page 48.

The in-event on the provides port (p.a) blocks (does not return) until a reply is handled. This happens in the handling of the requires port out-event (r.b).

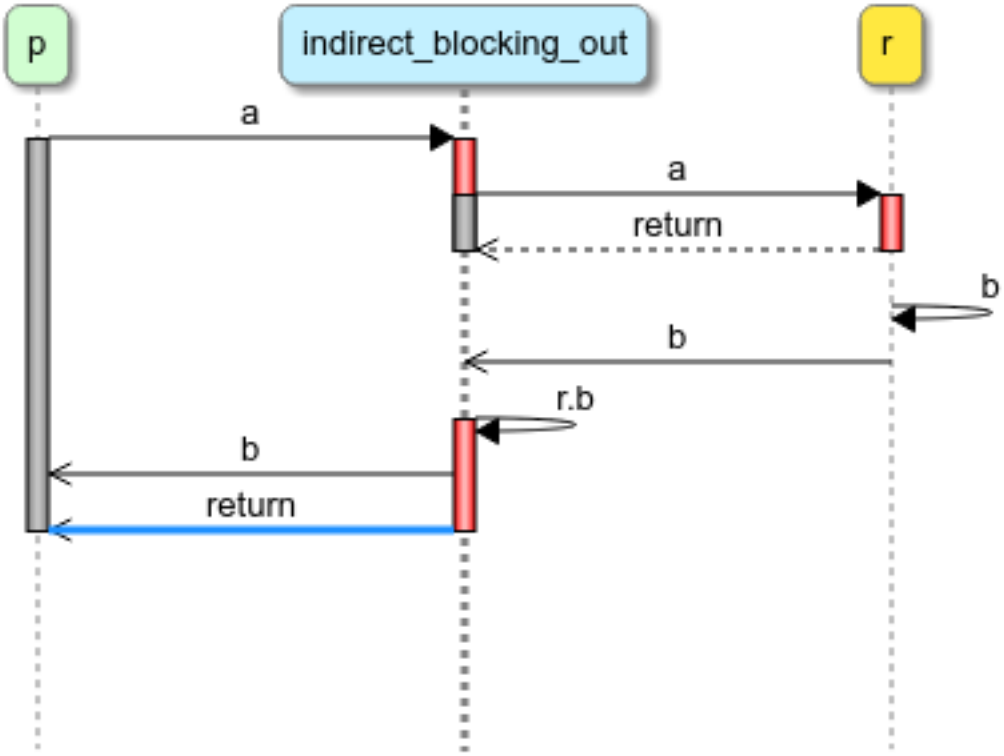
```
interface I
{
```

```
  in void a();
  out void b();
  behaviour
  {
    on a: b;
  }
}

interface I2
{
   in void a();
   out void b();
   behaviour
   {
     bool idle = true;
     [idle] on a: idle = false;
     [!idle] on a: illegal;
     [!idle] on inevitable: {idle = true; b;}
   }
}

component indirect_blocking_out
{
  provides I p;
  requires I2 r;
  behaviour
  {
    blocking on p.a(): r.a();
    on r.b(): {p.b(); p.reply();}
  }
```

```
}
```



If the keyword blocking in above example would be omitted it would lead to an erroneous situation since the provided in-event (p.a) would return before the provided out-event (p.b) would have been generated.

4.7 External multiple out events

Also see Section 3.3.4 [External], page 19.

The addition of external on a required interface removes the atomicity of an action list, i.e: {a; b;}.

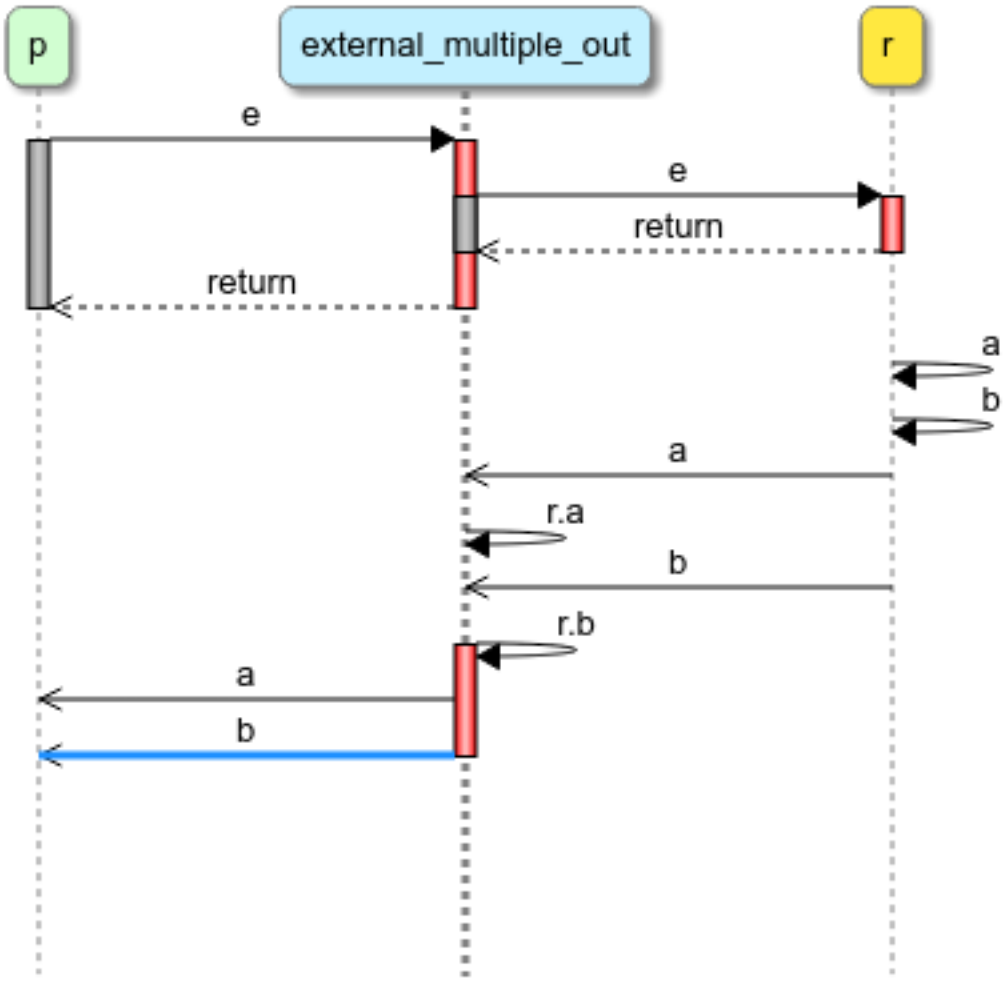
Note: the addition of the in-event e is required to constrain the occurrence of the inevitable event in combination with external.

```
interface I
{
  in void e();
  out void a();
  out void b();
  behaviour
  {
    bool idle = true;
```

```
    [idle] on e: idle = false;
    [!idle] on e: illegal;
    [!idle] on inevitable: {idle = true; a; b;}
  }
}

component external_multiple_out
{
  provides I p;
  requires external I r;
  behaviour
  {
    bool idle = true;
    [idle] on p.e(): {idle = false; r.e();}
    [!idle] on p.e: illegal;
    on r.a(): {}
    on r.b(): {idle = true; p.a(); p.b();}
  }
```

```
}
```



4.8 Indirect blocking multiple external out events

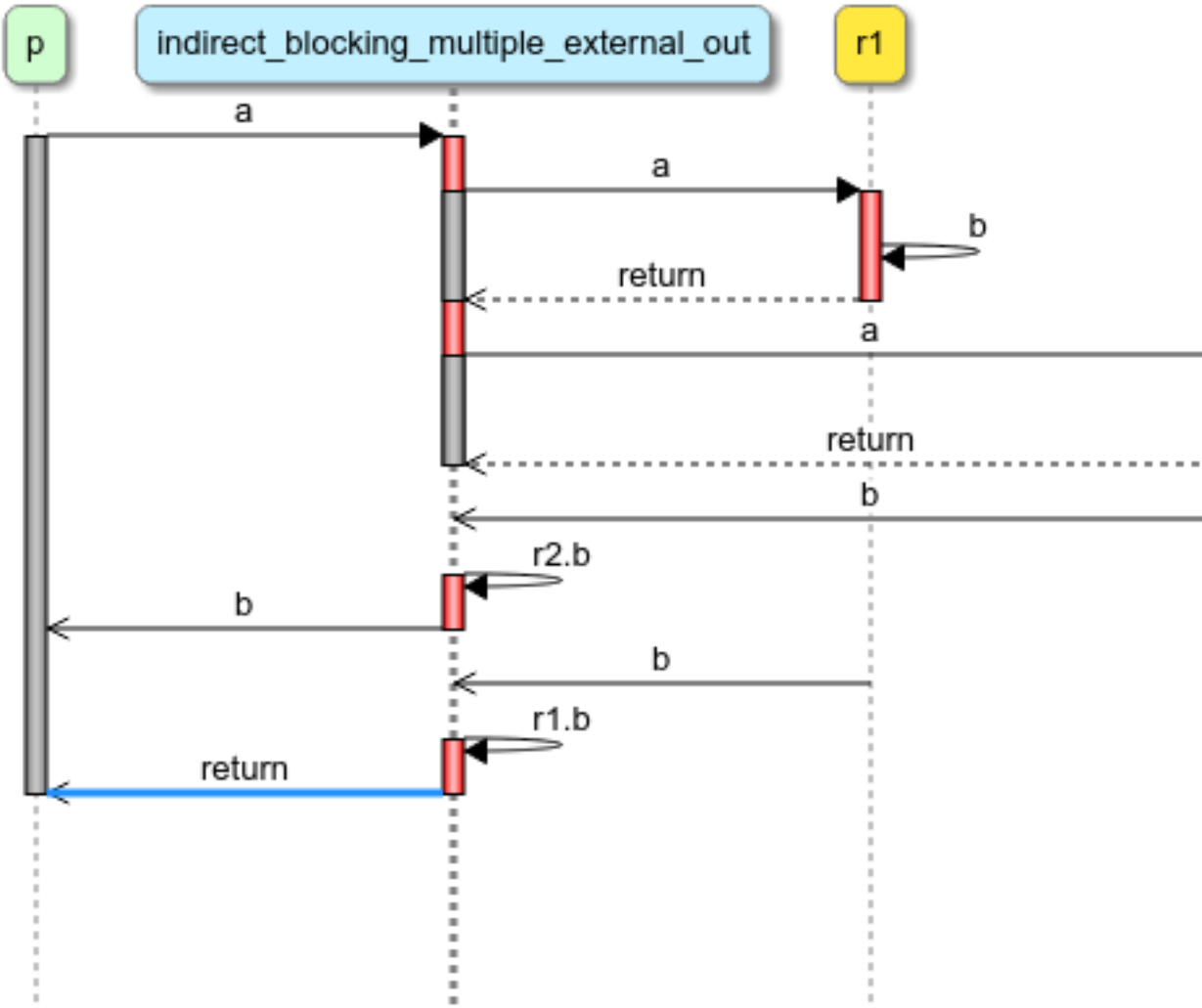
The 2 required out-events (r1.b, r2.b) can come in any order. The message sequence chart shows only one scenario. The implementation of the component is such that the provided behaviour is the same in both cases.

```
interface I
{
  in void a();
  out void b();
  behaviour
  {
    on a: b;
```

```
  }
}

component indirect_blocking_multiple_external_out
{
  provides I p;
  requires external I r1;
  requires external I r2;
  behaviour
  {
    bool ready = true;
    on p.a(): blocking {ready = false; r1.a(); r2.a();}
    [!ready] on r1.b(), r2.b(): {ready = true; p.b();}
    [ready] on r1.b(), r2.b(): p.reply();
  }
```

}


p
indirect_blocking_multiple_external_out
r1
a
a
b
return
a
return
b
r2.b
b
b
r1.b
return

5 Verifying Models

5.1 Introduction

Verification in Dezyne focuses on verifying properties that are hard for humans to verify. These properties mainly concern ordering of events, asynchronous behaviour, deadlock and/or livelock. Verifying a component together with its provided and required interfaces ensures that the component behaves correctly in its environment according to the specified behaviour.

5.2 What is verified

Components developed in Dezyne can be verified, after proven to be valid for verification

These are the checks that can be performed:

- *completeness*: guards in Dezyne enable and disable event clauses. It is a required that in every state of a model each event is enabled, either by being unguarded, or by having a guard that evaluates to "true" for the given state.
  Upon violation, The following error is reported :
  M is incomplete: e not handled
- *deterministic*: in Dezyne all components are required to be deterministic. The most common cause of non-determinism in a component is overlapping guards, i.e. two different set of actions for the same might occur in a specific situation.
  Upon violation, The following error is reported :
  Component M is non-deterministic due to overlapping guards
- *illegal*: it checks that there are no protocol violations between a component and its required interfaces.
  Upon violation, The following error is reported :
  illegal
- *range error*: it checks that each subint variable is always within its defined range.
  Upon violation, The following error is reported :
  integer range error in model M
- *type error*: when a non-void event is triggered, a value has to be replied. The type of that value has to match the return type of the event.
  Upon violation, The following error is reported :
  type error in model M
- *queue full*: a Dezyne model with provides ports has a queue where notification events are stored before they are processed. During verification it is checked that that this queue does not overflow, i.e. that it remains non-blocking. The queue size can be specified for verification with the "-q" option. The default size is 4.
  Upon violation, The following error is reported :
  queue full
- *deadlock*: a deadlock is a situation where none of the components in a system can make progress; nothing can happen and the system simply does not respond. This commonly occurs when two components each require an action from the other before they can perform any further action themselves. Another common cause is when a component

is waiting for some external event which fails to occur.
In general, deadlocks can be hard to find, because the entire system needs to be reviewed to discover them and freedom from deadlocks is a property of the system as a whole. For example, component A might be waiting for B which is waiting for C while C is waiting for A. Dezyne ensures that this never happens in the following way:
Each component by itself can be verified as being deadlock free; Components can only be composed in ways that have been proven not to cause deadlock. Note: Dezyne can only verify what it knows; therefore, e.g. handwritten code can still cause deadlocks.
Upon violation, The following error is reported :
deadlock in model M

- *compliance*: it checks that the component together with the required interfaces implements the behaviour: it checks that the component together with the required interfaces implements the behaviour specified in the provided interface(s).
  Upon violation, The following error is reported :
  Component M is non-compliant with interface of provided port
- *livelock*: a component is said to be livelocked when it is permanently busy with internal behaviour but ceases to serve its clients. For example, due to a design error such that the design is constantly interacting with its used components and starving the client; or due to the arrival rate of unconstrained external events such that processing them starves the client. As seen from the outside of a component, this appears very similar to deadlock. The difference is that a deadlocked component does nothing at all whereas a livelocked component might be performing lots of actions, but none of them are visible to the component's clients.
  Upon violation, The following error is reported :
  livelock in model M

The checks that are executed differ for interfaces and components:

- For interfaces:
  - completeness
  - deadlock
  - illegal
  - range error
  - type error
  - livelock
- For components:
  - completeness
  - deterministic
  - illegal
  - range error
  - type error
  - queue full
  - deadlock
  - compliance

- livelock

For interfaces, the illegal check, range error check, and type error check are reported as part of the deadlock check. For components, the range error check, the type error check, and queue full check are reported as part of the illegal check.

6 Code Integration

6.1 Integrating C++ Code

6.1.1 Purpose

This text briefly describes the C++ code that is generated from Dezyne models and the integration thereof.

6.1.2 Introduction

Every wellformed Dezyne model can be automatically converted into a corresponding well-formed C++ representation. A verified Dezyne model can be automatically converted into a corresponding C++ representation which when executed exhibits the same behaviour as one can observe in the Dezyne simulation and verification of said model.

In Dezyne there are three model types: interface, component and system.

In this chapter we cover the code which is generated from these models as well as the way the generated code might be integrated.

6.1.3 Interfaces

Dezyne turns an interface such as:

```
interface MyInterface
{
  in void in_event();
  out void out_event();
  behaviour
  {
    on in_event: out_event;
  }
}
```

into the following C++ class representation:

```
struct MyInterface
{
  struct
  {
    dezyne::function<void()> in_event;
  } in;
  struct
  {
    dezyne::function<void()> out_event;
  } out;
};
```

Each event in an interface is a slot to which a value of something with the appropriate callable signature can be assigned. A callable value in C++ is either: a function pointer or a functor. For example:

```
void foo(){}
```

```
MyInterface port;
port.out.out_event = foo;
port.in.in_event = port.out.out_event;
```

Note that the last statement above short circuits the in_event to the out_event as is described in the Dezyne interface model.

6.1.4 Components

One could consider a component to be no more than the connecting part between all of its ports. For example:

```
component MyComponent
{
  provides MyInterface provided_port;
  requires MyInterface required_port;
}
```

in which case a simplified C++ representation might look like this:

```
struct MyComponent
{
  MyInterface provided_port;
  MyInterface required_port;
  MyComponent()
  : provided_port()
  , required_port()
  {
    provided_port.in.in_event = dezyne::ref(required_port.in.in_event);
    required_port.out.out_event = dezyne::ref(provided_port.out.out_event);
  }
};
```

Note that dezyne::ref allows shot circuiting events which will be initialised at a later stage.

Of course for all practical purposes one would expect a component to be more complicated to be able to meet all of its interface contracts.

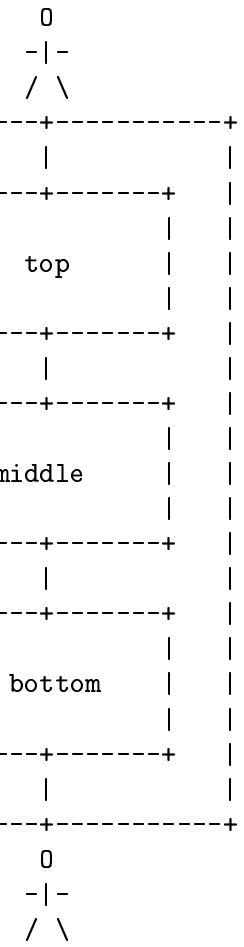
6.1.5 Systems

Along the same lines a Dezyne system may aggregate other components and systems and bind them together by their ports. For example:

```
component MySystem
{
  provides MyInterface provided_port;
  requires MyInterface required_port;
  system
  {
    MyComponent top;
    MyComponent middle;
    MyComponent bottom;
    provided_port <=> top.provided_port;
```

```
    top.required_port <=> middle.provided_port;
    middle.required_port <=> bottom.provided_port;
    bottom.required_port <=> required_port;
  }
}
```

or depicted in a diagram:



6.1.6 Integration

Constructing such a system using Dezyne is straightforward. Every model can be automatically converted into code and by the hierarchical nature of Dezyne all components and systems slot together automatically, however two facilities are required to allow this: the dezyne runtime and the dezyne locator. Both are provided by Dezyne.

In C++ the main function for this system might look like this:

```
#include "MySystem.hh"
#include "dezyne/runtime.hh"
#include "dezyne/locator.hh"
int main()
{
  dezyne::locator loc;
```

```
  dezyne::runtime rt;
  loc.set(rt);
  //construct the system
  MySystem system(loc);
  //connect the outer events directed at the system
  system.provided_port.out.event = []{
    std::cout << "system.provided_port.out.event" << std::endl;
  };
  system.required_port.in.event  = []{
    std::cout << "system.required_port.in.event" << std::endl;
  };
  //and finally fire some of the external events
  system.provided_port.in.event();
  system.required_port.out.event();
}
```

Runtime: The runtime takes care of decoupling the events between the caller and the callee when this is required.

Locator: The locator allows injecting the implementation behind a port deep into the system from the outside.

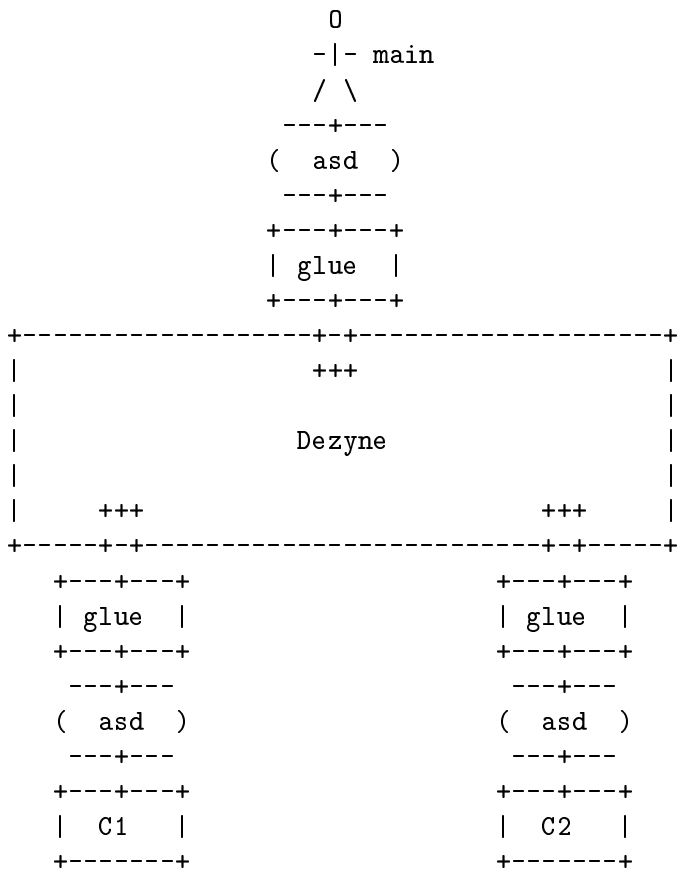
In the example you can see that the locator facility is also responsible for passing an instance of the runtime into the system. Injection example:

```
interface Foo
{
  in void bar();
  behaviour
  {
    on bar:{}
  }
}
component MyComponent2
{
  provides MyComponent2 provided_port;
  requires injected Foo required_port;
  behaviour { /* ...  */ }
}
int main()
{
  dezyne::locator loc;
  dezyne::runtime rt;
  loc.set(rt);
  Foo foo;
  foo.in.bar = []{/*no op*/};
  loc.set(foo);
  MyComponent comp(loc);
  comp.provided_port.in.in_event();
}
```

6.1.7 Migrating from ASD to Dezyne

When converting ASD models to Dezyne models a map file is generated. This allows automatic generation of a glue layer between the Dezyne generated code and any existing code behind an ASD interface. This glue layer comes in two flavours, which depend on whether the interface is provided or required.

Let us assume we are migrating a system that has a main function which instantiates and connects the ASD tree of components, which we want to preserve. And where there are two handwritten components C1 and C2 which have to be preserved as is. We would like to end up with a system as depicted below.



Here the ellipses represent the ASD interfaces. For every interface a glue layer is generated to connect the Dezyne system to the ASD interface which the existing code depends on.

Effectively this approach makes an ASD component appear like a Dezyne component and it makes a Dezyne component appear like an ASD component. The former case is aimed at reusing existing ASD components from Dezyne. The latter case is aimed at reusing a main function. In either case this approach would allow migrating an ASD system component by component to Dezyne.

6.2 Thread-safe Shell

Note: this feature was introduced in Dezyne release 2.0.0.

A Dezyne Thread-safe Shell guarantees safe use of a Dezyne system component in a multi-threaded environment. It also enables the use of the 'blocking' keyword.

6.2.1 Shell Syntax

Use the dzn command-line client to generate code and a thread-safe shell:

```
dzn code -l c++ -s SYSTEM FILE    (1)(Since 2.0.0)
```

Explanation:

1) Generates code for all components and interfaces referred to in the SYSTEM component. In addition a thread-safe shell is generated for SYSTEM.

6.2.2 Semantics

A thread-safe shell wraps a Dezyne system component. In addition to an instance of the Dezyne component it contains a thread and an event queue. External code can call event functions on system ports. The thread-safe shell defers each external call by posting a function object in the event queue. A thread private to the thread-safe shell takes deferred functions from the queue and executes them one by one. Thus, external calls are serviced in the order of arrival.

An external call of a provided port in event blocks until the thread-safe shell private thread has completed the deferred function call. The external call blocks until a reply statement has been executed for the input event port. A subsequent call on a blocked port will block until the prior call returns.

An external call of a required port out event returns a soon as the event call is scheduled. The external call return is not synchronized with the actual execution of the event by a thread-safe shell private thread.

6.2.3 Shell Example

Generating c++ code with a thread-safe shell for component SYS results in files: SYS.hh, SYS.cc, BHV.hh and IA.hh.

A call of SYS::pp.in.iv() captures input parameters by value to prevent data races. The call schedules a call to SYS::bhv.pp.in.iv() and blocks the calling thread until the scheduled call returns.

A call of SYS::rp.out.o() captures input parameters by value to prevent data races. The call schedules a call to SYS::bhv.rp.out.o() and returns immediately.

```
component SYS {
  provides IA pp;
  requires IA rp;
  system {
    BHV bhv;
    pp <=> bhv.pp;
    bhv.rp <=> rp;
  }
}
component BHV {
  provides IA pp;
  requires IA rp;
```

```
}
extern int $int$;
interface IA {
  in void iv(int i);
  out void o(int i);
  behaviour {
    on iv: {}
    on optional: o;
  }
}
```

File SYS.hh:

```
#ifndef SYS_HH
#define SYS_HH
#include #include #include #include #include "BHV.hh"
#include "IA.hh"
#include "IA.hh"
namespace dzn {struct locator;}
struct SYS
{
  dzn::meta dzn_meta;
  dzn::runtime dzn_runtime;
  dzn::locator dzn_locator;
  BHV bhv;
  IA pp;
  IA rp;
  dzn::pump dzn_pump;
  SYS(const dzn::locator&);
  void check_bindings() const;
  void dump_tree(std::ostream& os=std::clog) const;
};
#endif // SYS_HH
```

File SYS.cc:

```
#include "SYS.hh"
SYS::SYS(const dzn::locator& locator)
: dzn_meta{"","SYS",0,{additionalbhv.dzn_meta},{}}
, dzn_locator(locator.clone().set(dzn_runtime).set(dzn_pump))
, bhv(dzn_locator)
, pp(bhv.pp)
, rp(bhv.rp)
, dzn_pump()
{
  pp.in.iv = [&] (int i) {
    return dzn::shell(dzn_pump, [&,i] {return bhv.pp.in.iv(i);});
  };
  rp.out.o = [&] (int i) {
    return dzn_pump([&,i] {return bhv.rp.out.o(i);});
```

```
  };
  bhv.pp.out.o = std::ref(pp.out.o);
  bhv.rp.in.iv = std::ref(rp.in.iv);
  bhv.dzn_meta.parent = additionaldzn_meta;
  bhv.dzn_meta.name = "bhv";
}
void SYS::check_bindings() const
{
  dzn::check_bindings(additionaldzn_meta);
}
void SYS::dump_tree(std::ostream& os) const
{
  dzn::dump_tree(os, additionaldzn_meta);
}
```

See also:

- Section 3.6.12 [Blocking], page 48,
- Chapter 7 [The Dezyne command-line tools], page 82,

7 The Dezyne command-line tools

7.1 Invoking `dzn`

The `dzn` command is a front-end to Dezyne functions.

```
dzn dzn-option... command command-option... FILE...
```

Invoking `dzn` without `command` shows a list of available commands.

The *dzn-option*s can be among the following:

`--debug`
`-d` Enable debug ouput.

`--help`
`-h` Display help on invoking `dzn`, and then exit.

`--json`
`-j` Output json

`--peg`
`-p` Use plain *peg* parser, skip Well-formedness checking.

`--verbose`
`-v` Be more verbose, show progress.

`--version`
`-V` Display the current version of `dzn`, and then exit.

7.2 Invoking `dzn code`

The `dzn code` command generates C++ code for a Dezyne model file. See Chapter 6 [Code Integration], page 74.

```
dzn dzn-option... code option... FILE
```

The *option*s can be among the following:

`--calling-context=`*`TYPE`*
`-c` *`TYPE`* Generate extra parameter of *type* for every event.

`--glue=`*`TYPE`*
`-g` *`TYPE`* Generate glue for *TYPE* var.

`--import=`*`DIR`*
`-I` *`DIR`* Add directory *DIR* to import path.

`--help`
`-h` Display help on invoking `dzn code`, and then exit.

`--language=`*`LANG`*
`-l` *`LANG`* Generate code for language *LANG* var.

`--model=`*`MODEL`*
`-m` *`MODEL`* Generate main for *MODEL*.

`--output=`*DIR*
`-o` *DIR* Write output to directory *DIR* (use `-` for stdout).

`--shell=`*MODEL*
`-s` *MODEL* Generate thread-safe system shell for *MODEL*. See Section 6.2 [Thread-safe Shell], page 78.

7.3 Invoking `dzn verify`

The `dzn verify` command checks a Dezyne file for models with verification problems. See Chapter 5 [Verifying Models], page 71.

```
dzn dzn-option... verify option... FILE
```

The *option*s can be among the following:

`--all`
`-a` Show all errors, i.e., keep going after finding an error. By default, verification stops after finding a verification error.

XXX: –all is more like make's –keep-going.

`--help`
`-h` Display help on invoking `dzn verify`, and then exit.

`--model=`*MODEL*
`-m` *MODEL* Limit verification to *MODEL*, and for behavioural component model, to its interfaces.

XXX: Verification cannot be limited to system components models; verifying a system model is a no-op. XXX

`--queue_size=`*SIZE*
`-q` *SIZE* Use queue size *SIZE* for verification, the default is `3`.

XXX: the underscore in –queue_size is weirdXXX

8 Dezyne Syntax

8.1 Identifiers

In Dezyne identifiers are used to denote objects like interfaces, components, events, user defined types, variables, etc. A keyword cannot be used as an identifier and identifiers are case-sensitive.

An identifier starts with a letter, which can be followed by further letters, digits, or underscores. Following are legal identifiers:

Alarm, turn_on, VAL_123

8.1.1 Scoping

In order to properly refer to types and objects defined in a context different from the current one, that context has to be included in the name of such a type or object. This is done using a 'dot notation', like in `MyInterface.MyType`. Possible dot constructs are:

- `MyEnumType.MyEnumValue` denotes an enumeration value, where `MyEnumType` denotes its enumeration type defined in the current context.
- `SomeInterface.MyEnumType` denotes an enumeration type defined in the context of interface `SomeInterface`.
- `SomeInterface.MyEnumType.MyEnumValue` denotes an enumeration value, where `SomeInterface.MyEnumType` denotes its enumeration type defined in the context of interface `SomeInterface`.
- `myPort.myEvent` denotes an event defined in the context of some interface `MyInterface` of which port `myPort` is an instance.
- `MyInstance.myPort` denotes a port defined in the context of some component `MyComponent` of which `MyInstance` is an instance.
- `MyNamespace.MyName` denotes an object named `MyName`, such as a component, interface, enumeration value defined in a namespace `MyNamespace`. Since namespaces can be nested, long names can be constructed, like `MyNamespace1.MyNamespace2.MyNamespace2.MyInterface.MyEnumType.MyEnumValue`.

Where appropriate in the sections below, examples will be given to illustrate this notation. The following list shows the keywords in Dezyne. These keywords may not be used as constant or variable or any other identifier names.

```
behaviour     blocking      bool          component
else          extern        external      enum
false         if            illegal       import
inevitable    injected      inout         interface
in            namespace     on            optional
otherwise     out           provides      reply
requires      return        subint        system
true
```

Dezyne supports single-line and multi-line comments very similar to C++. Multi-line comments may be nested. All characters available inside any comment are ignored by the Dezyne compiler.

```
component Alarm {
  /* This is an example of multi-line comment.
   * The line below is ignored also:
   * interface IA
   */
  provides IAlarm ia; // a single-line comment
}
```

8.2 Dezyne Files

Dezyne components and interfaces are organized in files. A file, with extension '.dzn', contains a number of components and interfaces.

Components can refer to interfaces and other components. An 'import' clause is needed when the referred information is defined in another file (see below).

The order of specification in one file is as follows:

- All necessary imports.
- All interface and component definitions in arbitrary order.

8.2.1 Import

An `import` clause makes available all types, interfaces and components that are defined in another file. From an imported interface or component the 'public' parts are available, i.e. all information but the interface or component behaviour, or the component system details. Syntax:

```
import ModelName.dzn;
```

where `ModelName.dzn` is a Dezyne file.

An imported file may contain imports itself, which are 'expanded' also. When a file occurs twice in the resulting set of imports, it is expanded only once. This prevents the introduction of duplicate definitions. Also recursive imports are handled correctly.

8.3 Types and Expressions

In Dezyne all variables and constants are typed. A number of type constructs are available.

8.3.1 Bool

`bool` is pre-defined and denotes the boolean type, with constants `true` and `false`.

Available boolean operators are:

- Logical negation, denoted by `! b1`.
- Logical and, denoted by `b1 && b2`.
- Logical or, denoted by `b1 || b2`.
- Equality of two boolean expressions, denoted by `b1 == b2`.
- Inequality of two boolean expressions, denoted by `b1 != b2`.

where `b1` and `b2` denote booleans.

8.3.2 Enum

An interface or component can specify a user defined enumerated type. Such a type has a name and a collection of values. An example:

```
enum MyEnumType { MyValue1, MyValue2, MyValue3 };
```

where `enum` is a keyword; this defines the enum type `MyEnumType` with three values.

In expressions the enum values are referred to with a dot notation: `MyEnumType.MyValue2` . Available enum operators are:

- Equality of two enum expressions, denoted by `e1 == e2`.
- Inequality of two enum expressions, denoted by `e1 != e2`.
- Testing the value of an enum variable, denoted by `v.MyValue2`, which is shorthand for `v == MyEnumType.MyValue2`

where `e1` and `e2` denote enum expressions, and `v` an enum variable.

8.3.3 Subint

The integer type is available in Dezyne in a restricted way: only a finite subset of integers can be used. An explicit type definition is needed for such a subset, where a C++-like syntax is used. An example:

```
subint MyIntType {2..5};
```

where `subint` is a keyword. This defines the finite type `MyIntType`with possible values 2, 3, 4, and 5. Available integer operators are:

- A number of integer comparisons, denoted by:
  - `i1`
  - `i1 <= i2`
  - `i1 >= i2`
  - `i1 > i2`
  - `i1 == i2`
  - `i1 != i2`
- Integer addition, denoted by `i1 + i2`.
- Integer subtraction, denoted by `i1 - i2`.

where `i1` and `i2` denote integers.

8.3.4 Data Types

Apart from the `bool`, `enum`, and `int` types introduced above, which play an important role in the description of interface and component behaviour, so-called 'data types' can be defined. These play a restricted role and are used as types of 'data parameters' in events (see below). A data type is defined as follows:

```
extern MyDataType $someExternalTypeExpresssion$;
```

where `extern` is a keyword, and in `$someExternalTypeExpresssion$`the dollar signs is a type expression in the target language (C++, java, ...).

No Dezyne supported expressions are available for data types, apart from 'dollar expressions': `$some expression$` will be passed as-is.

8.4 Interface Models

An interface has a name, can define some local types, defines a collection of events, and a behaviour describing the protocol on its events. A syntax example:

```
interface MyInterface
{
  enum MyEnumType { Val1, Val2, Val3 };
  extern MyString $std::string$;
  in MyEnumType myValuedEvent(MyString s);
  out void myReply();
  behaviour
  {
    on myValuedEvent(s) : { reply MyEnumType.Val3; }
  }
}
```

In this example a number of keywords are introduced: `interface`, `in`, `out`, `void`, `behaviour`, `on`, and `reply`.

The example defines an interface named `MyInterface` with two local types, two events, and the interface behaviour (see below for a syntax description of events and behaviour).

8.4.1 Events

An event has (in this order):

- a direction (`in` or `out`),
- optionally a reply type (`void` if it is absent),
- a user defined name,
- a data parameter list, which may be empty.

Some examples:

```
in void e2();            // a void in event called 'e2' with an empty parameter list
in MyEnumType e3();      // a valued in event called 'e3'
out void e4(MyString s); // a void out event called 'e4' with one data parameter
```

There is a restriction on event declarations: an `out` event must be `void`.

8.4.2 Behaviour

The `behaviour` section of an interface contains the protocol description of the interface. See Section 8.6 [Interface and Component Behaviour], page 88, for an elaborate description.

8.5 Component Models

A component has a name, a collection of ports by which it communicates with the outside world, and optionally a description of its internals. A syntax example:

```
component MyComponent
{
  provides MyInterface1 myPort1;
  requires MyInterface2 myPort2;
  requires MyInterface3 myPort3;
```

```
  /*
   * internals description....
   */
}
```

In this example a number of keywords are introduced: `component`, `provides`, and `requires`.

The example defines a component named `MyComponent` with three ports.

8.5.1 Ports

A component provides some functionality, and requires functionality of other components. Communication between components is performed through their ports, which are instances of interfaces. Each port has a direction according to its intention (`provides` or `requires`), the interface it implements and a local name. A component refers to these ports in its behaviour.

8.5.1.1 Injection

A `requires` port can be specified to be `injected`:

```
requires injected MyInterface myPort;
```

The intention is, that binding such a port will occur at a high level in the system hierarchy to some component port `someComponent.somePort`, and that more than one injected port can be bound to that port. For that reason `MyInterface` cannot define `out` events.

See Section 8.7 [System Components], page 93, for a detailed description of the binding of injected ports.

8.5.2 Internal Description

The component's internal description comes in three flavors:

- Behavioural: Like in the interface protocol the internals are given as a behavioural description, in which for each event the required actions are given. See Section 8.6 [Interface and Component Behaviour], page 88.
- Compositional: The functionality is described as the composition of a number of sub components. See Section 8.7 [System Components], page 93.
- Hand-written: The internals are missing indicating the intention to have a hand-written implementation of the component. This implementation must be such that it satisfies the specified ports. The component is a place-holder only.

8.6 Interface and Component Behaviour

A behaviour description might look like this:

```
behaviour
{
  enum MyEnumType { Value1, Value2 };
  MyEnumType myEnumVariable = MyEnumType.Value1;
  bool myBoolVariable = true;
  on myPort1.myEvent1():
  {
```

```
      bool blocal = myBoolVariable;
      myBoolVariable = false;
      myPort2.myEvent2();
      f(blocal);
    }
    [myEnumVariable.Value2]
    {
      on myPort1.myEvent2(): illegal;
    }
    void f(bool b)
    {
      if (b) myPort2.myEvent5();
      else   { myPort2.myEvent5(); myBoolVariable = true; }
    }
  }
```

This behaviour section introduces a local enum type (`MyEnumType`), two variables (called `myEnumVariable` and `myBoolVariable`), a number of statements, and a function definition (`void f(bool b)`) with one parameter. Note that in a sub scope a local variable `blocal` is defined. The different constructs are handled in more detail below.

8.6.1 Variable Declarations

As demonstrated in the example above, variables can be defined in a behaviour description, both at top level and in a local scope. Variables must be given an initial value.

The top-level variables are often referred to as 'state variables'. Their scope extends to the complete behaviour description, including the body of functions defined in the behaviour.

Variables defined at other places have a scope which is restricted to the compound statement (`{ .... }`) they are defined in.

8.6.1.1 State Variables

8.6.2 Declarative versus Imperative Statements

There are two 'kinds' of behavioural statements: 'declarative' and 'imperative'. The declarative part describes the trigger events and the conditions in which they have to be handled, while the imperative part describes the response that has to occur as result of a trigger.

8.6.3 Compound Statement

Statements of the same 'kind' can be grouped using curly braces:

```
{ Statement1; Statement2; Statement3; }
```

It results in a single 'compound statement', of the same 'kind' as its sub statements, and can be used as such. Note that the list of statements within the curly braces can be empty:

```
{ /* this is an empty statement sequence */ }
```

8.6.4 Declarative Statements

The triggers an interface or component has to handle and the conditions in which they occur are described as declarative statements. The conditions are expressed as 'guard statements', the triggers as 'on-event statements'.

8.6.4.1 Guard Statement

A guard statement looks like:

```
[ myBooleanExpression ] myStatement
```

The square brackets (`[` and `]`) must contain a boolean expression expressing the validity of the guard. The statement following the guard can be either declarative or imperative.

8.6.4.2 On-event Statement

In an interface behaviour specification an on-event statement is expressed as:

```
on myEvent: myStatement
```

The statement contains a keyword `on`, the name of an event introduced in the interface, a colon (`:`), and a statement, which can be either be declarative or imperative. A declarative statement must not contain further on-event statements: nesting these statements is not allowed.

Note that no event parameters must be specified in an interface.

In a component behaviour specification an on-event statement is expressed as:

```
on myPort.myEvent(myParameter1): myStatement
```

where `myPort` is a port of the component with an interface type that contains event `myEvent` with one data parameter. Note that in a component behaviour specification data parameters have to be used where specified. On the trigger each data parameter must be a fresh variable, implicitly typed conform the event specification.

8.6.5 Imperative Statements

The response on a (guarded) trigger event are expressed in the imperative statement following the declarative part. All variable assignments, event actions, function calls, etc. are handled here.

8.6.5.1 Variable Declaration Statement

New variables can be declared as follows:

```
myType myVariable = myInitialValueExpression;
```

where `myType` denotes a previously defined type, `myVariable` is a 'fresh' name, and `myInitialValueExpression` an expression of type `myType`.

After the variable has been declared it can be referred to as long as the referring place is in scope (scopes being introduced by compounds). 'Fresh' should be interpreted as follows:

- No two variables with the same name can be declared in one scope.
- A variable `myVar` that has been declared in a wider scope can be 'shadowed' by a variable declaration with the same name in a sub scope

8.6.5.2 Assignment Statement

A previously declared variable can change value using an assignment statement:

```
myVariable = myValueExpression;
```

8.6.5.3 Action Statement

In handling the response on a trigger event, an event action can be called. The syntax in an interface declaration is:

```
myEvent;
```

where `myEvent` is the name of an out event introduced in the interface. Note that no event parameters must be specified.

In a component behaviour specification an action statement is expressed as:

```
myPort.myEvent(myParameterExpression);
```

where `myPort` is a port of the component with an interface type that contains event `myEvent` with one data parameter.

Note that the event in an action statement must be a void event. For a valued event the return value must be caught, so an assignment statement is the appropriate way to handle that:

```
myVariable = myEvent;
```

or

```
myVariable = myPort.myEvent(myParameterExpression);
```

in an interface or component behaviour respectively.

8.6.5.4 Function Call Statement

A function defined in a behavioral specification can be called with properly typed parameter expressions. Given a function declaration

```
void myFunc(MyEnumType e, bool b) { /* ...  */ }
```

a function call statement might look like:

```
myFunc(myEnumExpression, myBooleanExpression);
```

Note that the function in a function call statement must be a void function. For a valued function the return value must be caught, so an assignment statement is the appropriate way to handle that:

```
myVariable = myValuedFunc(myBooleanExpression);
```

8.6.5.5 Reply Statement

A valued trigger event will have to set the appropriate return value in its response handling. Setting that value (not returning yet) is denoted as:

```
reply(myValueExpression);
```

8.6.5.6 Return Statement

A valued function has to return with a value of the appropriate type. This looks like:

```
return myValuedExpression;
```

8.6.5.7 If Statement

Conditional handling of statements is supported by the if statement, which can have an optional else part:

```
if (myBooleanExpression) myStatement1
```

or:

```
if (myBooleanExpression) myStatement1 else myStatement2
```

where `if` and `else` are keywords, and the parentheses are part of the construct. The expression between the parentheses must be a boolean expression. The statements `myStatement1` and `myStatement2`are arbitrary imperative statements.

Note that nested if statements are allowed, and that

```
if (b1) if (b2) myStatement1 else myStatement2
```

will be interpreted as

```
if (b1)
{ if (b2) myStatement1 else myStatement2 }
```

and not as

```
if (b1)
{ if (b2) myStatement1 }
else myStatement2
```

In other words: `else` binds to the closest if.

8.6.6 Functions

A function declaration introduces the signature of a function and its body. The signature is built up from the return type, the name of the function, and a list of typed parameters. The body is a compound statement in the imperative part of the language, so no on-event and guarded statements are allowed. It is allowed however to refer to action events and global state variables, and of course to the function parameters.

An example:

```
MyReturnType myFunc(bool myPar1, MyEnum myPar2)
{
  bool myLocalBool = myPar2;
  if (myPar1)
  {
    myPort.myAction();
    myLocalBool = false;
  }
  myGlobal = myPar2;
  myOtherFunc(myLocalBool);
  return myRetuenValue;
}
```

This declares a function called `myFunc` with two parameters (of type `bool` and `MyEnum` respectively) and returning a value of type `MyReturnType`. In the function body a local variable is declared, a conditional action event is called, a global state variable is set, another function is called, and the return value is set.

Functions are allowed to be recursive, and also mutual recursive (function `f`calling function `g` and vice versa), under one condition: A recursive function must be "tail recursive"; this means that the recursive call must be the last statement in the function. The rationale for this restriction comes from the need for verification: the generic recursive case cannot be verified.

8.7 System Components

A component which is composed from a number of sub components has a 'system' internal description. This description specifies the available sub component instances, and all connections between the component ports. An example: suppose the following components to be defined:

```
component Foo {
  provides IFoo foo;
  requires IBar bar;
  requires INut nut;
  //  ...  internals ...
}
component Bar {
  provides IBar b;
  // ...  internals ...
}
component Nut {
  provides INut nt;
  requires IWell wl;
  // ...  internals ...
}
```

Then a system component `MySys` could be defined:

```
component MySys {
  provides IFoo top;
  requires IWell bot;
  // ...  internals ...
}
```

where its internals are composed of the `Foo`, `Bar`, and `Nut` components:

```
component MySys {
  provides IFoo top;
  requires IWell bot;
  system {
    Foo cfoo;
    Bar cbar;
    Nut cnut;
    top <=> cfoo.foo;
    cfoo.bar <=> cbar.b;
    cfoo.nut <=> cnut.nt;
    cnut.wl <=> bot;
  }
```

```
}
```

The system description shows the instantiation of the three sub components, and four connections between various ports.

8.7.1 Component Instances

In a system description a sub component is specified by its type and local name:

```
MyComponent myInstance;
```

The definition of MyComponent has to be available, potentially through an 'import'.

It is allowed to have more than one instance of the same type:

```
MyComponent myInstance1;
MyComponent myInstance2;
```

8.7.2 Binding

Communication between components is achieved through component ports. Communication is specified as binding:

```
MyComp1.port1 <=> MyComp2.port2;
```

indicates components `MyComp1` and `MyComp2`communicate through their respective ports `port1`and `port2`. This is often expressed as: `port1` is bound to `port2`.

Binding is symmetric: `MyComp1.port1 <=> MyComp2.port2`and `MyComp2.port2 <=> MyComp1.port1` have identical meaning.

Communication is restricted to ports of the same (interface) type. Moreover communication 'direction' has to be preserved. There are two cases:

- Two sub components communicating: always a `provides` port binds to a `requires` port, like in `cfoo.bar <=> cbar.b`in the example above.
- Port forwarding (an external port is forwarded to a sub-component port): always the directions have to be the same, like in `top <=> cfoo.foo`and `cnut.wl <=> bot` in the example above.

8.7.2.1 Injection

Binding of `injected` ports is done at a higher system level (see Section 8.5 [Component Models], page 87). A wild-card character (`*`) is used to achieve the binding of one `provides` port to all `injected requires` ports.

Let's take a typical example involving logging:

```
interface ILog {
  ...
}
component Log {
  provides ILog log;
  ...
}
```

Suppose a lot of components require logging:

```
...
component MyComponent12 {
```

```
  provides MyInterface12 intf;
  requires injected ILOg l;
  ...
}
component MyComponent13 {
  provides MyInterface13 intf;
  requires injected ILOg l;
  ...
}
...
```

then some system component can bind all logging in one go:

```
component MySystem {
  ...
  system {
    Log clog;
    ...
    MyComponent12 c12;
    MyComponent13 c13;
    ...
    clog.log <=> *;
  }
}
```

It is allowed to group some components in a sub system:

```
component MySubSystem {
  ...
  system {
    ...
    MyComponent12 c12;
    MyComponent13 c13;
    ...
  }
}
```

and do the wild-card binding for that sub system only:

```
component MySystem {
  ...
  system {
    Log clog;
    MySubSystem subsys;
    ...
    clog.log <=> subsys.*;
  }
}
```

8.8 Namespaces

All component, interface, and type definitions can be defined in a *namespace*, which provides name scoping. The scope can be used when referring to such a definition.

8.8.1 Namespace Syntax

A namespace has a name, and within its scope components, interfaces, types, and sub-namespaces can be defined.

A syntax example:

```
namespace MyNamespace {
  extern MyString $std::string$;
  interface MyInterface {
    enum MyEnumType { Val1, Val2, Val3 };
    in MyEnumType myValuedEvent(MyString s);
    out void myReply();
    behaviour {
      on myValuedEvent(s) : { reply MyEnumType.Val3; }
    }
  }
}
```

In this example namespace `MyNamespace` is introduced. In its scope data type `MyString` and interface `MyInterface`are defined.

An example with nested namespaces introduces namespace `Mynamespace1`, and within that (among others) namespace `MyNamespace2`, which itself contains interface `MyInterface`:

```
namespace MyNamespace1 {
  extern MyString $std::string$;
  namespace MyNamespace2 {
    interface MyInterface {
      enum MyEnumType { Val1, Val2, Val3 };
      in MyEnumType myValuedEvent(MyString s);
      out void myReply();
      behaviour {
         on myValuedEvent(s) : { reply MyEnumType.Val3; }
      }
    }
  }
}
```

8.8.1.1 Namespace Re-definition

It is allowed to spread the definition of types, interfaces, components, and sub-namespaces over multiple instances of a namespace scope. This is most useful since in a 'real' project definitions are spread over multiple files.

So

```
namespace MyNamespace1 {
  extern MyString $std::string$;
  interface MyInterface { ...  }
```

```
}
```

is equivalent to

```
namespace MyNamespace1 {
  extern MyString $std::string$;
}
namespace MyNamespace1 {
  interface MyInterface { ...  }
}
```

8.8.2 Referencing

When within namespace `MyNamespaceMyThing` is defined, then outside that namespace it is referred to by prefixing it with the name of that namespace and a dot, as in: `MyNamespace.MyThing`

Within its own namespace the short name `MyThing` is to be used.

In complex cases it may be necessary to refer to the *global* namespace which has an empty name; this results in name references starting with a dot, as can be seen in the following somewhat convoluted example.

```
namespace foo {
  interface I {
    enum Bool {F,T};
    in Bool e();
    out void a();
    behaviour {
      on e: {a; reply (Bool.T); }
    }
  }
}
namespace inner {
  namespace foo {
    interface I {
      enum Bool {f,t};
      in Bool e();
      out void a();
      behaviour { ....  }
    }
  }
  component space {
    provides foo.I inner;
    provides .foo.I fooi;
    behaviour {
      foo.I.Bool inner_state = foo.I.Bool.t;
      .foo.I.Bool foo_state = .foo.I.Bool.T;
      on inner.e(): {...}
      on fooi.e(): {...}
    }
  }
```

```
}
namespace bar {
  component c {
    provides foo.I i;
    behaviour {
      foo.I.Bool state = foo.I.Bool.T;
      on i.e(): {...}
    }
  }
}
```

which defines:

- interface `foo.I` with local enum `foo.I.Bool`
- interface `inner.foo.I` with local enum `inner.foo.I.Bool`
- component `inner.space`
- component `bar.c`

The two variables defined in component `inner.space` have types `foo.I.Bool` and `.foo.I.Bool` respectively. The first type expands to `inner.foo.I.Bool` since it is defined in namespace `inner`. The starting dot in the second definition prevents this expansion.

8.8.3 Shorthand Namespace Syntax

In trivial cases, e.g. where only one definition is provided within a namespace, a dot notation can be used. Using this notation,

```
namespace MyNamespace1 {
  namespace MyNamespace2 {
    extern MyString $std::string$;
  }
  interface MyInterface { ...  }
}
```

can be written as:

```
extern MyNamespace1.MyNamespace2.MyString $std::string$;
interface MyNamespace1.MyInterface { ...  }
```

9 Well-formedness

The syntax as defined in Chapter 8 [Dezyne Syntax], page 84, defines an asd language which is not restrictive enough for practical use. This page describes a collection of well-formedness checks that are defined on top of the syntax. Some of these checks are "common sense", other ones are needed to define appropriate semantics.

9.1 Summary

The well-formedness checks in alphabetical order:

See [Out Parameter is not allowed for out Event], page 118,

See [Parameter Binding not allowed here], page 119,
See [Parameter type must be a data type Code], page 118,
See [Port is bound twice *alarm.console*], page 111,
See [Port is not bound *alarm.console*], page 111,
See [Port is not bound *console*], page 111,

See [Reply not allowed on 'requires' Port 'sensor'], page 107,
See [Return Statement not allowed here], page 117,

See [Statement violates tail recursion in recursive Function], page 117,
See [System composition is recursive], page 114,

See [Wildcard can be bound to a provided port only], page 114,

9.2 Checks

Well-formedness checks on the Behaviour part of a model come in a number of categories:

- Directional: *triggers* and *actions* are expected at different places.
- Nesting: the imperative part of the language (AssignmentStatements, ActionStatements, FunctionCallStatements) are only allowed in the 'leaves' of the language constructs.
- Mixing: The mix of Statements within a CompoundStatements is restricted.
- Guards: There are some restrictions on what is allowed in a Guard.
- Functions: a function body should be imperative, and have a well-defined return.

Note: a trigger is an event prefixed by 'on' in the behaviour, an action is an event inside the body of a trigger.

Well-formedness checks for a composite Component are:

- Binding: all ports should be bound correctly.

9.3 Well-formedness Checks – Top level

9.3.1 Interface must define behaviour: *Sensor*

Interfaces without behaviour are not allowed. No adequate default behaviour is available:

```
1 interface Sensor {
2   in void enable();
3   in void disable();
4   out void triggered();
5   out void disabled();
6 }
```

This results in the following error message:

```
1:11: Interface must define behaviour: Sensor
```

9.3.2 Interface must define at least one event: *Sensor*

Completely 'passive' interfaces are not allowed; at least one **in** or **out** event is required:

```
1 interface Sensor {
2   behaviour {
3     // ...
4   }
5 }
```

This results in the following error message:

```
1:11: Interface must define at least one event: Sensor
```

9.3.3 Out Event with non void return type is not allowed: *evt*

Only *in* Event can have a return type.

```
1 interface Sensor {
2   enum Value{ New, Old };
3   in void enable();
4   in void disable();
5   out Value triggered();
6   out void disabled();
7   ...
8 }
```

This results in the following error message:

```
5:13: Out Event with non void return type is not allowed: triggered
```

9.3.4 Component with behaviour needs at least one trigger event: *Alarm*

Any Component with a **behaviour** specification is supposed to be 'reactive'. This implies that it should have at least one **provides** Interface with an **in** Event, or at least one **requires** Interface with an **out** Event. Such an Event acts as a 'trigger' for the Component to react on. So-called 'active' Components are not allowed (yet) in the ASD language.

An example:

```
 1 interface Sensor {
 2   in void activate();
 3   in void deactivate();
 4   behaviour { ...  }
 5 }
 6
 7 component Alarm {
 8   requires Sensor sensor;
 9   behaviour { ...  }
10 }
```

Another example:

```
1 interface Siren {
2   out void start();
3   out void stop();
```

```
 4   behaviour { ...  }
 5 }
 6
 7 component Alarm {
 8   provides Siren siren;
 9   behaviour { ...  }
10 }
```

Both examples result in the following error message:

```
7:11: Component with behaviour needs at least one trigger event: Alarm
```

9.3.5 Component with behaviour must have at least one provides port: *Alarm*

Any Component with a **behaviour** specification must have a provides port through which the Component is activated.

An example:

```
1 component Alarm {
2   requires Sensor sensor;
3   behaviour { ...  }
4 }
```

The examples result in the following error message:

```
1:11: Component with behaviour must have at least one provides port: Alarm
```

9.4 Well-formedness Checks – Directional

9.4.1 Event is not an action: *console.arm*

Event *console.arm* is used as *action*, but is introduced as a *trigger*.

In an InterfaceModel this indicates the Event is defined as an **in** event.

In a DesignModel this indicates that either it is an **in** Event of a **provides** interface of the design, or an **out** Event of a **requires** interface.

9.4.2 Event is not a valid trigger: *console.detected*

Event *console.detected* is used as *trigger*, but is introduced as an *action*.

In an InterfaceModel this indicates the Event is defined as an **out** event.

In a DesignModel this indicates that either it is an **out** Event of a **provides** interface of the design, or an **in** Event of a **requires** interface.

9.5 Well-formedness Checks – Nesting

The nesting of language constructs is restricted. When we conceptually 'expand' CompoundStatements, two Statement levels can be identified:

- The outer level we call **declarative**. It consists of exactly one OnEventStatement, optionally guarded at either side with GuardedStatements and at most one BlockingStatement. Valid examples are:

```
1 on sensor.triggered(): { ...  }
2 [guard1] [guard2] on sensor.triggered(): { ...  }
3 [guard1] on sensor.triggered(): [guard2] { ...  }
4 on sensor.triggered(): [guard1] [guard2] { ...  }
5 [guard1] blocking [guard2] on sensor.triggered(): { ...  }
6 blocking [guard1] on sensor.triggered(): [guard2] { ...  }
7 on sensor.triggered(): [guard1] [guard2] blocking { ...  }
```

- The inner level we call **imperative**. It consists of ActionStatements and AssignmentStatements. Imperative statements are only allowed within a declarative context:

```
1 [state.Armed]
2 {
3   on sensor.triggered():
4   {
5     // the imperative part:
6     console.detected();
7     state = States.Triggered;
8   }
9 }
```

9.5.1 AssignmentStatement only allowed within OnEventStatement

An AssignmentStatement occurred outside the scope of a declarative context:

```
1 [state.Armed]
2 {
3   state = States.Triggered;
4 }
```

This results in the following error message:

```
3:3: Well-formedness error: AssignmentStatement only allowed within OnEventStatement
```

9.5.2 ActionStatement only allowed within OnEventStatement

An ActionStatement occurred outside the scope of a declarative context:

```
1 [state.Disarming]
2 {
3   console.detected();
4 }
```

This results in the following error message:

```
3:3: Well-formedness error: ActionStatement only allowed within OnEventStatement
```

9.5.3 OnEventStatement not allowed within other OnEventStatement

A second OnEventStatement occurred in the declarative context:

```
1 on console.arm():
2 {
3   [state.Disarming]
```

```
4   {
5     on sensor.triggered():
6     { ...  }
7   }
8 }
```

This results in the following error message:

```
5:5: Well-formedness error: OnEventStatement not allowed within other OnEventStatement
```

9.5.4 BlockingStatement not allowed within other BlockingStatement

A second BlockingStatement occurred in the declarative context:

```
1 blocking on console.arm():
2 {
3   [state.Disarming]
4   {
5     blocking
6     { ...  }
7   }
8 }
```

This results in the following error message:

```
5:5: Well-formedness error: BlockingStatement not allowed within other BlockingStateme
```

9.5.5 BlockingStatement not allowed in interface behaviour

Events handling can be 'blocking' in component behaviour only. It is not allowed in interfaces. So:

```
1 interface Sensor {
2   in void enable();
3   in void disable();
4   behaviour {
5     on enable: { ....  }
6     blocking on disable: { ...  }
7   }
8 }
```

results in the following error message:

```
6:5: Well-formedness error: BlockingStatement not allowed in interface behaviour
```

9.5.6 BlockingStatement not allowed with multiple provides ports

A component with more than one provides port is not allowed to block events, due to implementation restrictions. So:

```
1 interface Sensor {
2   in void activate();
3   out void activated();
4   behaviour {
5     on activate: {}
6     on inevitable: activated;
```

```
 7   }
 8 }
 9
10 interface Test {
11   in void test();
12   behaviour {
13     on test: {}
14   }
15 }
16
17 component SensorComp {
18   provides Sensor sensor1;
19   provides Test test;
20   requires Sensor sensor2;
21   behaviour {
22     blocking on sensor1.activate(): sensor2.activate();
23     on sensor2.activated(): { sensor1.activated(); sensor1.reply(); }
24     on test.test(): {}
25   }
26 }
```

results in the following error message:

```
22:5: Well-formedness error: BlockingStatement not allowed with multiple provides port
```

9.6 Well-formedness Checks – Mixing

A behaviour description introduces a sequence of statements. A statement itself can be a CompoundStatement, which is a sequence of statements between curly braces.

In order to be able to define clear semantics, there are some restrictions on the mix of statements in such a sequence.

9.6.1 Only declarative Statement allowed here

If a sequence starts with a declarative Statement (as in *[state.Armed]....*or *on console.disarm(): ...*), all other statements shall be declarative Statements.

9.6.2 Only imperative Statement allowed here

If a sequence starts with either an imperative Statement (as in *state = States.Armed;* or *siren.turnon();*), all other statements must also be imperative.

9.6.3 Otherwise guard combined with second otherwise is not allowed

An *otherwise* guard catches the remaining cases for a list of guards. For that reason it is not allowed to have more than one *otherwise* statement in a list. So:

```
1 on console.arm():
2 {
3   [sounding] sounding = false;
4   [otherwise] sounding = true;
```

```
5   [otherwise] illegal;
6 }
7
```

will result in the following error message:

```
4:4: Well-formedness error: Otherwise guard combined with second otherwise is not allo
5:3: Well-formedness error: Second otherwise defined here
```

9.6.4 Otherwise guard combined with non GuardedStatement is not allowed

An *otherwise* guard catches the remaining cases for a list of guards. For that reason it is not allowed combine an *otherwise* statement with a non-guarded statement. So:

```
1 on console.arm(): sounding = false;
2 [otherwise] { on console.disarm(): sounding = true; }
3
```

will result in the following error message:

```
4:2: Well-formedness error: Otherwise guard combined with non GuardedStatement is not
1:2: Well-formedness error: non GuardedStatement defined here
```

9.6.5 Illegal must be the only Statement in a compound

An *illegal* ActionStatement must occur on its own; no other ActionStatements or AssignmentStatements are allowed. That also applies if the *illegal* occurs in a nested CompoundStatement:

```
1 on console.arm():
2 {
3   {
4     { illegal; }
5   }
6   sounding = true;
7 }
```

This results in the following error message:

```
4:7: Well-formedness error: Illegal must be the only Statement in a compound
```

Using *illegal* within a conditional statement *is* allowed. Also the condition may be accompanied by other action statements, e.g:

```
1 on console.arm():
2 {
3   s = sensor.enable();
4   if (s.NOK) {
5     illegal;
6   }
7 }
```

9.6.6 Illegal is not allowed in if-then-else statements

In an Interface declaration an Event can only be declared *illegal* in a direct way. This is due to the declarative character of interfaces. To be more specific, it must not occur in an if-then-else construct. An example:

```
1 bool b = true;
2 on arm:
3 {
4   if (b) { illegal; }
5   else { state = States.Armed; }
6 }
```

This results in the following error message:

```
4:12: Well-formedness error: Illegal is not allowed in if-then-else statements
```

9.6.7 Illegal is not allowed in functions

In an Interface declaration an Event can only be declared *illegal* in a direct way. This is due to the declarative character of interfaces. To be more specific, it must not occur in a function body. An example:

```
1 void ill()
2 {
3   illegal;
4 }
5 on arm:
6 {
7   ill();
8 }
```

This results in the following error message:

```
3:3: Well-formedness error: Illegal is not allowed in functions
```

9.7 Well-formedness Checks – Reply

A 'reply' is required in the handling of a valued (i.e. non-void) event. It is also required in case an event (that might be void) is used in 'blocking' mode; in that case the occurrence of the reply might be postponed. In general this is hard to check statically. What can be checked is described below.

9.7.1 Reply not allowed on 'requires' Port: 'sensor'

A 'reply' which is issued to release a blocking event refers to the event's port. Since blocking is effective on 'provides' ports only, a well-formedness error is issued when a 'requires' port name is used. An example:

```
1 component Test {
2   provides Start start;
3   requires Sensor sensor;
4   behaviour {
5     on start.start(): { sensor.activate(); }
6     blocking on start.stop(): { sensor.activate(); }
7     on sensor.deactivated(): { sensor.reply(); }
8   }
9 }
```

This results in the following error message:

```
7:32: Well-formedness error: Reply not allowed on 'requires' Port: 'sensor'
```

9.8 Well-formedness Checks – Valued Actions and Functions

Both Actions and Functions can be valued, and as such are considered to be expressions. The places where they can be called are severely restricted. The main reason is that Actions and Functions (at least the ones containing Actions) cause a side effect. The order of evaluation in complex expressions becomes an issue when side effects are considered. In order to exclude that, valued Actions and Function calls can only occur in isolation at the right hand side of an assignment.

An extra restriction to this rule is put on the initial value of a global variable in a behaviour. Such an expression can not contain Actions and Functions at all, since Actions are only allowed within an OnEventStatement.

9.8.1 Actions are not allowed here

An Action is used in a complex expression.

```
 1 interface Sensor {
 2   enum Status { OK, NOK };
 3   in Status enable();
 4 }
 5 component Alarm {
 6   provides Console console;
 7   requires Sensor sensor;
 8   behaviour {
 9     ....
10     bool b = true;
11     on console.arm():
12     {
13       if (sensor.enable() == Sensor.Status.OK) {
14         state = States.Armed;
15       }
16       bool b = sensor.enable();
17       bool b = b || sensor.enable();
18     }
19   }
20 }
```

This results in the following error messages:

```
13:10: Well-formedness error: Actions are not allowed here
17:20: Well-formedness error: Actions are not allowed here
```

The call of the *sensor.enable()* action on line 16 is the only acceptable one.

Action is used as initial value of a global variable.

```
 1 interface Sensor {
 2   enum Status { OK, NOK };
 3   in Status enable();
 4 }
 5 component Alarm {
 6   provides Console console;
 7   requires Sensor sensor;
```

```
 8   behaviour {
 9     ...
10     Sensor.Status st = sensor.enable();
11     on console.arm():
12     {
13       ...
14     }
15   }
16 }
```

This results in the following error message:

```
10:24: Well-formedness error: Actions are not allowed here
```

9.8.2 Function calls are not allowed here

A Function call is used in a complex expression.

```
 1 interface Sensor {
 2   enum Status { OK, NOK };
 3   in Status enable();
 4 }
 5 component Alarm {
 6   provides Console console;
 7   requires Sensor sensor;
 8   behaviour {
 9     ....
10     bool ok() {
11       return (enab() == OK);
12     }
13     Sensor.Status enab() {
14       Sensor.Status r = sensor.enable();
15       return r;
16     }
17     ...
18   }
19 }
```

This results in the following error message:

```
11:14: Well-formedness error: Function calls are not allowed here
```

9.8.3 Function value discarded: *negate*

A valued Function is called without assigning its return value.

```
1 ...
2 bool negate(bool b)
3 {
4   return !b;
5 }
6 void test(bool b)
7 {
```

```
 8   negate(b);
 9 }
10 ...
```

This results in the following error message:

```
8:7: Well-formedness error: Function value discarded: negate
```

9.8.4 Action value discarded: *enable*

A valued Function is called without assigning its return value. An example:

```
 1 interface Sensor {
 2   enum Status { OK, NOK };
 3   in Status enable();
 4 }
 5 component Alarm {
 6   provides Console console;
 7   requires Sensor sensor;
 8   behaviour {
 9     ....
10     on console.arm():
11     {
12       sensor.enable();
13     }
14   }
15 }
```

This results in the following error message:

```
12:7: Well-formedness error: Action value discarded: enable
```

9.9 Well-formedness Checks – Injection

9.9.1 Injected Port has out event: *i*

When a Component has a required injected Port, the corresponding Interface shall not have out events. An example:

```
1 interface Intf {
2   in void evt();
3   out void notif();
4 }
5
6 component Comp {
7   requires injected Intf i;
8   .....
9 }
```

The following error message will be issued:

```
7:26: Well-formedness error: Injected Port has out event: i
```

9.10 Well-formedness Checks – Binding

In a composite Component the Component's ports and all sub Component ports must be bound correctly.

We'll use the following (somewhat artificial) example to elaborate on binding errors below:

```
 1 import Console;
 2 import Alarm_Impl;
 3 import Sensor_Impl;
 4 import Siren_Impl;
 5
 6 component AlarmSystem
 7 {
 8   provides Console console;
 9   provides Console consoleAlt;
10
11   system
12   {
13     Alarm_Impl alarm;
14     Sensor_Impl sensor;
15     Siren_Impl siren;
16
17     // bindings .....
18   }
19 }
```

Bindings in which 'wildcards' are involved will be described at the end of this section.

9.10.1 Port is not bound: *console*

No binding is specified for a port of a composite Component. If no bindings are added in the AlarmSystem Component above, the following error message will be issued (among others):

```
8:19: Well-formedness error: Port is not bound: console
9:19: Well-formedness error: Port is not bound: consoleAlt
```

9.10.2 Port is not bound: *alarm.console*

No binding is specified for a port of a sub Component. If no bindings are added in the AlarmSystem Component above, the following error message will be issued (among others):

```
12:16: Well-formedness error: Port is not bound: alarm.console
```

9.10.3 Port is bound twice: *alarm.console*

More than one binding is specified for a port of a composite Component or one of its sub Components. If line 17 in the AlarmSystem Component above is followed by

```
18 console <=> alarm.console;
19 alarm.sensor <=> sensor.sensor;
20 consoleAlt <=> alarm.console;
```

the following error message will be issued (among others):

```
20:20: Well-formedness error: Port is bound twice: alarm.console
```

9.10.4 Binding directions do not match

The directions of the left and right port mentioned in the binding do not match. The following constructs are allowed:

- When binding a port of the Component to a port of a sub Component, the directions must be the same:
  - **provides** binds to **provides**
  - **requires** binds to **requires**
- When binding a port of the Component to another port of the Component (does this make sense??), the directions must be the opposite:
  - **provides** binds to **requires** or vice versa.
- When binding a port of a sub Component to a port of another (or the same) sub Component, the directions must be the opposite:
  - **provides** binds to **requires** or vice versa.

If line 17 in the AlarmSystem Component above is followed by

```
18 console <=> consoleAlt;
```

the following error message will be issued (among others):

```
18:5: Well-formedness error: Binding directions do not match
```

9.10.5 Binding two wildcards is not allowed

When a 'wildcard' is used in a binding the other side of the binding cannot have a wildcard also.

An example:

```
 1 component Sys {
 2   provides Console console;
 3   system {
 4     Logger logger;
 5     SubSystem sub;
 6     ...
 7     logger.* <=> sub.*;
 8     ...
 9   }
10 }
```

This will result in the following error message:

```
7:14: Well-formedness error: Binding two wildcards is not allowed
```

9.10.6 Component is part of a cyclic binding

We can define communication 'direction' for bindings as follows:

- For two sub components communicating: the `requires` port directs to the `provides` port in the binding.

- For port forwarding (an external port is forwarded to a sub-component port) or vice versa: A `provides` external port directs to a `provides` sub-component port, and a `requires` sub-component port directs to a `requires` external port.

To prevent component re-entrancy and guarantee run-to-completion semantics, cycles in 'directed' communication are not allowed within a system component.

In the most trivial example, which creates a one-component cycle:

```
 1 component Comp {
 2   provides Intf prov;
 3   requires Intf req;
 4 }
 5
 6 component Sys {
 7   system {
 8     Comp c;
 9     c.prov <=> c.req;
10   }
11 }
```

the following error message will be issued:

```
8:5: Well-formedness error: Component is part of a cyclic binding: c
```

A more elaborate example creates a cycle over three components:

```
 1 component Comp1 {
 2   provides Intf p;
 3   requires Intf r;
 4 }
 5
 6 component Comp2 {
 7   provides Intf p1;
 8   provides Intf p2;
 9   requires Intf r1;
10   requires Intf r2;
11 }
12
13 component Sys {
14   provides Intf p1;
15   provides Intf p2;
16
17   requires Intf r1;
18   requires Intf r2;
19
20   system {
21     Comp2 c21;
22     Comp2 c22;
23     Comp1 c11;
24     Comp1 c12;
25
```

```
26     p1 <=> c21.p1;
27     p2 <=> c22.p1;
28     c21.r1 <=> c11.p;
29     c21.r2 <=> c12.p;
30     c22.r1 <=> c21.p2;
31     c22.r2 <=> r2;
32
33     c11.r <=> r1;
34     c12.r <=> c22.p2;
35   }
36 }
```

As a result the following error messages will be issued:

```
21:5: Well-formedness error: Component is part of a cyclic binding: c21
22:5: Well-formedness error: Component is part of a cyclic binding: c22
24:5: Well-formedness error: Component is part of a cyclic binding: c12
```

9.10.7 Wildcard can be bound to a provided port only

Since **injected** Ports are always **requires**Ports, and a wildcard is used to bind such a Port, the other side of a wildcard binding must be a **provides** Port. In this example:

```
 1 component Comp {
 2   requires injected Logger logger;
 3   .....
 4 }
 5
 6 component MyLogger {
 7   requires Logger logger;
 8   ...
 9 }
10
11 component Sys {
12   provides Console console;
13   system {
14     MyLogger myLogger;
15     SubSystem sub;
16     ...
17     myLogger.logger <=> *;
18   }
19 }
```

the following error message will be issued:

```
17:20: Well-formedness error: Wildcard can be bound to a provided port only
```

9.10.8 System composition is recursive

A system component may instantiate an arbitrary set of other components, which can be system components themselves. It is not allowed to have a self-instance neither directly nor indirectly, since that would lead to an infinite tree of components.

In the example below five system components are defined that have mutual instances. Component C1 instantiates C3, which instantiates C4, which instantiates C1.

```
 1 component C1 {
 2   system {
 3     C2 c2;
 4     C3 c3;
 5   }
 6 }
 7
 8 component C2 {
 9   system {
10     C5 c5;
11   }
12 }
13
14 component C3 {
15   system {
16     C4 c4;
17     C2 c2;
18   }
19 }
20
21 component C4 {
22   system {
23     C1 c1;
24   }
25 }
26
27 component C5 {
28   system { } // an empty system
29 }
```

All components involved in the cyclic instantiation will be report in the error messages:

```
1:11: Well-formedness error: System composition is recursive: 'C1'
14:11: Well-formedness error: System composition is recursive: 'C3'
21:11: Well-formedness error: System composition is recursive: 'C4'
```

9.10.9 External port must be bound to external port: *r2*

There is a restriction in the binding of external ports: when an external requires port of a system Component is bound, the other side of the binding must be an external requires port also (this is only possible when that is a port of a sub Component)

In the example below some errors are reported.

```
1 interface I {
2   in void e();
3   behaviour {
4     on e: {}
5   }
```

```
 6 }
 7
 8 component C1 {
 9   provides I p;
10   requires external I r1;
11   requires external I r2;
12   behaviour {
13     on p.e(): {}
14   }
15 }
16
17 component C2 {
18   provides I p;
19   behaviour {
20     on p.e(): {}
21   }
22 }
23
24 component S1 {
25   provides I p;
26   requires I r;
27   system {
28     C1 c1;
29     C2 c2;
30     p <=> c1.p;
31     c1.r1 <=> r;
32     c1.r2 <=> c2.p;
33   }
34 }
35
36 component S2 {
37   provides I p1;
38   provides I p2;
39   requires external I r1;
40   requires external I r2;
41   system {
42     S1 s1;
43     p1 <=> s1.p;
44     p2 <=> r2;
45     r1 <=> s1.r;
46   }
47 }
```

the following error message will be issued on *S2*:

```
44:12: Well-formedness error: External port must be bound to external port: r2
45:5: Well-formedness error: External port must be bound to external port: r1
```

9.11 Well-formedness Checks – Functions

- A Function body can only contain imperative statements, including ActionStatements. See the sections on 'Mixing' and 'Direction' above.
- A valued Function is required to have an explicit return.
- A return is only allowed in a function body.
- A recursive Function is required to be tail recursive

9.11.1 Function does not return a value in all cases

A valued Function should return a value using the ReturnStatement. An error is issued when a return is not guaranteed. An example:

```
1 bool func()
2 {
3   if (a && b)
4   { return true; }
5   else if (c)
6   { illegal; }
7 }
```

For this code fragment the following error message will be issued:

```
1:10: Well-formedness error: Function does not return a value in all cases: func
```

9.11.2 Return Statement not allowed here

A **return** Statement is restricted to Function bodies. So

```
1 on sensor.disabled():
2 {
3   [sounding]
4   {
5     return; // triggers an error
6   }
7 }
```

This will cause the following error to be reported:

```
5:5: Well-formedness error: Return Statement not allowed here
```

9.11.3 Statement violates tail recursion in recursive Function

A function that is recursive must be tail recursive, i.e. in its body any recursive function call shall not be followed by other Statements. So

```
1 component Sensor {
2   ...
3   behaviour {
4     ...
5     void f() {
6       bool b = false;
7       if (b) {
8         f();
9         b = true;
```

```
10       }
11     }
12   }
13 }
```

will result in the following error message:

```
9:9: Well-formedness error: Statement violates tail recursion in recursive Function
```

Remark: two functions f and g that are defined in terms of each other are considered (mutual) recursive also.

9.12 Well-formedness Checks – Data Parameters

The restrictions on Data parameters are summarised here.

9.12.1 Parameter type must be a data type: Code

All event parameters specified in an event declaration must be data parameters; in other words, they must have a data type. An example:

```
1 interface Sensor {
2   enum Code { ok, nok };
3   in Code activate(Code c);
4   behaviour { ...  }
5 }
```

This will result in the following error message:

```
3:20: Well-formedness error: Parameter type must be a data type: 'Code'
```

9.12.2 Out Parameter is not allowed for out Event

An out Event is not allowed to have an out Data Parameter. This example:

```
1 interface Sensor {
2   extern INT $int$;
3   in void enable();
4   in void disable();
5   out void triggered(out INT value);
6   out void disabled();
7 }
```

will result in the following error message:

```
5:30: Well-formedness error: Out Parameter is not allowed for out Event: value
```

9.12.3 Inout Parameter is not allowed for out Event

An out Event is not allowed to have an inout Data Parameter. An example:

```
1 interface Sensor {
2   extern INT $int$;
3   in void enable();
4   in void disable();
5   out void triggered();
6   out void disabled(inout INT dis);
```

```
 7 }
```

will result in the following error message:

```
6:31: Well-formedness error: Inout Parameter is not allowed for out Event: dis
```

9.12.4 Parameter Binding not allowed here

Parameter binding, which is the binding of a global data variable 'D' to an event parameter 'p' using the 'p <- D' construct, is allowed in an 'on event' context only. Any other location is reported as an error. So:

```
 1 extern xint $int$;
 2 component Test {
 3   provides Intf intf;
 4   behaviour {
 5     xint NR;
 6     on intf.evt(number):
 7       fn(number <- NR);
 8
 9     void fn(xint i) { ...  }
10   }
11 }
```

will result in the following error message:

```
7:17: Well-formedness error: '<-' not allowed here
```

10 Contributing

This project is a cooperative effort, and we need your help to make it grow! Please get in touch with us on `dezyne-devel@nongnu.org` and `#dezyne` on the Freenode IRC network. We welcome ideas, bug reports, patches, and anything that may be helpful to the project.

10.1 The Perfect Setup

The Perfect Setup for using Dezyne is to use the Dezyne IDE, see *the Dezyne IDE Manual*.

The Perfect Setup to hack on Dezyne is basically the perfect setup used for GNU Guile hacking (see Section "Using Guile in Emacs" in *GNU Guile Reference Manual*). To work on real-life Dezyne projects, you need more than an editor: you need an IDE, see *the Dezyne IDE Manual*.

`GNU Emacs` To edit `.DZN` files, use `emacs/dzn-mode.el`.

... ...

11 Glossary

Term	Context	Definition	Chapter
Action	Concept	Statement as part of the response to a trigger event.	-
behaviour	Keyword	Start of the behaviour section within an interface or component definition; this keyword must be followed by {}.	See Section 8.6 [Interface and Component Behaviour], page 88,
Bind	Concept	To associate two ports within a system section using the Dezyne language's <=> operator.	-
blocking	Keyword	Qualifier to indicate that the executing of the trigger is blocked till a corresponding reply is given on the port of the trigger	See Section 3.6.12 [Blocking], page 48,
bool	Keyword	Boolean type identifier.	See Section 8.3 [Types and Expressions], page 85,
Compliance	Concept	Whether or not, the behaviour of a component complies to the behaviour specification of the interface of each provided port.	-
Component	Concept	Unit of definition and instantiation. A component has zero or more ports. A component has a behaviour section, system section, or none.	-
component	Keyword	Begins the definition of a component; this keyword must be followed by an identifier and {}.	See Section 8.5 [Component Models], page 87,
condition	Concept	If-then-else statement.	-
else	Keyword	Else part of condition statement.	See Section 8.1 [Identifiers], page 84,

enum	Keyword	Enumeration type declaration.	See Section 8.3 [Types and Expressions], page 85,
Event	Concept	An event is declared as part of an interface and represents the type of the messages communicated between components. An event is implemented as a function call.	-
extern	Keyword	Type definition which can be used for data variables and data parameters. Followed by an identifier and $$'s. The part between $$'s is used in the generated code as the type definition for these data variables and parameters.	See Section 8.3 [Types and Expressions], page 85,
external	Keyword	Qualifier of a provided port definition indicating that the out events can be possibly delayed which is taken into account during verification.	See Section 3.3.4 [External], page 19,
false	Keyword	Boolean value false.	See Section 8.1 [Identifiers], page 84,
Guard	Concept	A boolean expression between [ ] as prefix of a statement	-
Hand-written	Concept	A hand-written component is defined as a component without a behaviour or system section in the component definition. The implementation should be provided in the target source language.	-
if	Keyword	The "if" keyword is followed by a boolean expression and a statement and optionally the keyword "else" followed by a second statement	See Section 8.1 [Identifiers], page 84,
illegal	Keyword	illegal statement. As a direct response indicates that the given event in the given state is not allowed to be communicated. Within a component, the illegal statement is also allowed within a function or if-then-else statement, indicating that the illegal statement should never be executed, i.e. comparable with "assert(FALSE);" of the C-language.	See Section 3.6.10 [Using illegal], page 45,

import	Keyword	The file following the import keyword is loaded and expanded at the location of the import statement	See Section 8.1 [Identifiers], page 84,
in	Keyword	Indicates for an event of an interface that the event is an incoming event "'or"' indicates for a data parameter of an event or function that the value of the parameter is incoming	See Section 3.2.3 [Specifying Events], page 13,
inevitable	Keyword	A trigger event that will inevitably occur provided no other events occur.	See Section 3.6.9 [Using inevitable and optional], page 43,
injected	Keyword	Qualifier of a required port to indicate that the binding of the port occurs at a higher level in the system hierarchy.	See Section 8.5 [Component Models], page 87,
inout	Keyword	Indicates for a data parameter of an event or function that the value of the parameter is incoming at the begin of the event or function call, and outgoing at the end.	-
Interface	Concept	An interface defines a set of events and the underlying protocol how components communicate between each other via their ports.	-
interface	Keyword	Begins the definition of an interface; this keyword must be followed by an identifier and {}.	See Section 8.4 [Interface Models], page 87,
namespace	Keyword	-	See Section 8.8 [Namespaces], page 96,
on	Keyword	Identifies an on-event statement which for one or more trigger events defines the corresponding response.	See Section 3.6.3 [Actions], page 30,

optional	Keyword	A trigger that optionally can happen.	See Section 3.6.9 [Using inevitable and optional], page 43,
otherwise	Keyword	Defines a catch-all guard which is true only if all the other guard expressions in a list of guarded statements evaluate to false.	See Section 3.6.4 [Specifying Stateful Behaviour], page 33,
out	Keyword	Indicates for an event of an interface that the event is an outgoing event "'or"' indicates for a data parameter of an event of function that the value of the parameter is outgoing	See Section 3.2.3 [Specifying Events], page 13,
Port	Concept	Within a component definition, a named instance of an interface.	-
provides	Keyword	Used in a component to define a port and its type which the component will provide.	See Section 3.3.3 [Ports], page 17,
reply	Keyword	Statement that defines the value that will be returned as result of the triggering event.	See Section 3.6.8 [Reply], page 40,
requires	Keyword	Used in a component to define a port and its type which the component will use.	See Section 3.3.3 [Ports], page 17,
return	Keyword	return statement; specifies the return value of a function (if any) and exits the function.	See Section 3.6.7 [Using functions], page 38,
Sequence view	Window	Message sequence chart (result of simulation or verification).	-

State	Concept	A state consists of all values of the state variables in scope.	-
subint	Keyword	A type representing a finite sequence of integers, expressed as {m..n} where m and n are integers and m <= n, for example {-2..1}.	See Section 8.3 [Types and Expressions], page 85,
System	Concept	A component containing a system section.	-
system	Keyword	Starts the system section of a component. A system section consists of two parts: instantiations of subcomponents, followed by all the bindings among the sub-components and ports of the component.	See Section 3.7 [Decomposing a Component], page 50,
Trace	Concept	An execution trace is a sequence of events representing a possible execution scenario.	-
true	Keyword	Boolean value true.	See Section 8.1 [Identifiers], page 84,
Type	Concept	An enum, subint, bool, or extern.	-
Variable	Concept	Declared within a behaviour section. A variable can be either a state variable which can be used a part of a guard of condition or a data variable which can be used as a parameter of events or functions	-
Verification	Concept	The execution of the verification engine which verifies for interfaces and components whether they are free of livelock, deadlock, and illegal behaviour. Also, for a component it is verified whether a component adheres to all the protocols (behaviour) of the interfaces of the ports of the component. During verification, all possible execution scenarios are considered guaranteeing a 100% coverage.	-

Concept Index

S

T

U

V

Programming Index

(Index is nonexistent)

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year your name*.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.3
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
> Texts. A copy of the license is included in the section entitled ``GNU
> Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being *list their titles*, with
> the Front-Cover Texts being *list*, and with the Back-Cover Texts
> being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.