Dezyne Reference Manual

Component based, formally verified.

The Dezyne developers

Edition 2.14.0
6 December 2021

Table of Contents

1 Introduction

Dezyne is a programming language and a set of tools to specify, validate, verify, simulate, document, and implement concurrent control software for embedded and cyber-physical systems.

Dezyne incorporates both model based as well as component based development. It enables an incremental and collaborative approach to complex system development by using a novel way of design by contract. The Dezyne language allows defining not just the structure, but equally the detailed behavior of a software system using a C like syntax. Its rigorous notation enables automatically creating both abstract and detailed diagrams consistent with both the structure and the behavior.

The Dezyne language has formal semantics expressed in mCRL2 (`https://mcrl2.org`) developed at the department of Mathematics and Computer Science of the Eindhoven University of Technology (TUE (`https://tue.nl`)). Dezyne requires that every model is finite, deterministic and free of deadlocks, livelocks, and contract violations. This achieved by means of the language itself as well as builtin verification through model checking. This allows the construction of complex systems by assembling independently verified components.

What Dezyne sets apart from other programming languages is the fact that it treats the language primitives of the message passing programming model as first class citizens.

Dezyne is Free Software. Everyone is encouraged to share this software with others under the terms of the GNU Affero General Public License version 3 or later (see AGPL3+ (`https://gnu.org/licenses#AGPL`)). Fundamentally, the Affero General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

If you find Dezyne useful, please let us know. We are always interested to find out how Dezyne is being used.

You are also encouraged to help make Dezyne more useful by writing and contributing additional functions for it, and by reporting any problems you may have, (See Chapter 12 [Contributing], page 126).

1.1 Purpose

The main purpose of Dezyne is to systematically support the development and evolution of programs for which the validity is determined by their detailed behavior under operational conditions (E-type programs[1]). These are also the type of program which change the most and are most negatively affected by that change.

1.2 Conditions for Using Dezyne

The distribution terms for Dezyne-generated code permit using the code in free software programs as well as in non-free or proprietary programs: The Dezyne code generator *transpiles* the user's Dezyne program into a program in the target language, e.g., `C++`, making

[1] Lehman Laws of Software Evolution (`https://cs.uwaterloo.ca/~a78khan/cs446/additional-material/scribe/27-refactoring/Lehman-LawsOfSoftwareEvolution.pdf`)

the resulting `C++` program a derivative work, inheriting its copyright and–if applicable, licensing terms.

The Dezyne runtime is distributed under the GNU Lesser General Public License (see LGPL3+ (`https://gnu.org/licenses#LGPL`)), which means that it can be freely and unconditionally used in unmodified form, also if you are creating a non-free or proprietary software. If you *modify* the Dezyne runtime code that you distribute with your program, one condition applies: The modifications must be made available.

Note: Dezyne comes with NO WARRANTY, to the extent permitted by law.

2 Installation

There are two ways to get started with Dezyne: use a binary installation that we prepared especially for you, or you can build Dezyne from source yourself. When you plan on contributing to Dezyne (see Chapter 12 [Contributing], page 126), you may want to skip the sections on binary installation.

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/dezyne-2.14.0-x86_64-linux.tar.gz`, e.g.:

   ```
   $ wget https://dezyne.org/download/dezyne/\
   dezyne-2.14.0-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/\
   dezyne-2.14.0-x86_64-linux.tar.gz.sig
   $ gpg --verify dezyne-2.14.0-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-2.14.0-x86-64.tar.gz
   ```

   Then try:

   ```
   $ cd dezyne-2.14.0
   $ ./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
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 3 [Getting Started], page 5! 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/dezyne-2.14.0-x86_64-windows.zip`,

after which you can extract the archive by using 7zip `https://www.7-zip.org/`. Please do not use the built-in Windows archive extraction tool: It does not honor the time stamps of the files in the archive and thus produces a faulty installation.

If your system comes with a virus scanner, consider creating an exception for dezyne-2.14.0.

...

and get busy Dezyne'ing, see Chapter 3 [Getting Started], page 5! You can skip the following sections about building from source.

2.2 Requirements

This section lists requirements when building Dezyne from source. The build procedure for Dezyne is the same as for GNU software, and is not covered here. Please see the files `README` and `INSTALL` in the Dezyne source tree for additional details.

Dezyne is available for download from its website at `https://dezyne.org/download.html`.

Dezyne depends on the following packages:

- GNU Guile (`https://gnu.org/software/guile/`), version 3.0.x, with readline support;
- Guile-JSON (`https://savannah.nongnu.org/projects/guile-json/`) 4.x;
- GNU Make (`https://www.gnu.org/software/make/`);
- mCRL2 (`https://mcrl2.org`), version 202106.0.

To use the code that is generated by Dezyne, which includes running the regression test:

- GCC's g++ (`https://gcc.gnu.org`), version 5.4 or later.

[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.

3 Getting Started

In general a program in Dezyne consists of interfaces, components, and "handwritten" code, including a `main`. For simple cases such as the examples in this chapter, a generic `main` can be generated and no handwritten code is needed.

The examples used in this chapter can be found in the Dezyne source tree at `doc/examples/` or downloaded from doc/examples/ in git (`https://savannah.nongnu.org/cgit/dezyne.git/dezyne/tree/doc/examples`).

3.1 Hello World!

Consider the trivial Dezyne interface `ihello_world`

```
interface ihello_world
{
  in void hello ();
  out void world ();

  behavior
  {
    on hello: world;
  }
}
```

It defines two events, named `hello` and `world` of type `void` and a trivial protocol in its `behavior`: whenever the `hello` trigger is received (`on hello:`), it responds synchronously with a `world` action.

This scenario can be explored using the simulator (See Section 9.9 [Invoking dzn simulate], page 58):

```
$ dzn simulate doc/examples/ihello-world.dzn
(header ((client) ihello_world provides) ((sut) ihello_world interface))
(state ((client)) ((sut)))
labels: hello
eligible: hello
>
```

As expected, `hello` is the only trigger that is eligible to execute; entering `hello` gives

```
> hello
<external>.hello -> ...
... -> sut.hello
... <- sut.world
<external>.world <- ...
... <- sut.return
<external>.return <- ...
(state ((client)) ((sut)))
(trail "hello" "world" "return")
labels: hello
eligible: hello
```

```
>
```

The simulator can also be run non-interactively to produce a friendlier trace view or sequence diagram

```
client          ihello_world
  .                   :
  .                   :
  .hello              :
  .------------------>:
  .                   :
  .              world:
  .<------------------:
  .                   :
  .             return:
  .<------------------:
```

Now consider a trivial component `hello_world`

```
import ihello-world.dzn;

component hello_world
{
  provides ihello_world p;

  behavior
  {
    on p.hello (): p.world ();
  }
}
```

it provides the `ihello_world` interface, which means that it promises to behave according to the protocol specified in the interface.

The trigger `p.hello` is the event `hello` when communicated over the port `p`, similarly the action is named `p.world`. Simulation gives:

```
$ dzn simulate --trail=p.hello doc/examples/hello-world.dzn
(header ((p) ihello_world provides) ((sut) hello_world component))
(state ((p)) ((sut)))
<external>.p.hello -> ...
... -> sut.p.hello
... <- sut.p.world
<external>.p.world <- ...
... <- sut.p.return
<external>.p.return <- ...
(state ((p)) ((sut)))
(trail "p.hello" "p.world" "p.return")
(state ((p)) ((sut)))
(labels "p.hello")
(eligible "p.hello")
```

with this trace diagram

```
p          hello_world
.               :
.               :
.hello          :
.-------------->:
.               :
.          world:
.<--------------:
.               :
.         return:
.<--------------:
```

From this component an executable program can be created (See Section 9.2 [Invoking dzn code], page 53)

```
$ dzn code doc/examples/ihello-world.dzn
$ dzn code --model=hello_world doc/examples/hello-world.dzn
$ g++ -I runtime/c++ hello-world.cc main.cc \
  runtime/c++/runtime.cc runtime/c++/pump.cc -lpthread
```

When running this executable and feeding it the trail, we get

```
echo 'p.hello p.world p.return' | ./a.out
<external>.p.hello -> sut.p.hello
<external>.p.world <- sut.p.world
<external>.p.return <- sut.p.return
```

3.2 A Simple State Machine

The `ihello_bool` interface introduces stateful behavior that is somewhat more interesting

```
interface ihello_bool
{
  in void hello ();
  in bool cruel ();
  out void world ();

  behavior
  {
    bool idle = true;
    [idle] on hello: idle = false;
    [!idle]
      {
        on cruel: {idle = true; reply (idle);}
        on cruel: reply (idle);
        on inevitable: {world; idle = true;}
      }
  }
}
```

This example introduces some new language aspects

`bool idle = true;`
: A boolean state variable, defining `idle=true` as the initial state,

`[idle]`
: A *guard*. Only when the expression between the brackets evaluates to `true` the `on` is eligible to execute. In the initial state, the `hello` trigger is the only thing that can occur. The guard and the on are *declarative* statements. After the declarative statements follows a,

`idle = false;`
: An *imperative* statement. When `hello` trigger occurs, the interface transitions to state `!idle`,

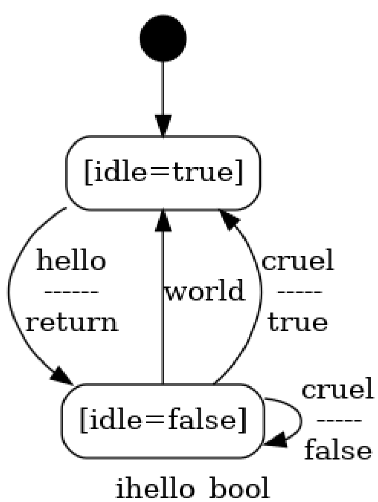
`on cruel: ... on cruel: ...`
: A non-deterministic choice[1]. In the `!idle` state, `cruel` is accepted; it can either...

`reply (true)`
: reply `false` and remain not idle, or

`{idle = true; reply (idle);}`
: execute a compound of two imperative statements: Set the reply value to `true` and transition to the idle state,

`inevitable`
: If no `cruel` trigger occurs, *inevitably* the `world` action will occur. `inevitable` is a *modeling* event and is not visible on the trail. The effect is that `world` action now has become *decoupled* from the caller.

The state diagram (See Section 9.4 [Invoking dzn graph], page 55) depicts this protocol graphically:



This model is already interesting enough to have the mCRL2 model-checker verify if all is well (See Section 9.12 [Invoking dzn verify], page 61, and See Section 6.1 [Verification Checks and Errors], page 35)

```
$ dzn -v verify doc/examples/ihello-bool.dzn
```

[1] the caller does not resolve the choice between the two cruel triggers, this is decided by the implementation

```
verify: ihello_bool: check: deadlock: ok
verify: ihello_bool: check: livelock: ok
verify: ihello_bool: check: deterministic: ok
```

which is luckily the case. The model-checker can also be used to generate all possible[2] traces (See Section 9.11 [Invoking dzn traces], page 60) for `ihello_bool`:

```
$ dzn -v traces doc/examples/ihello-bool.dzn
```

produces three trace files (`ihello_bool.trace.0`,`ihello_bool.trace.1`, and `ihello_bool.trace.2`) with these traces (the order may differ):

1. hello,return,world
2. hello,return,cruel,true
3. hello,return,cruel,false

The sequence for the first trace with the asynchronous `world` looks like this

```
client          ihello_bool
  .                  :
  .                  :
  .hello             :
  .----------------->:
  .                  :
  .            return:
  .<-----------------:
  .                  :
  .                  :
  .             world:
  .<-----------------:
```

and for the second trace where `cruel` happens looks like this

```
client          ihello_bool
  .                  :
  .                  :
  .hello             :
  .----------------->:
  .                  :
  .            return:
  .<-----------------:
  .                  :
  .                  :
  .cruel             :
  .----------------->:
  .                  :
  .              true:
  .<-----------------:
```

the third trace is looks like this

```
client          ihello_bool
```

[2] the algorithm produces [traces that cover every transition and every state

```
.                  :
.                  :
.hello             :
.----------------->:
.                  :
.            return:
.<-----------------:
.                  :
.                  :
.cruel             :
.----------------->:
.                  :
.             false:
.<-----------------:
```

You may have noticed that the first two traces start and end in the initial state, while the third trace starts in the initial state and ends in the `!idle` state (also see the corresponding state diagram).

Now have a look at the component `simple_state_machine`

```
import ihello-bool.dzn;

interface iworld
{
  in void hello ();
  out void world ();
  behavior
  {
    on hello: {}
    on hello: world;
  }
}

component simple_state_machine
{
  provides ihello_bool p;
  requires ihello_bool r1;
  requires iworld r2;
  behavior
  {
    enum status {A, B, C};
    status s = status.A;
    [s.A]
    {
      on p.hello (): {s=status.B; r1.hello (); r2.hello ();}
    }
    [s.B]
    {
```

```
          on p.cruel (): {if (r1.cruel ()) s=status.A; reply (s.A);}
          on r2.world (): s=status.C;
        }
        [s.B || s.C] on r1.world (): {s=status.A; p.world ();}
        [s.C] on p.cruel (): reply (s.A);
      }
    }
```

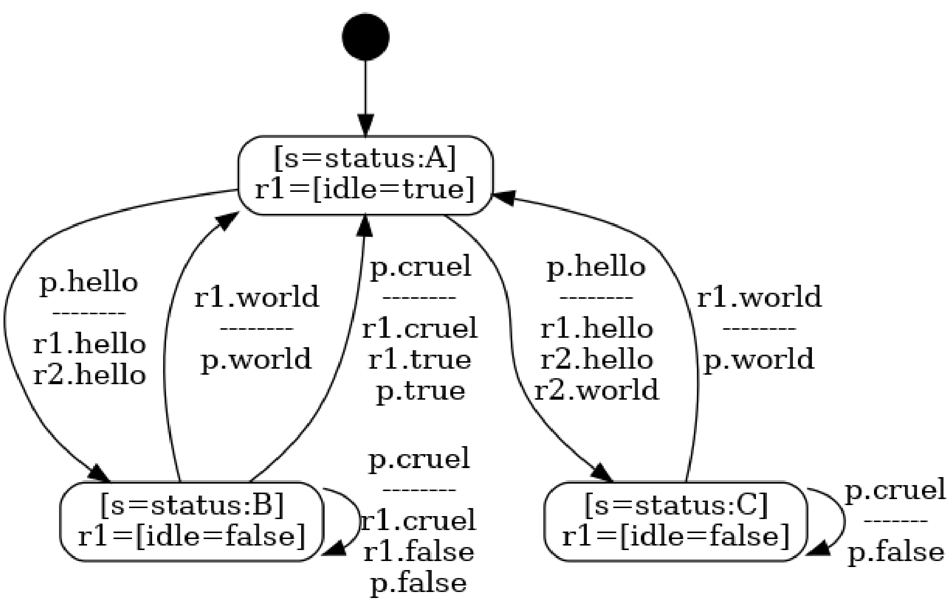
It introduces the following concepts:

`enum status {A, B, C}`
: User defined `enum` type named `status`,

`[s.A]`
: Field test of enum variable `s`: evaluates to `true` if `s` has field value `A`, it is equivalent to `s == status.A`,

`[s.B || s.C]`
: Logical or `||` in guard expression (see See Section 10.3.6 [Expressions], page 68),

`on r2.world (): {}`
: A skip statement: upon receiving the `r2.world` trigger, the component does "nothing" and is ready for the next event. Omitting this line would make the occurrence of `r2.world` *illegal*.



simple_state_machine

Verification suceeds

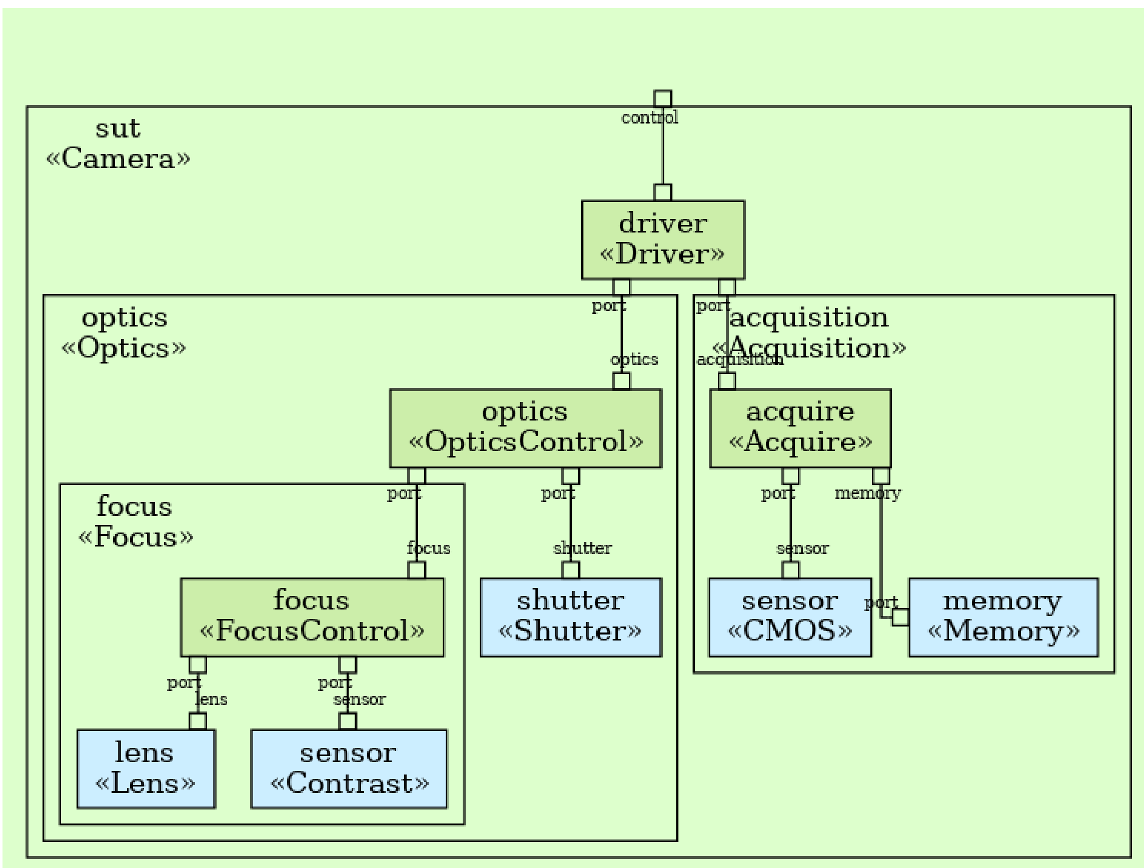
```
$ dzn -v verify doc/examples/simple-state-machine.dzn
verify: ihello_bool: check: deadlock: ok
verify: ihello_bool: check: livelock: ok
verify: ihello_bool: check: deterministic: ok
verify: iworld: check: deadlock: ok
```

```
verify: iworld: check: livelock: ok
verify: iworld: check: deterministic: ok
verify: simple_state_machine: check: deterministic: ok
verify: simple_state_machine: check: illegal: ok
verify: simple_state_machine: check: deadlock: ok
verify: simple_state_machine: check: livelock: ok
verify: simple_state_machine: check: compliance: ok
```

you may want to see what happens to verification or the state diagram when you comment-out a statement of your choosing in the component's behavior.

3.3 A Camera Example

The `Camera` example introduces the system component (See Section 10.6 [Systems], page 88). The system diagram (See Section 9.4 [Invoking dzn graph], page 55) looks like this:



This is what the Camera system looks like in Dezyne:

```
component Camera
{
  provides IControl control;
```

```
    system
    {
      Driver driver;
      Acquisition acquisition;
      Optics optics;

      control <=> driver.control;
      driver.acquisition <=> acquisition.port;
      driver.optics <=> optics.port;
    }
  }
```

It introduces the following concepts:

`provides IControl control;`
: Similar to a regular component, it defines ports,

`system`
: The `system` specification defines *instances* of components and their *bindings*,

`Driver driver;`
: A component instance named `driver` of type `Driver`,

`Acquisition acquisition;`
: A component instance named `acquisition` of type `Acquisition`, which is a `system` component itself,

`Optics optics;`
: An instance of another `system` component,

`control <=> driver.control;`
: A binding of the `Camera`'s port `control` to the port named `control` of the `driver` instance.

`driver.acquisition <=> acquisition.port;`
: A binding between pairs of ports on component instances.

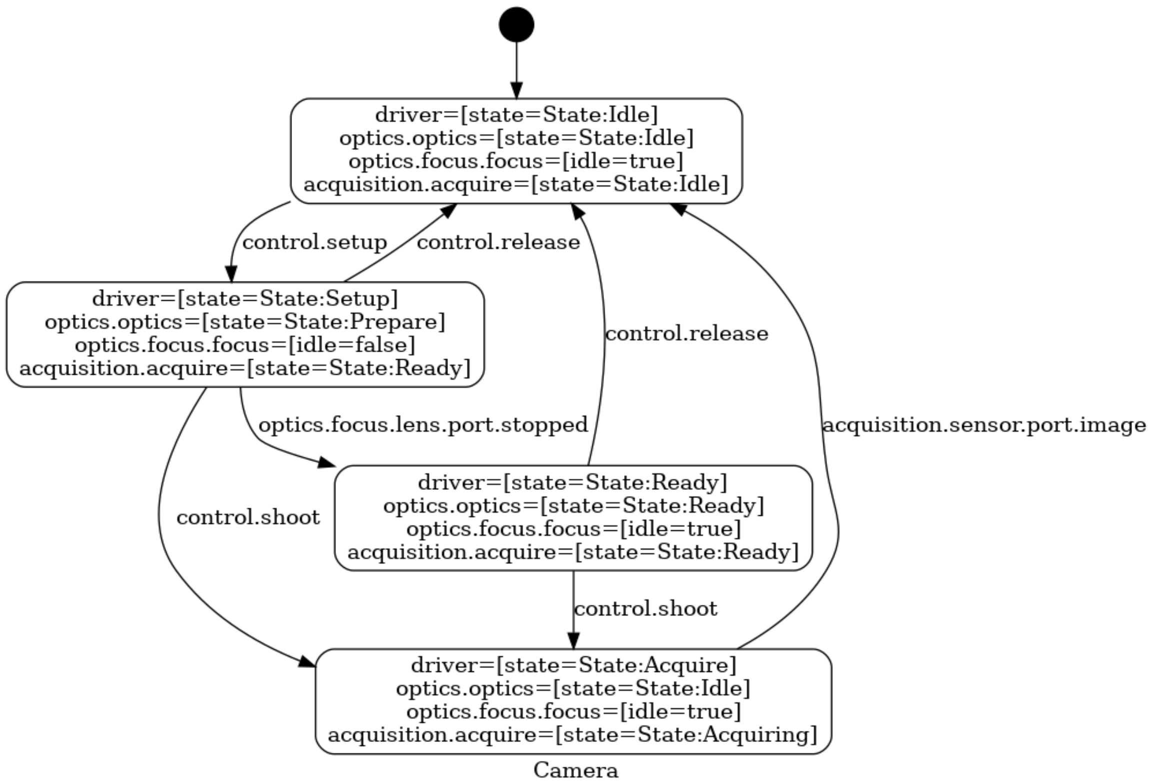
The light blue components in the system view, such as `lens` are *foreign* components (See Section 10.5 [Components], page 77); their definition looks like this:

```
component Lens
{
  provides ILens port;
}
```

A foreign component does not specify any implementation: neither a `behavior` nor a `system`; its behavior is said to be implementation elsewhere, and in a foreign language (in this case `C++`).

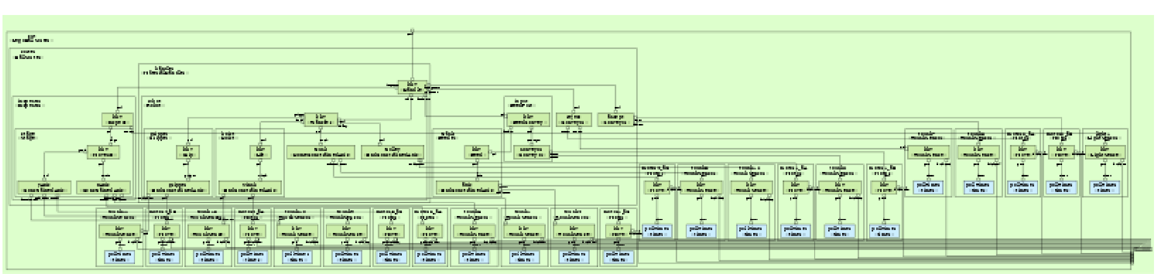
The full example is contained in the source tree at `test/all/Camera/Camera.dzn` or Camera.dzn (`https://savannah.nongnu.org/cgit/dezyne.git/dezyne/tree/test/all/Camera/Camera.dzn`).

The simplified[3] state diagram:



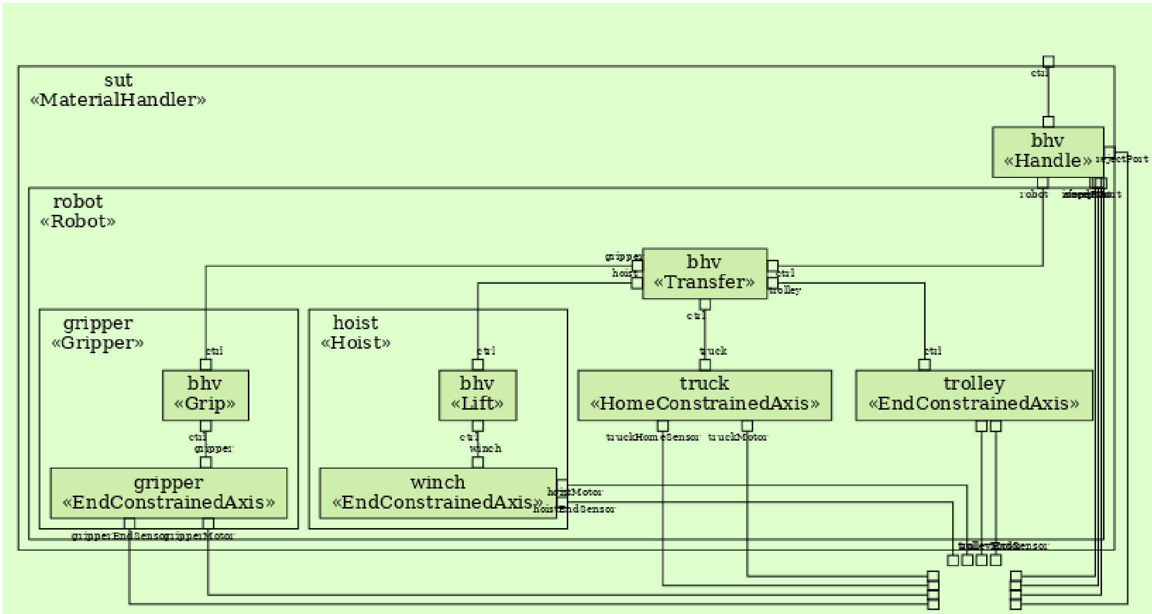
3.4 The Lego Ball Sorter

The Lego Ball Sorter example demonstrates how Dezyne can be used to write the operating system for a machine. The system view is already somewhat overwhelming:



[3] A simplified state diagram shows only triggers on state transitions and hides any actions or reply values. Also, the state of the ports or even all extended state can be removed. For this diagram, the command `dzn graph --backend=state --hide=actions --remove=extended test/all/Camera/Camera.dzn` was used.

so it makes more sense to look at smaller parts of the system, such as the `MaterialHandler`:



The full example is contained in the source tree at `test/all/LegoBallSorter/LegoBallSorter.dzn` or LegoBallSorter.dzn (`https://savannah.nongnu.org/cgit/dezyne.git/dezyne/tree/test/all/LegoBallSorter/LegoBallSorter.dzn`).

4 Ideas and Concepts

Dezyne aspires to evolve into a general-purpose operating-system language. The operating-system qualification refers to programs that are stateful, highly concurrent, long-lived, resilient, and exceptionally reliable. In contrast, short lived programs or programs that can be completely written in a pure functional way are not the primary target of Dezyne. By bringing mathematics and computer engineering together we hope to foster the creation and evolution of verified "operating system"[1] like applications.

The syntax of Dezyne may feel pretty familiar. The semantics is quite distinct from most other languages. Simply put Dezyne is the super position of a process calculus onto a general purpose language. As a result it adds new levels of organizational structure to the general purpose programming language.

4.1 Concurrency

Dezyne is based on a message passing programming model. Messages are explicitly represented in the language and expressed in the underlying process algebra. The approach allows reasoning about equivalences which in turn is used in verification and allows compositions to retain their individually verified properties.

Message passing is a natural way of describing concurrency, from cooperative multitasking to multi threading and multi processing. It abstracts away from cumbersome primitives like semaphores, mutexes, condition variables, critical sections, etc. It also removes the passing of time completely and focuses our reasoning on the ordering of messages. Which allows combining synchronous and asynchronous activity in a single formalism.

4.2 Component Based

In Dezyne programs are divided into components by means of formal interfaces isolating the components from their surroundings. Components are composed into systems by connecting their ports. Communication across port must follow the behavior as defined by their respective interfaces. An interface behavior describes the message exchange between the components on either side of the interface that separates them. A component behavior defines the interactions in terms of the messages exchanged across all of the component ports.

4.3 Model Based

As Dezyne is typically used to operate a machine, usually real world identities are represented. They are identified by name and their interaction with their environment is captured as a behavior. A behavior is the observable interaction in terms of message exchange. Interface models and component models both define behaviors.

---

[1] preferably microkernel based or at least distributed, the GNU Hurd (`https://hurd.gnu.org/hurd`) anyone?

4.4 Design by Contract

As interface behaviors prescribe an interaction protocol, they provide a convenient and compact way to define contracts. A contract lists both expectations and obligations. Components in turn are a convenient and compact way to implement such contracts using other sub contractors. Components distinguish two levels of hierarchy in their interface contracts: The interfaces they provide and the interfaces they require. The essential difference between the two is that an interface which is provided must be completely implemented. While an interface that is required may or may not be used completely.

4.5 Managing Complexity

The single biggest challenge, when programming at scale, is managing complexity. What do we mean by complexity? The literal meaning is derived from weaving together. In programming it refers to the resulting behavior that emerges from combined interaction. As the number of parts and their dependencies increase, the resulting behavior increases exponentially and very soon it reaches the point where it is no longer humanly possible to know and understand it. As a result making changes will inadvertently lead to unknowingly interfering with those interactions and defects are introduced. With increasing complexity existing techniques, methods and paradigms no longer suffice to enable the programmer to adequately manage it. Dezyne offers both encapsulation as well as abstraction of unencapsulated interaction to manage complexity.

5 Execution Semantics

The semantics of Dezyne derives from implementing message passing as component based ineraction by means of non-reentrant recursive function invocations. In Dezyne, the occurrence of an event is mapped onto a (member) function call. Every event function implements the recursive procedural execution of all of the side effects, e.g. actions, i.e., event function invocations, state updates, i.e., assignments, runtime library interaction: tracing, queueing and dequeueing and context switching, i.e., blocking and unblocking.

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 behavior.

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

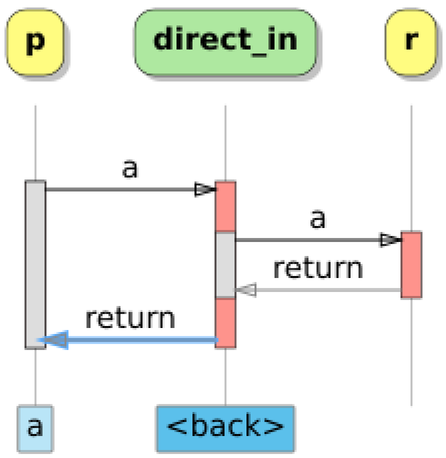
5.1 Direct in event

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 ();
 behavior
 {
   on a: {}
 }
}

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

```
}
```



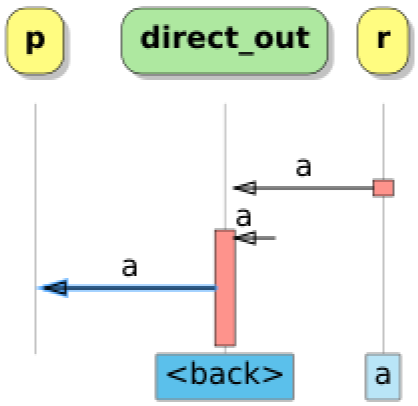
5.2 Direct out event

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 ();
  behavior
  {
    on inevitable: a;
  }
}

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

```
}
```



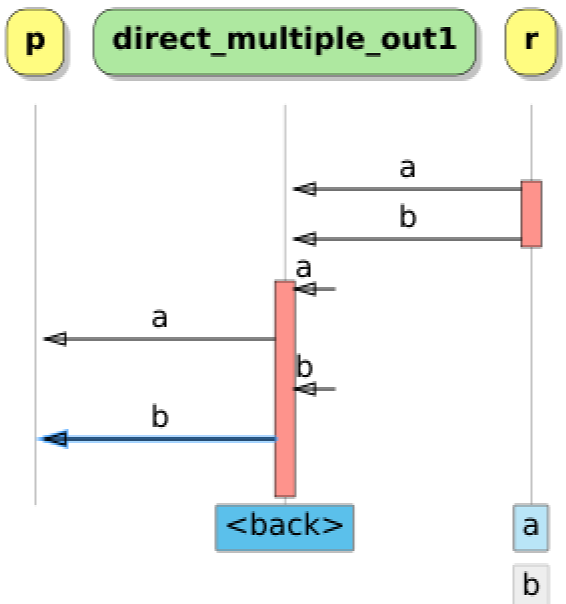
5.3 Direct multiple out events

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 indistinguishable looking from the outside.

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

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

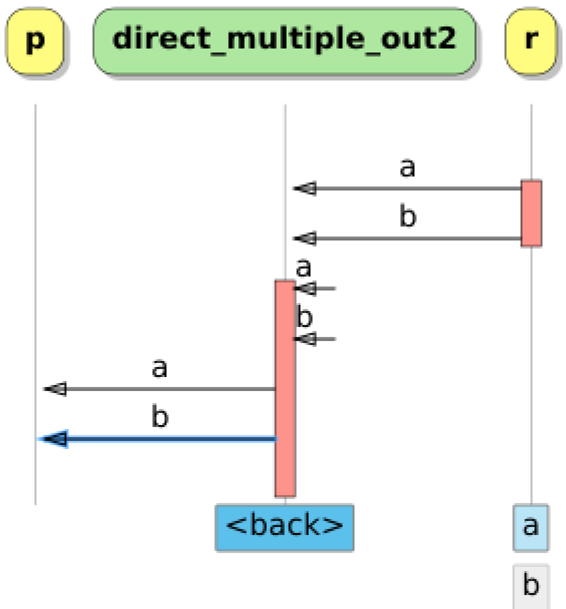
```
}
```




```
import direct_multiple_out.dzn;

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

```
}
```



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

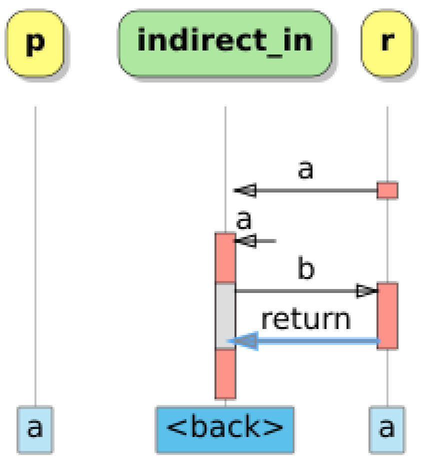
5.4 Indirect in event

A requires port in-event (`r.a`) call resulting in a requires port out-event (`r.b`).

```
interface U
{
  out void unused ();
  behavior
  {
    on inevitable: unused;
  }
}


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

```
component indirect_in
{
  provides U p;
  requires I r;
  behavior
  {
    on r.a (): r.b ();
  }
}
```



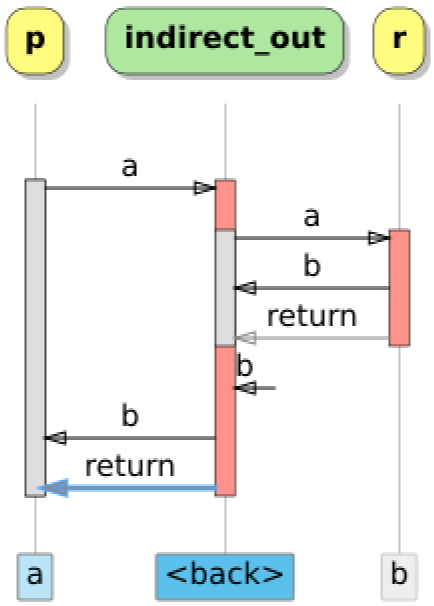
5.5 Indirect out event

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 ();
  behavior
  {
    on a: b;
  }
}

component indirect_out
{
  provides I p;
  requires I r;
  behavior
```

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



5.6 Indirect multiple out events

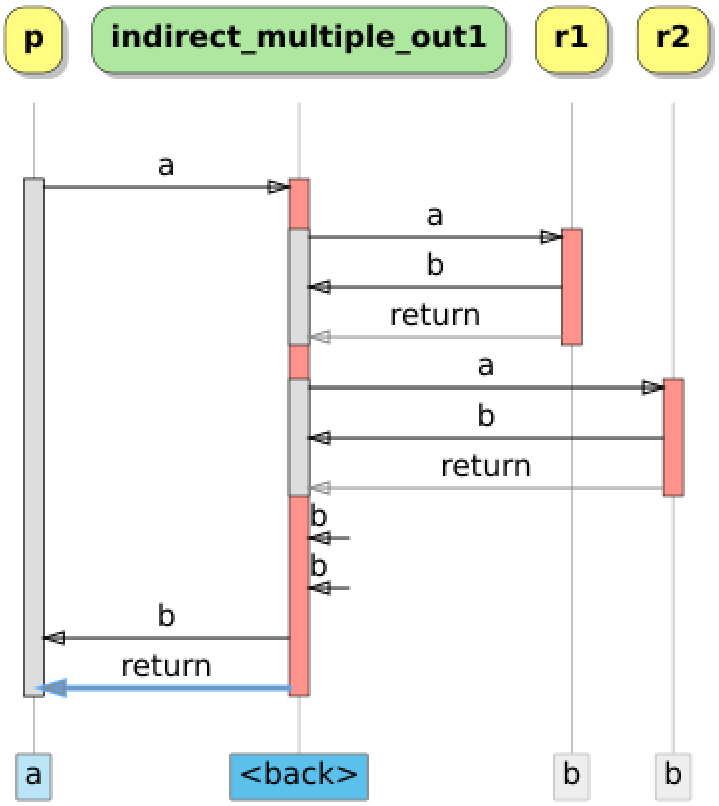
Since the provided interface is the same in the three cases below the externally visible behavior is identical.

The three different behavior implementations of the component show the subtle differences in the internal handling of messages.

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

component indirect_multiple_out1
{
  provides I p;
  requires I r1;
  requires I r2;
```

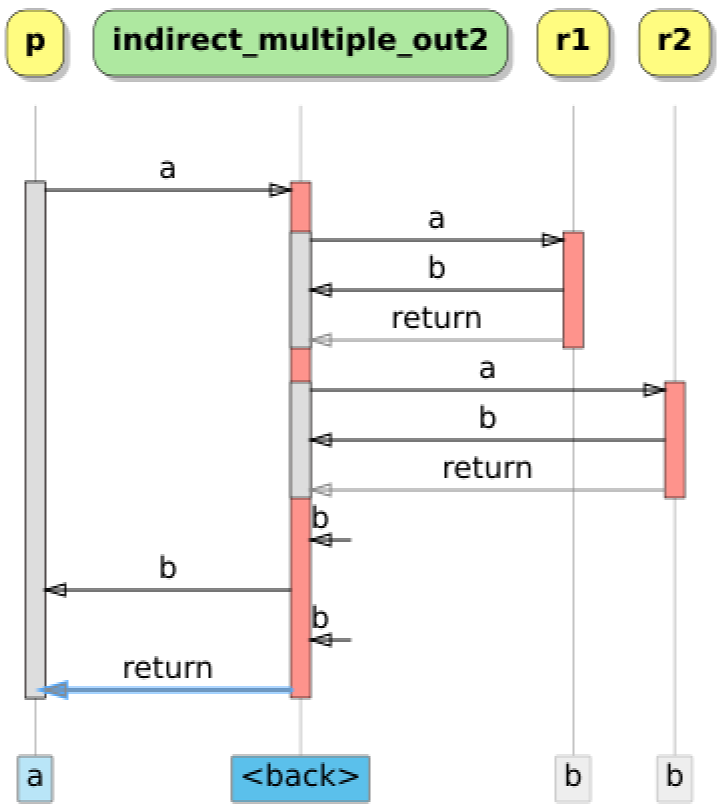
```
  behavior
  {
    on p.a (): {r1.a (); r2.a ();}
    on r1.b (): {}
    on r2.b (): p.b ();
  }
}
```




```
import indirect_multiple_out.dzn;

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

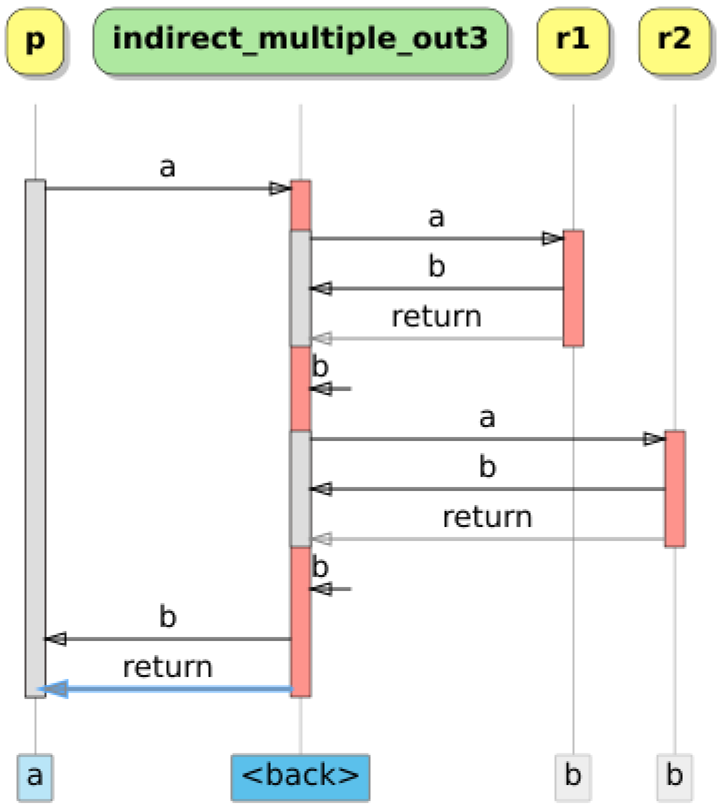
```
}
```




```
import indirect_multiple_out.dzn;

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

```
}
```



5.7 Indirect blocking out event

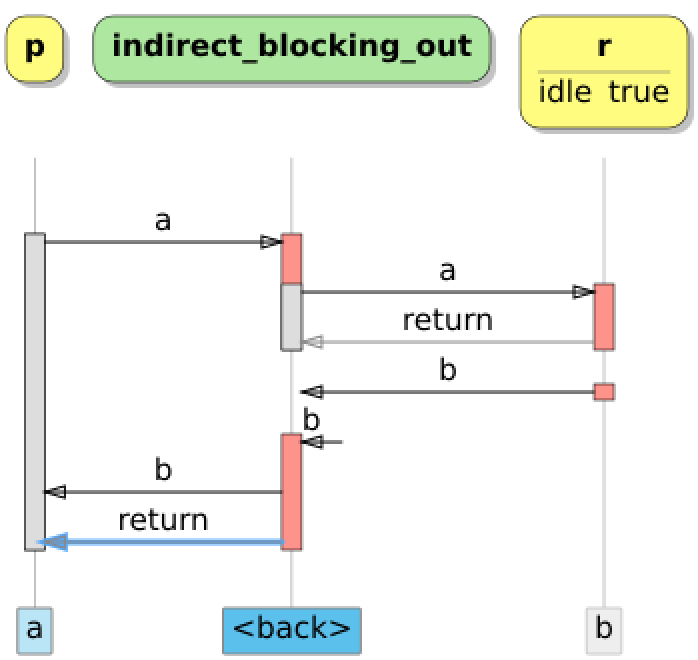
The in-event on the provides port (`p.a`) blocks (does not return) until a reply is handled. This happens in the handling of the requires port out-event (`r.b`). Also see Section 10.5.5.2 [Blocking], page 82.

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

interface I2
{
   in void a ();
   out void b ();
```

```
   behavior
   {
     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;
  behavior
  {
    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 provides in-event (`p.a`) would return before the provides out-event (`p.b`) would have been generated.

5.8 External multiple out events

The addition of `external` on a `requires` interface removes the atomicity of an action list, i.e: `{a; b;}`. Also see Section 10.5.1.2 [External], page 78.

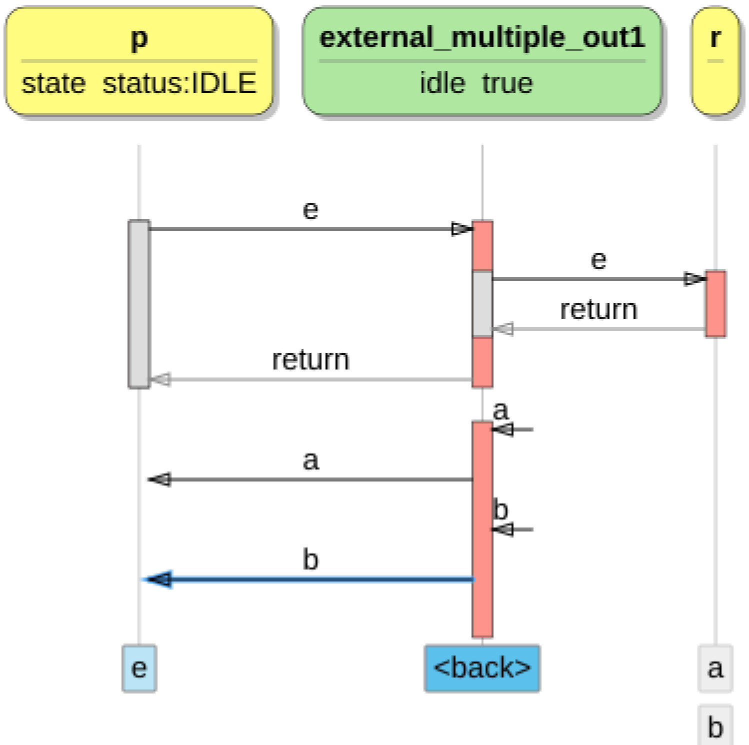
The first example shows how the behavior of `external J1` interface transforms into the interface behavior of `I1` by forwarding the events in the `external_multiple_out1` component behavior.

```
interface I1
{
  in void e ();
  out void a ();
  out void b ();
  behavior
  {
    enum status {IDLE, A, B};
    status state = status.IDLE;
    [state.IDLE] on e: state = status.A;
    [!state.IDLE] on e: illegal;
    [state.A] on inevitable: {state = status.B; a;}
    [state.B] on inevitable: {state = status.IDLE; b;}
  }
}

interface J1
{
  in void e ();
  out void a ();
  out void b ();
  behavior
  {
    on e: {a; b;}
  }
}

component external_multiple_out1
{
  provides I1 p;
  requires external J1 r;
  behavior
  {
    bool idle = true;
    [idle] on p.e (): {idle = false; r.e ();}
    [!idle] on p.e: illegal;
    on r.a (): p.a ();
    on r.b (): {idle = true; p.b ();}
  }
```

```
}
```



Two variations of the model above can be considered. Both variants provide the same interface behavior (I2 and I3 are identical), but differ in their requires interface behavior and as a result in their component behavior.

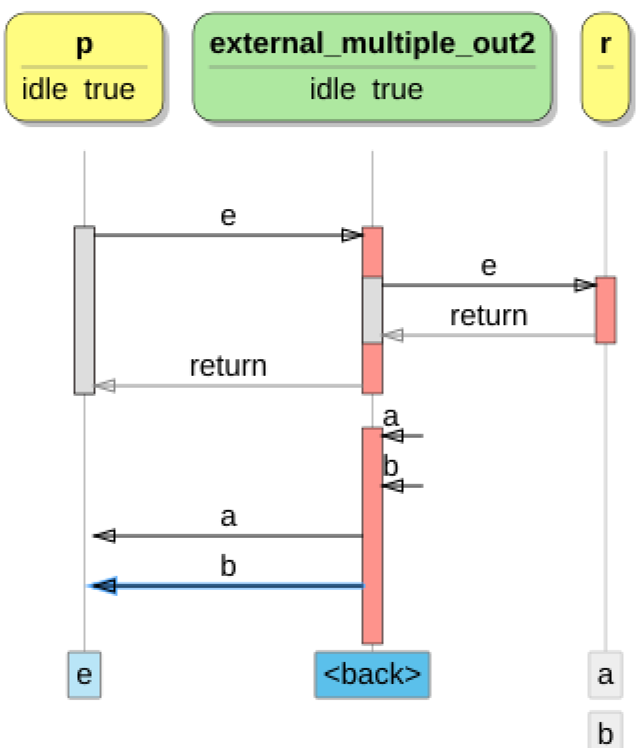
The first variant uses the requires behavior (J1 and J2 are identical) as the first example. The component takes care of joining the independently received events a and b as required by its provides interface.

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

interface J2
```

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

component external_multiple_out2
{
  provides I2 p;
  requires external J2 r;
  behavior
  {
    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 ();}
  }
}
```

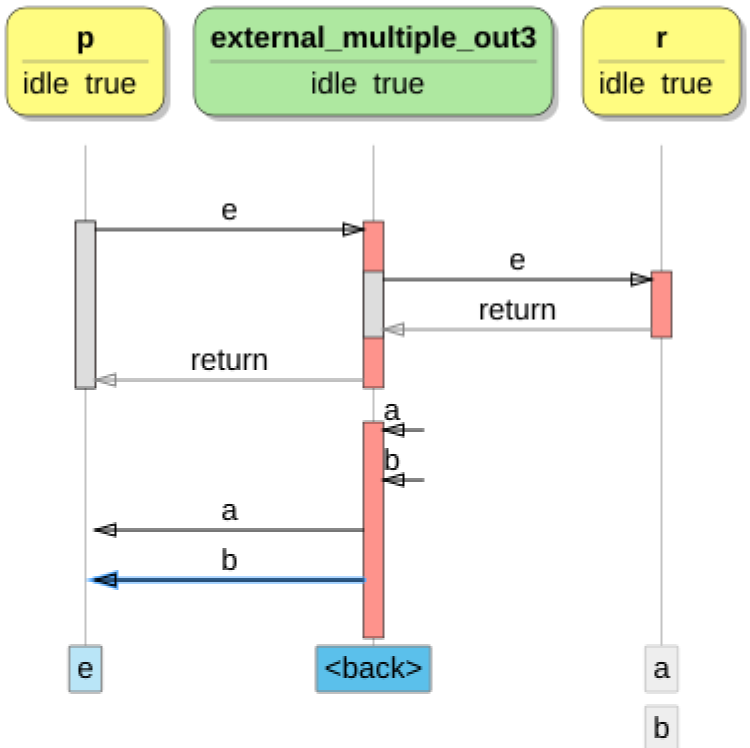


This variation provides the same interface as it requires. The component however, must make sure to join `a` and `b` again to implement its provides interface behavior.

```
interface I3
{
  in void e ();
  out void a ();
  out void b ();
  behavior
  {
    bool idle = true;
    [idle] on e: idle = false;
    [!idle] on e: illegal;
    [!idle] on inevitable: {idle = true; a; b;}
  }
}

component external_multiple_out3
{
  provides I3 p;
  requires external I3 r;
  behavior
  {
    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 ();}
  }
```

```
}
```



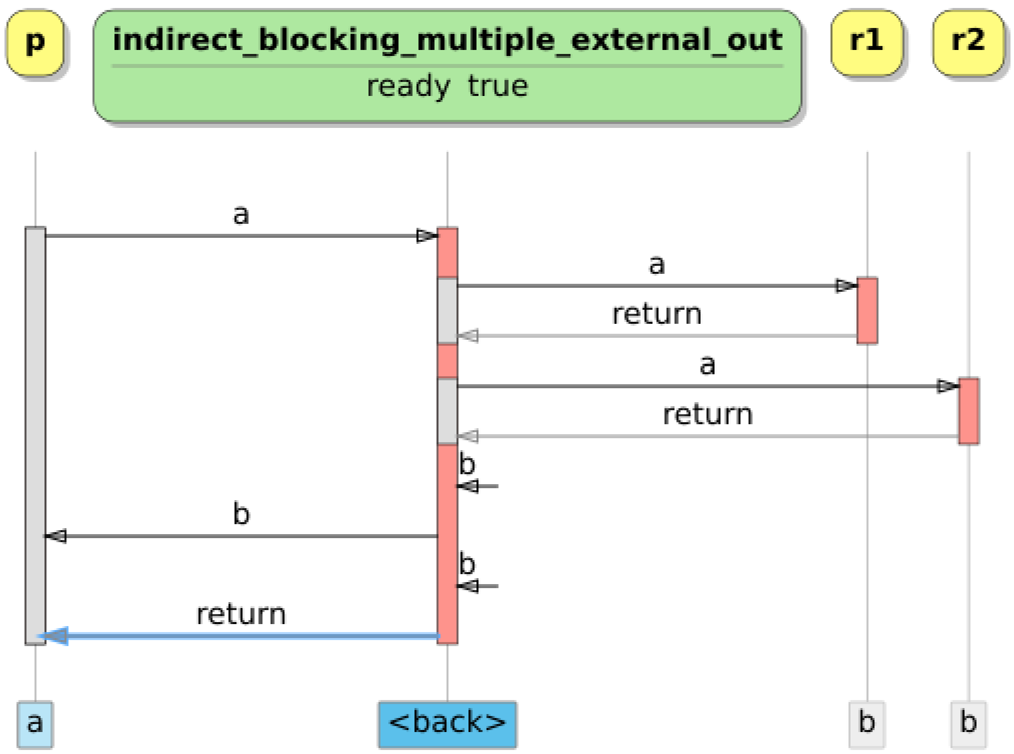
5.9 Indirect blocking multiple external out events

The two requires 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 behavior is the same in both cases.

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

component indirect_blocking_multiple_external_out
{
  provides I p;
  requires external I r1;
  requires external I r2;
  behavior
```

```
    {
      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 ();
    }
  }
```



6 Formal Verification

Dezyne provides automated, formal verification of a number of properties of interfaces, of components, and of components in relation to their interfaces[1].

By running `dzn verify`, Dezyne code is translated to mCRL2 (`https://mcrl2.org`) code and fed to a "verification pipeline", i.e., a series of `mcrl2` and `dzn` commands (See Section 9.12 [Invoking dzn verify], page 61).

The checks that Dezyne offers are of properties that are notoriously hard for humans to get right in all their detail, and which are relatively easily translatable to process algebra.

These properties concern the ordering of events, synchronous versus asynchronous calls and transactions, deadlock, livelock, and strict adherence to contract. Verifying a component together with its provides and requires interfaces ensures that the component behaves correctly in its environment according to the specified behavior. It also ensures that all possible error paths are fully and correctly handled.

All properties that Dezyne verifies on interface and component level are *compositional*, which implies, e.g., that as system consisting of verified components that do not deadlock, is also free of deadlock.

6.1 Verification Checks and Errors

A prerequisite for running the verification checks is for Dezyne code to be syntactically correct: any parse error prohibits the verification from running and must be fixed first. Apart from syntactic parse errors, the parser also checks for a number of so-called "well-formedness" errors. A "well-formedness" error is a static check, i.e. a check that the parser can perform without considering runtime behavior (see See Section 9.8 [Invoking dzn parse], page 57, and See Chapter 11 [Well-formedness], page 94).

Dezyne verifies for interfaces and for components:

`deadlock` A deadlock in an interface occurs when the interface reaches a state in which no `in`-events are specified.

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. Each component by itself can be verified as being deadlock free and within Dezyne this deadlock property is compositional, which means that 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., hand-written code can still cause deadlocks.

[1] Verification of systems and of functional properties are under development

Upon violation, the following error is reported:

```
error: deadlock in model <name>
```

`illegal` A trigger that is not handled in a certain state, results in an `illegal`. For components this is also verified for the use of the interfaces of its requires ports.

Upon violation, the following error is reported:

```
error: illegal action performed in model <name>
```

`livelock` A livelock in an interface occurs when in a certain state an `inevitable` event can occur without any restriction, i.e., its state does not change. This could starve the client that is interacting with this interface.

A livelock in a component occurs when it is permanently busy with internal behavior and fails to serve a provides port. For example, due to a design error such that the design is constantly interacting with its requires ports and starving a provides port; or due to the arrival rate of unconstrained external events such that processing them starves a provides port. 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 a component's provides port.

Upon violation, the following error is reported: `livelock in model <name>`

`range error`

Every possible assignment to a `subint` variable must be within its defined range.

Upon violation, the following error is reported:

```
error: integer range error in model <name>
```

`type error`

A trigger of a valued (i.e., non-`void`) event must reply a value of the type of the event.

Upon violation, the following error is reported

```
error: type error in model <name>
```

Note that trivial cases that can be checked statically, may be reported by the parser (See Chapter 11 [Well-formedness], page 94).

In addition, Dezyne verifies for interfaces:

`observable non-determinism`

Interfaces may specify non-deterministic behavior, as long as this non-determinism is observable by the client of that interface: after getting the response from the interface, a client must be able to determine what state the interface is in.

The snippet below shows observable non-determinism, i.e., an example of allowed non-determinism:

```
...
[idle] on hello: {world; idle=false;}
[idle] on hello: cruel;
...
```

in the `idle` state, upon sending `hello` either `world` or `cruel` may happen. This non-deterministic choice cannot be predicted. However, when the client sees `world`, the state of the interface is `not idle`, after seeing `cruel`, the state is `idle`.

This is an example of non-observable non-determinism, which is not allowed:

```
...
[idle] on hello: {world;idle=false;}
[idle] on hello: world;
...
```

as for a client it is impossible to tell if the interface is in state `idle` or in state `not idle`.

Upon violation, the following error is reported:

```
error: interface <name> is unobservably non-deterministic
```

In addition, Dezyne verifies for components:

`compliance`

The component together with its required interfaces implements the component behavior. The compliance check verifies that the component together with the required interfaces implements the behavior specified in the provided interface(s), i.e., whether the component honors its contracts.

Upon violation, the following error is reported:

```
error: component <name> is non-compliant with interface(s)\
    of provides port(s)
```

`determinism`

Components in Dezyne are required to be deterministic. The most common cause of non-determinism in a component is overlapping guards, i.e., in one state, for an event two different imperative statements are specified. Upon violation the following error is reported:

`error: component <name> is non-deterministic due to overlapping guards`

`queue full`

a Dezyne component 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 `--queue-size` option. The default queue size is `3`.

Upon violation, the following error is reported

```
error: queue full in model <name>
```

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.

6.2 Verification Counter Examples

A verification error does not only show the error it has detected, it also shows *where* it occurs. Where an error occurs is specified by means of a `counter example`, or an event trace.

Verifying

```
interface ihello
{
  in void hello ();
  in void world ();
  behavior
  {
    on hello: {}
  }
}

component illegal_requires
{
  provides ihello h;
  requires ihello w;

  behavior
  {
    on h.hello (): w.world ();
  }
}
```

gives:

```
$ dzn verify doc/examples/illegal-requires.dzn
model: hello
h.hello
w.hello
<illegal>
```

at the end of running this trace, an `illegal` action occurs.

6.3 Interpreting Verification Errors

Understanding why a certain verification error occurs, or how to fix it, is not always easy. The simulator can help to interpret the error and identify what is going on (See Section 9.9 [Invoking dzn simulate], page 58): It can show the source locations where the error occurs and the state the interface(s) and/or the component(s) are in.

The simulator can interpret the counter example from the verifier:

```
$ dzn verify doc/examples/illegal-requires.dzn \
  | dzn simulate doc/examples/illegal-requires.dzn
error: illegal action performed in model illegal_requires
(header ((h) ihello provides) ((sut) illegal_requires component) ((w) ihello requires))
(state ((h)) ((sut)) ((w)))
```

```
doc/examples/illegal-requires.dzn:6:3: error: illegal
<external>.h.hello -> ...
... -> sut.h.hello
sut.w.world -> ...
... -> <external>.w.world
<illegal>
(state ((h)) ((sut)) ((w)))
doc/examples/illegal-requires.dzn:6:3: error: illegal
(trail "h.hello" "w.world" "<illegal>")
(labels "h.hello" "h.world")
(eligible)
```

7 Defensive Design

As Dezyne is intended for operating system like applications, qualifications like trustworthy, secure, safe, robust, and resilient come to mind. Here we discuss how these might be achieved.

If you are dealing with untrustworthy partners, you had better check that they behave as agreed or otherwise stop the transaction. Practically this means that one must not rely blindly on external behavior and external input.

Dezyne interfaces allow you to specify what the implementation can expect from their client and what they must do in return. This is not unlike a contract in terms of a pre-condition and a post-condition. Moreover, verification can be used to exhaustively show that for each Dezyne component these pre- and post-conditions hold. This is what we call See Section 4.4 [Design by Contract], page 17, or See Section 7.1 [Interface Contracts], page 40.

Of course any interface contract can be written at the discretion of the designer/programmer. It can either be permissive or restrictive. An astute reader/thinker may realize that pre- and post-conditions are transitive and eventually there will not be a Dezyne implementation behind an interface. This means that verification cannot be used to assert upholding the pre- and post-conditions of the boundary interface. For this boundary we might define a permissive interface (anything goes) to guard the restricted interface and design an adapter component to deal with every request outside of the restricted protocol. This type of component is referred to as an armor (see See Section 7.3 [Armoring], page 41).

7.1 Interface Contracts

Dezyne does not have an exception mechanism like other languages. An exception mechanism is designed to prevent accidentally ignoring missed pre- or post-conditions. Instead, in Dezyne the interfaces establish these restrictions by means of verification (See Chapter 6 [Formal Verification], page 35). So where traditional programming languages must handle protocol violations using an exception mechanism at runtime, Dezyne prevents them using the static verification checks[1]. Interfaces in Dezyne are inherently complete with respect to their event alphabet. The generated code will accept every `trigger` but give an `illegal` response.

The illegal response is mapped to `std::abort ()` in `C++`. Note that for a fully verified Dezyne system, operated by clients that adhere to the interface specifications, it is impossible for an `illegal` response to be triggered. In other words, when an `illegal` is triggered, it means that some non-Dezyne code is violating a protocol (interface specification).

7.2 Error Handling and Recovery

The errors of concern here are not programming or design errors, but behavior that may occur and must be handled appropriately. Like a file open request because of a non existing file. Therefore these errors are at least runtime errors.

---

[1] This is not unlike languages that use static type analysis and checking (such as `C++` and `Haskell`) versus languages that check types at runtime

For a system, which behavior emerges as a result of its function and its interaction with an unpredictable environment, the Pareto principle holds for the distribution of its main functions and its error handling across its behavior. Typically about 10%-20% of the events that signal an error, result in 90%-80% of the behavior associated with error handling. While 90%-80% of the events that relate to the main functions of the system typically result in 10%-20% percent of the overall behavior which is unrelated to error handling.

Error handling is most often a matter of redirecting the handling to the party in charge to allow them to attempt recovery by retrying, continue with reduced or gracefully degraded function, by failing safely altogether, or continue as normal treating the error as a warning.

Dezyne is very effective in allowing engineers to discover the emergent error behaviors– i.e., without having to resolve to devising test scenarios, writing test code and running tests—as well as designing the handling of the respective error conditions.

7.3 Armoring

An armor is a defensive layer of components that protects the armored components who rely on their interface contracts from any behavior which would violate those contracts. An armoring component can be developed in Dezyne itself by creating a permissive interface from the strict interface behavior and letting the armor component map one behavior onto the other making sure the permissive behavior never violates the strict behavior.

The example below shows a simple but strict interface `istrict` and a permissive interface `ipermissive` that share the same event alphabet. The permissive interface is used on both sides of the `armored_system`. The system connects each permissive interface to a dedicated armor component, one for the top of the system and one for the bottom. Both protecting the inside component called `middle`.

```
interface istrict
{
  in void request();
  in void cancel();
  out void notify();

  behavior
  {
    bool idle = true;
    [idle]
    {
      on request: idle = false;
    }
    [!idle]
    {
      on cancel: idle = true;
      on inevitable: {idle = true; notify;}
    }
  }
}
```

```
interface ipermissive // derives from istrict
{
  in void request();
  in void cancel();
  out void notify();

  behavior
  {
    on request: {}
    on cancel: {}
    on optional: notify;
  }
}

component armored_system // is permissive, but armored
{
  provides ipermissive p;
  requires ipermissive r;

  system
  {
    p <=> ta.p;
    top_armor ta;

    ta.r <=> m.p;
    middle m; // the soft but strict middle
    m.r <=> ba.p;

    bottom_armor ba;
    watchdog w;
    ba.w <=> w.w;
    ba.r <=> r;
  }
}

component top_armor
{
  provides ipermissive p;
  requires istrict r;

  behavior
  {
    bool idle = true;
    [idle]
    {
      on p.request(): {idle = false; r.request();}
      on p.cancel(): {}
```

```
    }
    [!idle]
    {
      on p.request(): {}
      on p.cancel(): {idle = true; r.cancel();}
      on r.notify(): {idle = true; p.notify();}
    }
  }
}

component middle // a trivial proxy, see what happens with 2.17.0
{
  provides istrict p;
  requires istrict r;

  behavior
  {
    bool idle = true;
    [idle] {
      on p.request(): {r.request(); idle = false;}
    }
    [!idle] {
      on p.cancel(): {r.cancel(); idle = true;}
      on r.notify(): {p.notify(); idle = true;}
    }
  }
}

component bottom_armor
{
  provides istrict p;
  requires ipermissive r;
  requires iwatchdog w;

  behavior
  {
    bool idle = true;
    [idle]
    {
      on p.request(): {idle = false; w.set(); r.request();}
      on r.notify(): {}
    }
    [!idle]
    {
      on p.cancel(): {idle = true; w.cancel(); r.cancel();}
      on r.notify(),
         w.timeout(): {idle = true; w.cancel(); p.notify();}
```

```
    }
  }
}

interface iwatchdog
{
  in void set();
  in void cancel();
  out void timeout();

  behavior
  {
    bool idle = true;
    [idle]
    {
      on set: idle = false;
    }
    [!idle]
    {
      on cancel: idle = true;
      on inevitable: timeout;
    }
  }
}

component watchdog
{
  provides iwatchdog w;
}
```

8 Code Integration

Although Dezyne models can be simulated, the code is not suited for direct execution or compilation into machine executable code, instead, the a code generator is used to translate (transpile) Dezyne into an intermediate language, such as C++ (See Section 9.2 [Invoking dzn code], page 53).

The Dezyne code generator will produce human readable code that strongly resembles the Dezyne code without adding any unnecessary deviations.

8.1 Integrating C++ Code

This chapter describes the C++ code that is generated by Dezyne and the integration thereof.

8.1.1 Introduction

Every wellformed Dezyne model can be automatically converted into a corresponding wellformed C++ representation. A verified Dezyne model can be automatically converted into a corresponding C++ representation which when executed exhibits the same behavior 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.

8.1.2 Interfaces

Dezyne turns an interface such as:

```
interface some_interface
{
  in void in_event();
  out void out_event();

  behavior
  {
    on in_event: out_event;
  }
}
```

into the following C++ class representation:

```
struct some_interface
{
  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(){}
some_interface 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.

8.1.3 Components

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

```
import some_interface.dzn;

component some_component
{
  provides some_interface provided_port;
  requires some_interface required_port;
  behavior{}
}
```

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

```
struct some_component
{
  some_interface provided_port;
  some_interface required_port;
  some_component ()
  : 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 short circuiting events which will be initialized 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.

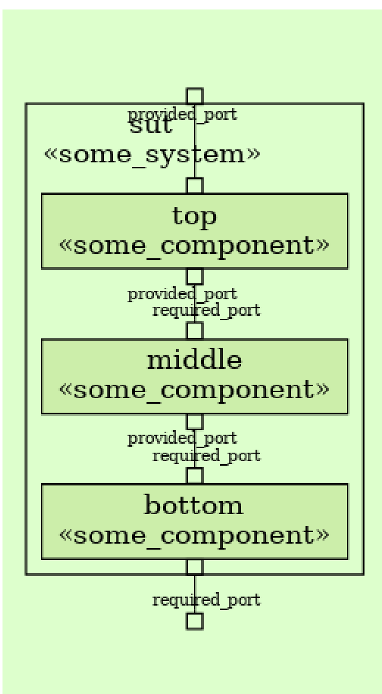
8.1.4 Systems

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

```
import some_component.dzn;

component some_system
{
  provides some_interface provided_port;
  requires some_interface required_port;
  system
  {
    some_component top;
    some_component middle;
    some_component 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:



8.1.5 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 "some_system.hh"
#include "dezyne/runtime.hh"
#include "dezyne/locator.hh"
int main()
{
  dezyne::locator loc;
  dezyne::runtime rt;
  loc.set(rt);
  //construct the system
  some_system 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();
  behavior
  {
    on bar:{}
  }
}
component some_component2
{
  provides some_component2 provided_port;
  requires injected Foo required_port;
  behavior { /* ...  */ }
}
int main()
{
  dezyne::locator loc;
  dezyne::runtime rt;
  loc.set(rt);
```

```
    Foo foo;
    foo.in.bar = []{/*no op*/};
    loc.set(foo);
    some_component comp(loc);
    comp.provided_port.in.in_event();
  }
```

8.2 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.

8.2.1 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.

8.2.2 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 provides port in event blocks until the thread-safe shell private thread has completed the deferred function call. The external call blocks until a `reply` 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 requires 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.

8.2.3 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);
  behavior {
    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 10.5.5.2 [Blocking], page 82,
- Chapter 9 [The Dezyne command-line tools], page 53,

8.3 Integrating Scheme Code

> **Note:** The Scheme code generator is still considered experimental; use with caution.

To enable the Scheme code generator, configure by doing something like

```
./configure --enable-languages=scheme --with-courage
```

Dezyne comes with a code generator for GNU Guile see *Guile reference manual*. Scheme is an interesting language for using with Dezyne. It supports a functional programming style that can be applied in handwritten code.

Program code written in a purely functional style is more reasonable than imperative code and especially so for concurrent programs (see Section "Modularity Objects and State" in *Structure and Interpretation of Computer Programs*). The Scheme code for Dezyne components that is generated by the code generator (See Section 9.2 [Invoking dzn code], page 53) can still use assignments to store state in an imperative way, but that is not a problem as this code is verified: the most tricky aspects of the software are left to Dezyne!

8.3.1 Namespace to Module

The Scheme code generator introduces the "namespace to module" feature which means that a Dezyne file, when it contains a single namespace, is assumed to describe a module, such as found in languages like GNU Guile (see Section "Modules" in *GNU Guile Reference Manual*), JavaScript (Python, etc.). Similarly, foreigns are assumed to live in their own module, so that this module can be used/required/imported.

Things to note:

- Dezyne files that define more than one namespace are not supported for "namespace to module",
- foreigns go into their own module,
- interfaces used by a foreign need to go into their own Dezyne file to avoid introducing cyclic dependencies,
- avoid the name `foreign`, the class `<foreign>`
- foreigns that use the same interfaces need to form a chain of `use-module` and `re-export` their port accessors (see Section "Using Guile Modules" in *Guile reference manual*).

9 The Dezyne command-line tools

9.1 Invoking dzn

The `dzn` command is a front-end to Dezyne functions, such as verification, code generation, simulation, etc. Those functions all have their own sub `command`:

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

Running `dzn` without a sub *command* shows a brief help text and the list of available `dzn` commands.

The *dzn-options* can be used with every `dzn` command and can be among the following:

`--debug`
`-d` Enable debug output.

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

`--skip-wfc`
`-p` Skip well-formedness checking.

The well-formedness checking of a large program can take a significant amount of time. As the well-formedness check does not change a correct `AST` in any way, it can be safely skipped when parsing a previously checked and unmodified program (See Section 9.8 [Invoking dzn parse], page 57).

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

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

Note: The *dzn-options* are placed between `dzn` and the sub command, e.g. to increase verbosity when using `dzn verify`, use

```
dzn -v verify file.dzn
```

9.2 Invoking dzn code

While the simulator (See Section 9.9 [Invoking dzn simulate], page 58) can interpret Dezyne code directly, to create an executable program Dezyne uses a code generator.

This code generator, the command `dzn code`, generates compilable or runnable code for a Dezyne file, such as `C++`. Usually—i.e., except for trivial cases—this generated Dezyne code is combined with "handwritten" code in the target language to create a Dezyne application, See Chapter 8 [Code Integration], page 45.

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

The *options* can be among the following:

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

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

`--import=`*`dir`*
`-I` *`dir`* Add directory *dir* to the import path.

`--init=`*`PROCESS`*
When generating `mCRL2` code, use init *PROCESS*. For other language backends, this options is ignored.

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

`--model=`*`model`*
`-m` *`model`* Generate a trivial main for *model*. This "generated main" can execute an event trace read from stdin, and writes a code trace to stderr. See Chapter 3 [Getting Started], page 5.

`--output=`*`dir`*
`-o` *`dir`* Write output to directory *dir* (use `-` for standard output).

`--queue-size=`*`size`*
`-q` *`size`* When generating `mCRL2` code, use queue size *size* for verification, the default is `3`. For other language backends, this options is ignored.

`--shell=`*`model`*
`-s` *`model`* Generate thread-safe system shell for model *model*. See Section 8.2 [Thread-safe Shell], page 49.

9.3 Invoking `dzn explore`

> **Note:** explore is deprecated, please use graph See Section 9.4 [Invoking dzn graph], page 55.

The `dzn explore` command generates a state diagram or an `lts` for a Dezyne model (See Section 9.7 [Invoking dzn lts], page 56, See Section 9.11 [Invoking dzn traces], page 60).

Under the hood, `dzn explore` uses the Dezyne VM.

```
dzn dzn-option... explore option... FILE
```

The *option*s can be among the following:

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

`--import=`*`dir`*
`-I` *`dir`* Add directory *dir* to the import path.

`--lts` Generate an `lts` in Aldebaren format.

`--model=`*`model`*
`-m` *`model`* Explore model *model*. The default is to use the most "interesting" model.

`--queue-size=`*`size`*
`-q` *`size`* Use queue size *size* for exploration, the default is `3`.

`--state-diagram`
Generate a state-diagram it `dot` format. This is the default.

9.4 Invoking dzn graph

The `dzn graph` command can be used to generate different graphs from a Dezyne model.

```
dzn dzn-option... graph option... FILE
```

The *options* can be among the following:

`--backend=type`
`-b type`
: Generate a diagram using backend *type*; one of `dependency`, `lts`, `state`, or `system` and write it to standard output. The default is `system`.

  The `state` diagram can simplified using options `--hide` and `--remove`.

  Under the hood, `lts` and `state` use the Dezyne VM. LTSs can be queried and manipulated using `dzn lts` (Section 9.7 [Invoking dzn lts], page 56) and the mCRL2 (`https://mcrl2.org`) tooling.

  > **Note:** Generating an LTS for a large component or system using the VM can be very time-consuming. For generating an LTS using the verification engine, see (Section 9.12 [Invoking dzn verify], page 61) and (Section 9.11 [Invoking dzn traces], page 60).

`--format=format`
`-f format`
: Print trace in format *format*; one of `aut`, `dot`, or `json`. For `--lts` the default is `aut`, for other formats the default is `dot`.

  > **Note:** The `json` can be processed by Dezyne-P5 (`https://gitlab.com/rma.wieringa/dezyne-p5`) to draw state and system diagrams in a browser.

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

`--hide=hide`
`-H hide`
: Generate a state diagram and hide *hide* from the transitions; one of `labels` (hide everything) or `actions`.

`--import=dir`
`-I dir`
: Add directory *dir* to the import path.

`--model=model`
`-m model`
: Generate graph for model *model*. The default is to use the most "interesting" model.

`--queue-size=size`
`-q size`
: Use queue size *size* for exploration, the default is `3`.

`--remove=vars`
`-R vars`
: Generate a state diagram and remove variables from nodes *remove*; one of `ports` or `extended`.

  `ports` Hides the state of the component's or system's ports, `extended` hides the interface's or component's extended state, i.e., all but the main (first) state variable and implies `ports`.

9.5 Invoking `dzn hello`

The `dzn hello` command can be used to test your installation; it echos "hello" to standard output.

```
dzn dzn-option... hello
```

The *options* can be among the following:

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

9.6 Invoking `dzn language`

The `dzn language` command produces Dezyne language completion results and location information. It can be used by an editor or IDE to create a rich editing experience.

```
dzn dzn-option... language option... FILE
```

The *options* can be among the following:

`--complete`
`-c` Show completion result; this is the default action.

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

`--import=`*dir*
`-I` *dir* Add directory *dir* to the import path.

`--offset=`*offset*
Use offset *offset* to determine context.

`--line=`*line*,*column*
`--point=`*line*,*column*
`-p` *line*,*column*
Calculate offset from line *line* and column *column*.

`--lookup`
`-l` Show lookup result.

`--verbose`
`-v` Display input, parse tree, offset, context and completions.

9.7 Invoking `dzn lts`

The `dzn lts` command can be used to manipulate and query a labeled transition system (*lts*) in Aldebaran (`aut`) format (See Section 9.4 [Invoking dzn graph], page 55, See Section 9.12 [Invoking dzn verify], page 61, See Section 9.11 [Invoking dzn traces], page 60).

```
dzn dzn-option... lts option... [FILE]...
```

The *options* can be among the following:

`--cleanup`
`-c` Rewrite mCRL2 labels to Dezyne, optionally remove *prefix* as specified with `--prefix`.

`--deadlock`
`-d` Detect deadlock in *lts* (after introduction of failures) and produce a witness.

`--exclude-illegal`
Remove edges leading to illegal (in combination with `--failures`).

`--failures`
`-f` Introduce a failure for each 'optional' event into the `lts`.

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

`--illegal`
`-i` Detect whether `lts` contains `<illegal>` labels.

`--livelock`
`-l` Detect tau-loops in `lts` and produce a witness.

`--deterministic-labels=`*`label`*`[,`*`label`*`...]`
`-n` *`label`*`[,`*`label`*`...]`
Detect whether `lts` is deterministic by detecting multiple edges of *label* from a single state, and produce a witness.

`--prefix=`*`prefix`*
Optional *prefix* for `--cleanup`

`--tau=`*`event`*`[,`*`event`*`...]`
`-t` *`event`*`[,`*`event`*`...]`
Hide all *event*s from `lts`.

`--exclude-tau=`*`event`*`[,`*`event`*`...]`
Exclude given *event*s from '`--tau`' list.

`--single-line`
`-s` Report each error including its trace (witness) on a single line.

9.8 Invoking `dzn parse`

The `dzn parse` command parses a Dezyne file and reports any errors, both syntax errors as well as "well-formedness" errors. The Dezyne parser consists of three stages:

1. The PEG parser creates a raw `parse-tree`[1],
2. The `parse-tree` is converted into the abstract syntax tree (`AST`),
3. A number of so-called "well-formedness" checks are performed on the `AST` that ascertain type correctness and detect semantic errors (See Chapter 11 [Well-formedness], page 94),
4. After parsing, some commands perform a normalization on the `AST`.

The well-formedness checking of a large program can take a significant amount of time. As the well-formedness check does not change a correct `AST` in any way, it can be safely

[1] The `dzn language` command (See Section 9.6 [Invoking dzn language], page 56) works on this raw `parse-tree`.

skipped when parsing a previously checked and unmodified program (See Section 9.1 [Invoking dzn], page 53).

Usually, the parser is invoked implicitly by commands like `dzn verify` and `dzn code`. It can be useful to do an explicit check for errors, for example after saving a Dezyne file (See Section 12.3 [The Perfect Setup], page 127). Its syntax is:

```
dzn dzn-option... parse option... FILE
```

The *options* can be among the following:

`--preprocess`
`-E` Resolve imports and produce a content stream. This pre-processed content can also be processed later by the parser and it has the advantage of being independent of the file-system.

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

`--import=dir`
`-I dir` Add directory *dir* to the import path.

`--list-models`
List the Dezyne models defined in the file, with their type.

`--locations`
`-L` Show locations in output *ast*.

`--model=model`
`-m model` Only output *ast* for model *model*.

`--parse-tree`
`-t` Write the raw *peg* parse tree, skip generating a full *ast*,

`--output=file`
`-o file` Write *ast* to *file*, use "-" for standard output.

9.9 Invoking `dzn simulate`

The `dzn simulate` command starts a simulation run.

Under the hood, `dzn simulate` uses the Dezyne VM. The simulator can be used to explore Dezyne models (interfaces, components, and systems), and to interpret error traces (witnesses) from the verification engine (See Chapter 3 [Getting Started], page 5). It shows code locations, state, and state transitions and produces friendly error messages. The simulator and verification both report the same errors (See Chapter 6 [Formal Verification], page 35). The simulator, however, only reports errors that it encounters while interpreting a specific event trace. The verifier performs an exhaustive search for errors but only produces a witness and does not report any context information. Its syntax is:

```
dzn dzn-option... simulate option... FILE
```

The *options* can be among the following:

`--no-deadlock`
`-D` Do not run the deadlock check.

`--no-interface-determinism`
: Do not run the observable non-determinism check on interfaces.

`--no-refusals`
`-R`
: Do not run the compliance check for the failures model refusals check.

`--format=`*`format`*
`-f` *`format`*
: Print trace in format *format*; one of `diagram`, `event`, or `trace`. The default is `trace`.

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

`--import=`*`dir`*
`-I` *`dir`*
: Add directory *dir* to the import path.

`--internal`
`-i`
: Display internal events when using the `diagram` trace format.

`--locations`
`-l`
: Display locations in the trace, this implies `--format=trace`.

`--model=`*`model`*
`-m` *`model`*
: Start simulating *model*. The default is the most "interesting" model.

`--queue-size=`*`size`*
`-q` *`size`*
: Use queue size *size* for simulation, the default is `3`.

`--strict`
`-s`
: Use strict matching of trail, i.e., the trail must contain all observable events.

`--trail=`*`trail`*
`-t` *`trail`*
: Use trail *trail*. The default is to read from `stdin`.

`--verbose`
`-v`
: Display non-communication steps in the trace, this implies `--format=trace`, `--locations`.

9.10 Invoking `dzn trace`

The `dzn trace` command is a pseudo-filter to convert between different trace formats:

`event trace (trail)`
: An event trace or *trail* is a list of event names observable by interacting with a Dezyne model, for example, for `doc/examples/hello-world.dzn`:

```
p.hello
p.world
p.return
```

`event trace (character separated)`
: Some tools, such as the simulator also read an event trace separated by a comma or a space:

```
 p.hello,p.world,p.return
"p.hello p.world p.return"
```

`code trace (arrow trace)`
: The Dezyne executable code can produce a trace showing the sender and the receiver of an event on the same line:

```
<external>.p.hello -> sut.p.hello
<external>.p.world <- sut.p.world
<external>.p.return <- sut.p.return
```

`simulator trace (split-arrow trace)`
: The simulator produces a trace showing the sender and the receiver of an event both on their own line:

```
<external>.p.hello -> ...
... -> sut.p.hello
... <- sut.p.world
<external>.p.world <- ...
... <- sut.p.return
<external>.p.return <- ...
```

which is especially useful when the lines are prefixed with location information.

The `dzn trace` command reads arrow traces and converts them to a code trace (the default) or an event trace. A split-arrow trace can also be converted to an ASCII sequence diagram. Its syntax is:

```
dzn dzn-option... trace option... [FILE]
```

The *options* can be among the following:

`--format=format`
`-f format` Display trace in format *format*, one of `diagram`, `event`, `json`, or `sexp`. The default is `code`.

> **Note:** The `json` can be processed by Dezyne-P5 (`https://gitlab.com/rma.wieringa/dezyne-p5`) to draw a trace diagram in a browser.

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

`--internal`
: Show communication between components in the system. When using the option `--format=diagram` on a system trace, the communication between components in the system is hidden by default.

`--locations`
`-L` Show locations in output.

`--trace=trace`
`-t trace` Use trace *trace*. The default is to read from standard input.

9.11 Invoking `dzn traces`

The `dzn traces` command generates an exhaustive set of event traces or trails for a behavioral Dezyne model. It can also be used to generate an *lts* in Aldebaran format (See

Section 9.7 [Invoking dzn lts], page 56, See Section 9.4 [Invoking dzn graph], page 55, See Section 9.12 [Invoking dzn verify], page 61).

Under the hood, `dzn traces` uses `dzn code` and mCRL2.

```
dzn dzn-option... traces option... FILE
```

The *option*s can be among the following:

`--flush`
`-f` Include `<flush>` events in trace.

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

`--illegal`
`-i` Include traces that lead to an illegal.

`--import=dir`
`-I dir` Add directory *dir* to the import path.

`--lts`
`-l` Instead of generating trace files, generate an *lts* in Aldebaran format.

`--model=model`
`-m model` Generate traces for model *model*.

`--output=dir`
`-o dir` Write trace files to directory *dir*.

`--queue-size=size`
`-q size` Use queue size *size* for generation, the default is `3`.

`--traces`
`-t` Generate trace files, this is the default. Using `--traces` will generate trace files even when `--lts` is used.

9.12 Invoking dzn verify

The `dzn verify` command exhaustively checks a Dezyne file for verification errors in Dezyne models. See Chapter 6 [Formal Verification], page 35.

```
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.

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

`--import=dir`
`-I dir` Add directory *dir* to the import path.

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

Note: Verification cannot be limited to `system` component models; verifying a system model is a no-op[2].

`--no-interface-determinism`
Do not run the observable non-determinism check on interfaces.

`--out=`*`format`*
Run a partial verification pipeline to produce *format*.

Interesting formats are `mcrl2`, `aut`, `aut-dpweak-bisim`, `aut-weak-trace`, and `aut+provides-aut`. Use `--out=help` for a full list.

The verification pipeline starts by generating `mCRL2` code, which is converted into an *lps* and then into an *lts* (See Section 9.7 [Invoking dzn lts], page 56). The *lts* is then manipulated further.

Using the `--debug` on `dzn` (See Section 9.1 [Invoking dzn], page 53) shows the pipelines with all their commands that are being used, ready for use on the command line.

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

---

[2] The compositional property of the Dezyne component-based programming paradigm guarantees that the verification of a `system` component model amounts to the verification of all its `interface` models and behavioral `component` models.

10 Dezyne Language Reference

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 behavior 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 representations: textual, graphical, mathematical, program and execution.

The `C`-like syntax of Dezyne should give it a familiar feel to many programmers. Dezyne has some unique language concepts and syntax elements that are described in this chapter.

10.1 Lexical Analysis

Dezyne is a `C`-like language. This means that identifiers must be separated by either whitespace, delimiters or operators and is otherwise whitespace invariant. The Dezyne parser is defined using a partial expression grammar or "PEG" (see Section "PEG Parsing" in *GNU Guile Reference Manual*).

10.1.1 Identifiers

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

```
identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
```

An identifier starts with a letter or an underscore, which can be followed by further letters, digits, or underscores. The following are all valid identifiers:

```
p, hello, Alarm, turn_on, VALUE_123, _
```

> **Note:** That by convention Dezyne identifiers are also used in the target language, however the target language may impose further restrictions on identifiers.

10.1.2 Keywords

The following list shows identifiers that are reserved words in Dezyne, or *keywords*. These keywords may not be used as an identifier name.

```
behavior      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
```

[1] Structural: events and their direction in an interface, ports on a component, components in a system, bindings between the ports; behavioral: guarded triggers performing actions | assignments | if-else | functions

10.1.3 Operators

Dezyne uses infix notation for expressions. The following are operators in Dezyne:

```
+        -        !
<        >        <=       >=       ==       !=       <=>
```

10.1.4 Delimiters

The following are delimiters in Dezyne, for introducing lists:

```
(        )        {        }        $
```

and for elements in lists:

```
,        .        ;        =
```

10.1.5 Lexical Scoping

A lexical *scope* adds locality to a name. Names in one lexical scope do not interfere (collide or shadow) with another scope. Referring to a scoped identifier

```
reference :: scope* identifier
scope :: identifier "."
```

Dezyne defines the following scopes:

`enum` The field values of an enum:

```
enum result {TRUE, FALSE, ERROR};
```

are referenced to by using the enum type name as scope:

```
result.TRUE
```

`interface`
A type defined in an interface:

```
interface ihello
{
   enum result {TRUE, FALSE, ERROR};
}
```

can be used in a component, e.g., to define a variable:

```
ihello.result status = ihello.result.TRUE;
```

`behavior` All definitions in a behavior are local to that behavior and cannot be referenced from outside it[2],

`port` A port is an interface instance; events that are communicated over a port use the name of the port as their scope:

```
provides ihello p;
...
on p.hello (): p.world ();
```

`instance` An `instance` (or component instance), is an instance of a component. A port defined in a component

```
component hello
```

[2] This may change when Dezyne gains support for hierarchical behaviors, a.k.a. submachines.

```
{
  provides ihello p;
  requires ihello r;
}
```

can be referenced to by using the component instance name as their scope

```
component sys
{
  provides ihello sp;
  requires ihello sr;
  system
  {
    hello h;
    sp <=> h.p;
    h.r <=> sr;
  }
}
```

`namespace`
: Types defined in a namespace are referenced to by using the name of the namespace as their scope.

10.1.6 Comments

Dezyne supports single-line and multi-line comments very similar to `C`. Multi-line comments may be nested. All characters part of a comment are skipped by the parser.

```
/* This is an example of multi-line comment.
 * The line below is ignored also:
 * this component implements...
 */
component hello
{
  provides ihello p; // a single-line comment
}
```

10.2 Dezyne Files

Dezyne types, interfaces, and components are organized in files. A file, with extension '.dzn' by convention, may contain zero or more of type definitions, interfaces, and/or components.

The toplevel Dezyne program text is defined as follows:

```
text ::= root
root ::= (import | data-expression
          | namespace | type | interface | component)*
```

An interface can refer to a global type definition. A component can refer to types, interfaces and other components. An explicit `import` clause is needed when the referred information is defined in another file.

10.2.1 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 behavior, or the component system details.

```
import    ::= "import" (file-name "/")* file-name ";"
file-name ::= [a-zA-Z0-9_+.-]+
```

Note: That by convention the basename of the Dezyne file-name is used as the target language basename, however the target platform may impose further restrictions on a file-name.

By convention, Dezyne files use the extension `.dzn`. Some examples:

```
import file-name.dzn;
import ../global-types.dzn;
import some/directory/prefix/library.dzn;
```

An imported file may contain imports itself, these are also imported. When a file occurs twice in the resulting set of imports, it is expanded only once. This avoids introducing duplicate definitions. Mutually recursive imports are allowed (See Section 9.8 [Invoking dzn parse], page 57).

10.3 Types and Expressions

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

```
type      ::= bool / enum / subint / void
data-type ::= extern
```

`types` are used for event reply types, variables, function parameters, function, function return types, and function call arguments. `data-types` are used for event parameters, action arguments, and variables.

10.3.1 `void`

`void` is used for defining untyped events and functions, e.g.,

an event without reply value:

```
in void hello ();
```

a function without return value:

```
void foo ()
{
  world;
}
```

10.3.2 `bool`

Dezyne has a builtin boolean type `bool` with constants `false` and `true`.

Available boolean operators are:

`!b` Logical negation of a boolean expression,

`b1 && b2` Logical and of two boolean expressions,

`b1 || b2` Logical or of two boolean expressions, `b1 == b2` Equality of two boolean expressions,

`b1 != b2` Inequality of two boolean expressions,

where `b`, `b1`, and `b2` are boolean expressions.

It is used to define boolean events

```
in bool test ();
```

boolean variables

```
bool idle = true;
```

and parameters and functions

```
bool negate (bool input)
{
  return !input;
}
```

10.3.3 enum

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

```
enum   ::= "enum" identifier "{" fields "}" ";"
fields ::= identifier ("," identifier)* ","?
```

An example:

```
enum result {FALSE,TRUE,ERROR};
```

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

In expressions the enum values are referred to with a dot notation: `result.FALSE`.

Available enum operators are:

`e1 == e2` Equality of two enum expressions,

`e1 != e2` Inequality of two enum expressions,

`v.ERROR` A field-test: testing the value of an enum variable, denoted by `v.ERROR`, which is shorthand for `v == result.ERROR`

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

10.3.4 subint

The integer type is available in Dezyne in a restricted way[3]: only a finite contiguous subrange of integer numbers can be used. An explicit type definition is needed for each subset, where a `C`-like syntax is used.

```
subint  ::= "subint" identifier "{" range "}" ";"
range   ::= integer ".." integer
integer ::= ("-")? [0-9]+
```

An example:

```
subint int {-1..2};
```

[3] the subint definition allows range checking and prevents accidental unboundedness during model checking

where `subint` is a keyword. This defines the finite type `int` with possible values -1, 0, 1, and 2. Available integer operators are:

`comparison`

    i1

    i1 <= i2

    i1 >= i2

    i1 > i2

    i1 == i2

    i1 != i2

`i1 + i2,` Integer addition,

`i1 - i2` Integer subtraction,

where `i1` and `i2` denote integers.

> **note:** Integers of different `subint` types can be used in comparison, assignment, and function calls. The verifier will check that the resulting integer value is within the defined `subint` range.

10.3.5 `extern` data

Apart from `bool`, `enum`, and `int` types introduced above, also `extern` data types can be defined. An `extern` data type is defined as follows:

```
extern          ::= "extern" identifier data-expression ";"
data-expression ::= "$" (!"$")* "$"
```

The `data-expression` is a type expression in the target language.

No Dezyne-supported expressions are available for data types, apart from `data-expressions`. The content of the data-expression is passed to the target language verbatim.

For example, a `C++` string type could be defined as follows:

```
extern string $std::string$;
```

10.3.6 Expressions

Expressions in Dezyne are strictly typed.

```
expression ::= bool-expression | data-expression | enum-expression
             | integer-expression
```

> **Note:** The well-formedness check (See Chapter 11 [Well-formedness], page 94) verifies that expressions are of the correct type.

Bool Expressions

```
bool-expression ::= bool-literal
                    | action
                    | call
                    | field-test
```

```
                    | "!" bool-expression
                    | "(" bool-expression ")"
                    | bool-expression bool-operator bool-expression
                    | int-expression comparison-operator int-expression
bool-literal    ::= "false" | "true"
field-test      ::= enum-variable "." enum-field
bool-operator   ::= "==" | "!=" | "&&" | "||"
comparison-operator
                ::= "==" | "!=" | "<" | "<=" | ">" | ">="
```

where `action` and `call` are of type `bool`.

Enum Expressions

```
enum-expression ::= enum-literal
                    | action
                    | call
enum-literal    ::= enum "." field
```

where `action` and `call` are of the correct `enum` type.

Int Expressions

```
int-expression ::= int-literal
                    | action
                    | call
                    | int-expression int-operator int-expression
int-literal    ::= "-"? [0-9]+
int-operator   ::= "+" | "-"
```

where `action` and `call` are of a `subint` type.

10.4 Interfaces

Interfaces describe the interaction between two components: the events (or messages) that can and cannot be communicated, i.e., the interaction protocol.

```
interface ::= "interface" identifier "{" types-and-events behavior "}"
type-and-events
          ::= (type | event)*
behavior  ::= "behavior" "{" behavior-statement* "}"
```

Each event has a direction specified by the `in` or `out` keywords. An event labeled with `in` (`in`-event) is received by the implementation providing the interface. Conversely, an event labeled with `out` (`out`-event) is emitted by the implementation providing the interface. Note that from the point of view of an implementation requiring an interface the interpretation of `in` and `out` is inverted.

The interface protocol is specified in the `behavior` section.

```
$include <string>;$
interface ihello
{
  enum result {FALSE,TRUE,ERROR};
```

```
  extern string $std::string$;
  in result hello (string greeting);
  out void world ();
  behavior { ... }
}
```

10.4.1 Events

Events are messages or function calls and returns that are communicated between components.

```
event     ::= direction type identifier "(" parameter-list? ")" ";"
direction ::= "in" | "out"
parameter-list
          ::= parameter ( "," parameter)*
parameter ::= parameter-direction? data-type identifier
parameter-direction
          ::= "in" | "out" | "inout"
```

Some examples.

A void in-event called `e` with an empty parameter list:

```
in void e ();
```

a valued in-event called `e2`:

```
in enum_type e2 ();
```

a void in-event called `e3` with two data parameters

```
out void e3 (some_id in_id, out some_id out_id);
```

a void out-event called `e4` with a data parameter

```
out void e4 (some_string s);
```

Note: There are two restrictions on `out`-event definitions:

- `out`-events must be of type `void`, and
- `out`-events can only take `in` parameters.

10.4.1.1 Modeling Events

Apart from user-defined events, Dezyne has two special builtin events called `optional` and `inevitable`. These are called "modeling events" and are used in interface to specify *decoupled* behavior (See Section 10.4.3.3 [Using inevitable and optional], page 72).

10.4.2 Behavior

The `behavior` section of an interface defines the protocol of the interface. The protocol prescribes the causal relation between events and state. The behavior is akin to a state machine.

```
behavior ::= "behavior" "{" behavior-statement* "}"
behavior-statement
         ::= type | variable | function | declarative-statement
```

10.4.2.1 Behavior `variable`

The `behavior` variables define the state of the behavior. They are sometime referred to as *state variables.*

```
variable ::= (type identifier = expression
                | data-type identifier = data-expression
                | data-type identifier) ";"
```

where `type` and `expression` must match.

For example:

```
bool idle = true;
```

> **Note:** The `expression` used in the definition of a behavior variable must be a constant expression, i.e.: no `action`, `call` or `variable`-reference is allowed.

10.4.3 Declarative Statements

A trigger is prescribed by an interface to be handled by an implementation as is the condition under which it occurs. Collectively this is referred to as a declarative statement. The condition is expressed as a `guard`, the trigger in an `on`. The code that is executed when both the guard expression evaluates to `true` and the trigger occurs, is called the *imperative statement* (See Section 10.4.4 [Imperative Statements], page 73).

```
declarative-statement ::= guard | on | "{" declarative-statement "}"
```

The combination of all declarative statements leading up to a `imperative statement` is also called the declarative *prefix.* The combination of all `guards`, the `on` and the `imperative statement` is sometimes referred to as a *transition.*

```
transition   ::= prefix imperative-statement
prefix       ::= <guards> <on>
```

10.4.3.1 `on`

The `on` defines which trigger is to be handled:

```
on        ::= "on" trigger ("," trigger)* ":" statement
trigger   ::= event-name | "inevitable" | "optional"
statement ::= declarative-statement | imperative-statement | illegal
imperative-statement
          ::= action | assign | call | if | reply | return | variable
                | imperative-compound
                | empty-statement
illegal   ::= "illegal"
imperative-compound
          ::= "{" (imperative-statement ";")+ "}"
empty-statement
          ::= ";" | "{" "}"
```

For example:

```
on hello: {}
on inevitable: {world; idle = true;}
```

When two or more observably distinct imperative statements are specified for a certain trigger, the interface is said to behave *non-deterministic* with respect to the trigger. For example:

```
on hello: world;
on hello: cruel;
```

when the trigger `hello` is sent, the response can either be `world` or `cruel` but which one it will be cannot be predicted. Non-determinism in interfaces is allowed as long as it is *observable non-determinism*, i.e., after the trigger has returned the client should be able to know which state the interface is in. For example, this is not allowed:

```
on hello: {}
on hello: idle = true;
```

and will lead to a verification error (See Section 6.1 [Verification Checks and Errors], page 35).

Note:

- There must be exactly one imperative statement for every combination of `guard` and `on`,
- There can be only one `on` leading to an imperative statement.

10.4.3.2 guard

```
guard ::= "[" bool-expression "]" statement
```

For example:

```
[idle]  on hello: idle = false;
[!idle]
{
  on hello: idle = true;
  on inevitable: {world; idle = true;}
}
```

10.4.3.3 Using inevitable and optional

In interfaces, two *modeling* events may be used as abstract triggers, i.e. `inevitable` and `optional`:

```
on inevitable: imperative-statement;
on optional:   imperative-statement;
```

Where `inevitable` implies that if no other triggers occur, this trigger is guaranteed to occur, and `optional` implies that the trigger may or may never occur.

Note that an inevitable event is not allways guaranteed to occur, it is only inevitable in the absence of other events.

An example of an interface using both inevitable and optional.

```
interface intevitable_optional
{
  in bool hello ();
  in void bye ();
  out void world ();
```

```
    out void cruel ();

    behavior
    {
      enum status {IDLE, WORLD, CRUEL};
      status state = status.IDLE;

      [state.IDLE]
      {
        on hello: {state = status.WORLD; reply (true);}
        on hello: {state = status.CRUEL; reply (false);}
      }
      [state.WORLD] on inevitable: {state = status.IDLE; world;}
      [state.CRUEL]
      {
        on optional: {state = status.WORLD; cruel;}
        on bye: state = status.IDLE;
      }
    }
  }
```

In the interface above a reply value of `true` on `hello` informs the client sending the `hello` that the `world` can be waited on. However in case the reply value of `hello` is `false` and the client would sit there waiting for `cruel` to happen, they may sit there forever because `cruel` might never happen. This is what we refer to as a *deadlock*. To avoid this *deadlock* as a client, they must make sure that they can handle a `cruel` in case it does happen and that they have another way of making progress in case `cruel` never happens.

Conversely, the implementation of this interface may choose to perform the cruel always, never or intermittently after a `hello` followed by a `false`, but it must (being contractually required) always do a `world` after a `hello` followed by a `true`.

10.4.4 Imperative Statements

The imperative statement is the statement that will be executed when a guarded trigger occurs (see also See Section 10.4.3 [Declarative Statements], page 71).

10.4.4.1 `action`

When handling a trigger (a `in`-event), an interface can emit zero or more `out`-events. The event that follows a trigger is are referred to as an `action`.

```
action ::= event-name ";"
```

where `event-name` is the name of an `out`-event defined in the interface.

For example

```
world;
```

10.4.4.2 `assign`

The value of a previously defined variable can be updated using an `assign`:

```
assign ::= variable "=" expression ";"
```

For example:

```
idle = true;
idle = !b;
idle = negate (idle);
```

where `b` and `idle` are variables of type `bool`, `negate` is a function with one `bool` parameter and return-type `bool` (see See Section 10.4.4.3 [Function Call], page 74).

10.4.4.3 `call`

```
call          ::= identifier "(" argument-list ")"
argument-list ::= (expression ",")*
```

For example:

```
foo ();
bar (true, 12);
```

Note that the value returned by a `call` to a non-`void` `function` is not allowed to be ignored. Therefore in the example above both `foo` and `bar` must be `functions` of type `void`. By capturing the value in a variable definition or the use of an `assign` to an existing variable is the proper way to handle the return value:

```
bool b = bool_function ();
b = bool_function ();
```

Another way is to properly use a return value is in simple expressions, possibly combined with: ==, !=, !, &&, `||` (since 2.14.0)

```
if (bool_function ()) ...;
if (!bool_function ()) ...;
if (!bool_function () && b) ...;
if (enum_function () == result.FALSE) ...;
if (enum_function () != result.TRUE || b) ...;
reply (enum_function ());
reply (enum_function () != result.ERROR);
```

10.4.4.4 Empty Statement

The empty statement or `skip` statement defines for *nothing* to happen.

```
empty-statement ::=  ";" | "{" "}"
```

For example:

```
on hello: {}
on cruel: ;
```

10.4.4.5 `if`

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

```
if ::= ("if" "(" bool-expression ")" imperative-statement)
         | ("if" "(" bool-expression ")" imperative-statement
              "else" imperative-statement)
```

For example

```
if (idle)
```

```
    {world; idle = false;}
  else
    cruel;
```

Since 2.14.0, a valued `call` may be used in an `if`-expression.

For example:

```
if (bool_function ()) ...;
if (!bool_function ()) ...;
if (!bool_function () && b) ...;
if (enum_function () == result.TRUE) ...;
if (enum_function () != result.ERROR || b) ...;
```

Note that nested `if`s are allowed:

```
if (b1) if (b2) then-statement else else-statement
```

is interpreted as

```
if (b1)
{ if (b2) then-statement else else-statement }
```

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

> **Note:** In an interface, an `illegal` is not allowed as a then-statement or an else-statement, however the same can be expressed using a `guard`.

10.4.4.6 illegal

A trigger can be explicitly marked as being `illegal` in a certain state. In that case, `illegal` must be the only imperative statement for that trigger.

```
illegal ::= "illegal" ";"
```

For example:

```
on hello: illegal;
```

> **Note:** Since 2.14.0, a declarative-statement followed by an `illegal` can be completely omitted, since it has the same meaning. It is however still available for backwards compatibility.

10.4.4.7 reply

Define the value to be returned at the end of an `on` with a valued trigger.

```
reply ::= "reply" "(" expression ")" ";"
```

For example:

```
on hello: reply (true);
```

> **Note:** Reply does not mean "return", it merely defines the value that is returned when the `on` has finished executing. `reply` does not have to be the final imperative statement, however it must occur exactly once on every path through every sequence statements.

10.4.4.8 return

`return` is used to return program execution from the body of a function to the caller, possibly providing a value.

Implicitly returning from a `void` function is allowed. Also it is not required to use `return` as the last statement of a `void` function, i.e., an early return skipping over remaining statements is allowed.

```
return ::= "return" ";"
         | "return" expression ";"
```

For example:

```
void foo ()
{
  if (idle) return;
  world;
}

bool negate (bool b)
{
  return !b;
}
```

10.4.4.9 variable

Defining a local variable is syntactically identical to a behavior variable:

```
variable ::= (type identifier = expression
             | data-type identifier = data-expression
             | data-type identifier) ";"
```

For example:

```
bool b = true;
bool not_idle = !idle;
bool c = negate (idle);
```

10.4.5 Functions

A function can is used to name and reuse a sequence of imperative statements.

```
function ::= type identifier "(" parameter-list ")"
             "{" imperative-statement* "}"
```

For example:

```
void foo ()
{
  bye;
  cruel;
  world;
}

bool bar (bool b, int i)
{
```

```
    if (b)
      world;
    idle = i == 12;
    return idle;
  }
```

Functions are allowed to be called recursively. This includes mutual recursive functions (function `f` calling function `g` and vice versa). However only as long as every function involved in the recursion is *tail recursive*; which means that a recursive call is the last statement in the function.

10.5 Components

Components are the building blocks in a Dezyne. They allow composition into bigger components called system components.

A component has a list of ports and optionally a behavior or a system block.

```
component ::= "component" "{" port+ (behavior | system)? "}"
behavior  ::= "behavior" "{" behavior-statement* "}"
system    ::= "system" "{" system-statement* "}"
```

10.5.1 Ports

A port is an instance of an interface. A component has ports through which it interacts with other components. As such a port is one of the two end-points connecting two components.

```
port      ::= ("provides" interface-type identifier ";")
              | ("requires" qualifier? interface-type identifier ";")
qualifier ::= "external" | "injected";
```

The keyword `provides` indicates that a component implements all of the interface behavior.

The keyword `requires` indicates that a component relies on some or all of the interface behavior in its implementation.

For example:

```
provides ihello p;
requires ihello r;
requires external itimer t;  // (1)
requires injected ilogger l; // (2)
```

1) port to a component with a potential delay in its communication (see Section 10.5.1.2 [External], page 78)

2) port to a shared resource (see Section 10.6 [Systems], page 88)

Furthermore a component receives its `trigger`s from its surroundings through its ports. Note that a component `trigger` is either a `provides-in` or a `requires-out` event. If the component emits events over its ports they are referred to as `action`s. An `action` is either a `provides-out` or a `requires-in` event.

10.5.1.1 Injection

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

```
requires injected ilogger l;
```

This indicates that the port can be bound to a corresponding port residing at any level in the system hierarchy. An `injected` port is the exception to the one to one rule, i.e., it allows many ports to be connected to a single instance. For this reason `out` events are not allowed in interfaces which are `injected`.

See Section 10.6 [Systems], page 88, for a detailed description of the binding of injected ports.

10.5.1.2 external

The `external` keyword specifies that communication over a port may experience a delay. This may for instance be caused by the switch between execution contexts as in inter-process communication or the use of threads.

```
requires external itimer t;
```

During verification the delay on an `external` interface is experienced an additional interleaving of events that would otherwise not occur.

10.5.1.3 Race condition due to external delay

Component `remote_timer_proxy` illustrates how a delayed communication channel may cause a race condition leading to illegal behavior.

The implementation of component `remote_timer_proxy` is correct (no illegal behavior) 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 ();
  behavior
  {
    bool is_armed = false;
    [!is_armed] on create: is_armed = true;
    on cancel: is_armed = false;
    [is_armed] on inevitable: {timeout; is_armed = false;}
  }
}

component remote_timer_proxy
{
  provides itimer pp;
  requires external itimer rp;
  behavior
  {
    bool is_armed = false;
    on pp.create (s):
```

```
      [!is_armed] {rp.create (s); is_armed = true;}
    on pp.cancel (): {rp.cancel (); is_armed = false;}
    on rp.timeout ():
      [is_armed] {pp.timeout (); is_armed = false;}
  }
}
```

10.5.2 Component Behavior

The `behavior` section of a component defines its behavior.

```
behavior           ::= "behavior" "{" behavior-statement* "}"
behavior-statement ::= type | variable | function | async-port
                         | declarative-statement
```

A component behavior describes the communication or the exchange of events between a itself and other components in its environment connected to its ports. Each port is defined by a local name and a behavior refers to these ports by name when it relates `triggers` and `actions` (see also See Section 10.5.1 [Ports], page 77).

10.5.3 Async Ports

An async port is a `requires` of builtin type `dzn.async` which is defined in the behavior of a component.

```
async-port
  ::= "requires" "dzn.async" async-parameter-list? identifier ";"
async-parameter-list
  ::= "(" async-parameter ("," async-parameter)* ")"
async-parameter
  ::= data-type identifier
```

For example:

```
requires dzn.async (int i) defer;
```

The builtin `dzn.async` interface can be represented like this:

```
namespace dzn
{
  interface async (formals)
  {
    in void req (formals);
    in void clr ();
    out void ack (formals);

    behavior
    {
      bool idle = true;
      [idle]
        {
          on req: idle = false;
          on clr: {}
        }
```

```
      [!idle]
        {
          on clr: idle = true;
          on inevitable: {ack; idle = true;}
        }
    }
  }
}
```

Note: The use of `dzn.async` is discouraged for new programs, it will be replaced by `defer` in version 2.16, see the "wip-defer" branch.

10.5.4 Component Types

There are tree types of component:

`component`, `regular component`, or `leaf`
: A component that defines its implementation in its `behavior`,

`foreign`
: A component that defines only ports. Its behavior is said to be defined elsewhere. This is a placeholder for a component that is implemented by some other means, like another programming language (e.g. `C++`),

`system`
: A component that comprises other components in its `system` specification, See Section 10.6 [Systems], page 88.

10.5.4.1 A Leaf Component

Every component in Dezyne is a leaf component, unless it is a system component. The following component implements one interface and a straightforward behavior section:

```
component hello
{
  provides ihello p;
  requires ihello r;
  requires itimer t;
  behavior
  {
    on p.hello (): t.create ();
    on t.timeout (): r.hello ();
    on r.world (): p.world ();
  }
}
```

10.5.4.2 A Foreign Component

This component does not reveal its implementation in Dezyne under this name. It represents a component implemented elsewhere. It may be implemented in another programming language, or it is implemented in Dezyne without exposing any of its implementation details.

```
component timer
{
  provides itimer t;
}
```

10.5.4.3 A System Component

A component `timer_system` decomposed into two components `ihello` and `timer` where these components are connected via their ports.

```
component timer_system
{
  provides ihello p;
  requires ihello r;
  system
  {
    hello h;
    timer t;
    p <=> h.p;
    h.t <=> t.t;
    h.r <=> r;
  }
}
```

10.5.5 Component Declarative Statements

For a component behavior, the list of declarative statements is extended with `blocking` (See Section 10.5.5.2 [Blocking], page 82). So we get:

```
component-declarative-statement ::= declarative-statement | blocking
```

Thus, in a component, the “declarative prefix” includes blocking such that the prefix consists of three elements.

```
transition  ::= prefix imperative-statement
prefix      ::= <blocking?> <guards> <on>
```

10.5.5.1 Component `on`

Similar to an interface, in a component the `on` defines which trigger is to be handled. Component triggers, however, belong to a port and carry formal parameters:

```
on          ::= ("on" triggers ":" statement)
                  | ("on" illegal-triggers ":" illegal)
triggers    ::= trigger ("," trigger)*
trigger     ::= port-name "." event-name "(" formal-list? ")"
formal-list ::= formal ("," formal)*
formal      ::= identifier | (identifier formal-binding)
statement   ::= declarative-statement | imperative-statement | illegal
imperative-statement
            ::= action | assign | call | if | reply | return | variable
                  | imperative-compound
                  | empty-statement
illegal     ::= "illegal"
imperative-compound
            ::= "{" (imperative-statement ";")* "}"
empty-statement
            ::=
```

```
illegal-triggers
            ::= illegal-trigger ("," illegal-trigger)*
illegal-trigger
            ::= port-name "." event-name
```

The `formal-list` to be used is defined by the parameters of the event definition in the interface. Their relation is position-based. Formal parameters may introduce another name than specified in the `event` definition in the interface.

For example:

```
on p.hello (greeting): w.hello (greeting);
on p.cruel, r.hello: illegal; // Note this is optional since 2.14.0.
```

When two or more imperative statements are specified for a certain trigger, the component is said to be *non-deterministic*. For example:

```
on p.hello (): w.hello ();
on p.hello (): ;
```

non-determinism in components is not allowed and will lead to a verification error (See Section 6.1 [Verification Checks and Errors], page 35).

The `formal-binding` is a feature for `blocking`, See Section 10.5.5.3 [Formal Binding], page 83.

10.5.5.2 `blocking`

The `blocking` keyword is a declarative statement that can be used in a component.

```
blocking ::= "blocking" statement
```

Using `blocking` requires an explicit `reply`. It can only be used in a component. If the `reply` is omitted for the `blocking` `trigger`, the imperative statement of another `trigger` must perform the `reply` for the blocked port. Thus, time and value of a blocked port reply depend on another `trigger`. `blocking` may be used once in the declarative prefix.

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

Guards or `on` is commutative with respect to blocking. If `blocking` appears before a guard or `on` it applies to the imperative statement after the guard or `on`.

Note:

- `blocking` may only be used in components with a single `provides` port. This limitation may be lifted in a future release.
- Systems containing `blocking` component instances must be contained in a thread-safe shell (see Section 8.2 [Thread-safe Shell], page 49).
- A `blocking` component may only appear as the top component in a system. This restriction will be lifted in version 2.15, or see the “wip-blocking” branch in git.

For example:

```
on trigger (): blocking imperative-statement;         (1)
blocking on trigger (): imperative-statement;         (2)
on trigger (): blocking [guard] imperative-statement; (3)
on trigger ():
```

```
{
  blocking [guard] imperative-statement1;              (4)
  [guard] imperative-statement2;
}
```

Explanation:

2) The `blocking` keyword applies to the `imperative-statement` following `on trigger:`. This form is semantically equivalent to 1).

3) The `blocking` keyword applies to the `imperative-statement` following `[guard]`. This form is semantically equivalent to `on trigger (): [guard] blocking imperative-statement;"`

4) The `blocking` keyword applies to `imperative-statement1`. It does *not* apply to `imperative-statement2`.

10.5.5.3 Formal Binding

A formal binding *binds* a member variable to an `out` or `inout` formal parameter. At the moment of the `reply`, the value of the bound member variable is assigned to the formal parameter.

```
trigger        ::= port-name "." event-name "(" formal-list? ")"
formal-list    ::= formal ("," formal)*
formal         ::= identifier | (identifier formal-binding)
formal-binding ::= "<-" identifier
```

The `identifier` in `formal-binding` must be a member variable of the component.

For example:

```
extern int $int$;
component blocking_binding
{
  provides ihello h;
  requires iworld w;

  behavior
  {
    int g = $123$;
    bool busy = false;
    [!busy] on h.hello (n <- g): blocking {w.hello (); busy = true;}
    [busy] on w.world (): {g = $456$; h.reply (); busy = false;}
    [busy] on w.cruel (): {h.reply (); g = $456$; busy = false;}
  }
}
```

in the case of `w.world` the assignment of `g = $456$` before the release of the blocked thread by `h.reply ()` ensures that parameter `n` returns with value `456`. However in the case of `w.cruel` the caller of `h.hello` receives `123` via parameter `n`.

> **Note:** The intent is to simplify this specific behavior in the future when data flow verification is added.

10.5.5.4 Joining Activities

Component `join` illustrates the use of `blocking` in synchronizing a `starter` with the activities of two `runners`.

```
interface starter
{
  in void start_and_wait ();
  behavior
  {
    on start_and_wait: {}
  }
}

interface runner
{
  in void start ();
  out void finished ();
  behavior
  {
    bool running = false;
    on start: running = true;
    [running] on inevitable: {running = false; finished;}
  }
}

component join
{
  provides starter ref;
  requires runner one;
  requires runner two;

  behavior
  {
    subint Runners {0..2};
    Runners running = 0;

    blocking on ref.start_and_wait ():
      {running = 2; one.start(); two.start ();}
    [running != 1] on one.finished (), two.finished ():
      running = running - 1;
    [running == 1] on one.finished (), two.finished ():
      {running = 0; ref.reply ();}
  }
}
```

10.5.6 Component Imperative Statements

10.5.6.1 Component action

When handling the response of a trigger, a component can send one or more events over its ports. The sending of a provides-out-event or a requires-in event is referred to as an action.

```
action        ::= port-name "." event-name "(" argument-list? ")"
argument-list ::= expression ("," expression)*
```

where port-name is the name of a port defined in the component, and event-name is the name of an event defined in the interface associated with the port.

Note that the event in an action statement must be of type void. For a valued action the reply value may not be ignored. A variable definition or an assign are the appropriate ways to handle a reply value:

```
bool b = r.bool_event ();
b = r.bool_event2 ();
```

or it can be used directly in a simple expression, optionally in combination with ==, !=, !, &&, or || (since 2.14.0)

```
if (r.bool_event ()) ...;
if (!r.bool_event ()) ...;
if (!r.bool_event () && b) ...;
if (r.enum_event () == result.FALSE) ...;
if (r.enum_event () != result.TRUE || b) ...;
reply (r.enum_event ());
reply (r.enum_event () != result.ERROR);
```

Note: Only one action or call may be used in such an expression.

10.5.6.2 Component if

In a component an illegal can be used as an imperative statement in the branch of an if as any other imperative statement (See Section 10.5.6.3 [Component Illegal], page 86).

```
if ::= ("if" "(" bool-expression ")" then-statement)
         | ("if" "(" bool-expression ")" then-statement
            "else" else-statement)
then-statement := imperative-statement | illegal
else-statement := imperative-statement | illegal
```

For example:

```
if (error) illegal;
```

Since 2.14.0, one valued action or valued call may be used in an if-expression.

For example:

```
if (r.bool_event ()) ...;
if (!r.bool_event ()) ...;
if (!r.bool_event () && b) ...;
if (r.enum_event () == result.FALSE) ...;
if (r.enum_event () != result.TRUE || b) ...;
if (bool_function ()) ...;
if (!bool_function ()) ...;
```

```
if (!bool_function () && b) ...;
if (enum_function () == result.TRUE) ...;
if (enum_function () != result.ERROR || b) ...;
```

10.5.6.3 Component `illegal`

A `trigger` can be explicitly marked as `illegal`. In that case, `illegal` must be the only imperative statement for that trigger.

```
on      ::= "on" illegal-triggers ":" illegal
illegal-triggers
        ::= illegal-trigger ("," illegal-trigger)*
illegal-trigger
        ::= port-name "." event-name
illegal ::= "illegal" ";"
```

Note that in this case the `trigger`'s formal parameter list may be omitted.

For example:

```
on p.hello,r,world: illegal;
```

> **Note:** A trigger with an `illegal` response can also be omitted since an `illegal` response is the default behavior for every `trigger`.

In a component an `illegal` can be used as an imperative statement in the branch of an `if` as any other imperative statement (See Section 10.5.6.2 [Component If], page 85).

10.5.6.4 Component `reply`

A valued trigger event requires an appropriate return value in its response handling, the `reply` only determines the value not the moment of returning it:

```
reply (valued_expression);
```

`reply` is also used to release a blocked call (See Section 10.5.5.2 [Blocking], page 82), like so:

```
port.reply ();
port.reply (expression);
```

10.5.7 Multiple Provides Ports

A component is not limited to a single provides port, it is allowed to offer multiple interfaces simultaneously. When a component provides multiple ports it can receive `in`-events via any of its provides ports. As a result the interface behaviors of the provides ports are effectively interleaved and the component is expected to handle that appropriately.

When providing multiple ports, two restrictions hold for the component behavior:

`V-fork`
: Within the handling of an `in`-event of a provides port, it is not allowed to directly post an `out`-event on another provides port.

`Y-fork`
: 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 rationale behind both limitations is that if V-forking or Y-forking would be allowed that it potentially leads to behavior which is beyond the scope of single component verification.

Violating of any of these restrictions is reported as a compliance error.

Here are examples of the two types of forking that lead to a compliance error:

```
interface ihello
{
  in void hello();
  out void world();
  behavior
  {
    on hello: {}
    on optional: world;
  }
}

component v_fork
{
  provides ihello left;
  provides ihello right;
  behavior
  {
    on left.hello():
    {
      right.world(); //is non-compliant with interface(s) of provides port(s)
    }
    on right.hello(): {}
  }
}

component y_fork
{
  provides ihello left;
  provides ihello right;
  requires ihello r;
  behavior
  {
    on left.hello(), right.hello(): {}
    on r.world():
    {
      left.world();
      right.world(); //is non-compliant with interface(s) of provides port(s)
    }
  }
}
```

10.6 Systems

A `system` component, or `system` is a component which is composed from one or more sub components. The `system` block instantiates each of the sub components and either connects their ports together or exposes them as its own, such that all ports are bound.

```
system-component ::= "component" "{" port* system "}"
system           ::= "system" "{" system-statement* "}"
system-statement ::= (instance | binding)
instance         ::= component-name identifier ";"
binding          ::= end-point "<=>" end-point ";"
end-point        ::= port-name
                   | wildcard
                   | (instance-name "." port-name)
                   | (instance-name "." wildcard)
wildcard         ::= "*"
```

Note: A binding can have only one `wildcard`, See Section 10.6.2.1 [Using Injection], page 89.

For example:

```
interface i
{
  in void event();
  behavior {}
}

component c
{
  provides i pp;
  requires i rr;
}

component top_middle_bottom
{
  provides i p;
  requires i r;
  system
  {
    c top;
    c middle;
    c bottom;
    p <=> top.pp;
    top.rr <=> middle.pp;
    middle.rr <=> bottom.pp;
    bottom.rr <=> r;
  }
}
```

The system description shows the instantiation of the two component instances `ci1` and `ic2` and two connections or bindings between ports.

10.6.1 Component Instances

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

```
instance ::= component-name identifier ";"
```

The component definition of `component-name` has to be available, potentially through an `import`.

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

```
hello h1;
hello h2;
```

10.6.2 Binding

Communication between components is achieved through component ports. The lines of communication are established by binding ports:

```
binding   ::= end-point "<=>" end-point ";"
end-point ::= port-name
            | wildcard
            | (instance-name "." port-name)
            | (instance-name "." wildcard)
wildcard  ::= "*"
```

Note that bindings are symmetrical, i.e., left and right `end-points` can be exchanged. Communication is restricted to ports of the same (interface) type. Moreover the communication 'direction' has to be compatible. There are two cases:

- Two sub components communicating: always a `provides` port binds to a `requires` port, like in `top.rr <=> middle.pp` in the `top_middle_bottom` system example above.
- In the case of port forwarding, where a sub-component port is exposed as a system port, the directions of the ports must be the same, like in `p <=> top.pp` and `bottom.rr <=> r` in the `top_middle_bottom` system example above.

10.6.2.1 Using Injection

Binding of `injected` ports is done at a higher system level (see Section 10.5 [Components], page 77). A wild-card character (`*`) is used to achieve the binding of the `provides` port of a single instance to all `injected requires` ports.

Let's take a logging interface as an example:

```
interface ilog
{
  ...
}
component logger
{
  provides ilog log;
  ...
}
```

Suppose a lot of components require logging:

```
...
component some_component12
{
  provides some_interface12 p;
  requires injected ilog l;
  ...
}
component some_component13 {
  provides some_interface13 p;
  requires injected ilog l;
  ...
}
...
```

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

```
component some_system
{
  ...
  system
  {
    logger clog;
    ...
    some_component12 c12;
    some_component13 c13;
    ...
    clog.log <=> *;
  }
}
```

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

```
component some_sub_system
{
  ...
  system {
    ...
    some_component12 c12;
    some_component13 c13;
    ...
  }
}
```

and use the wild-card binding for that sub system:

```
component some_system
{
  ...
  system
  {
```

```
      logger clog;
      some_sub_system subsys;
      ...
      clog.log <=> subsys.*;
    }
  }
```

10.7 Namespaces

All component, interface, and type definitions are defined in a `namespace`, which provides name scoping. The scope is used as a prefix when referring to the name from another scope.

```
namespace      ::= "namespace" identifier "{" namespace-root "}"
namespace-root ::= (namespace | type | interface | component)*
```

For example:

```
namespace space
{
  extern string $std::string$;
  interface ihello
  {
     enum result {FALSE, TRUE, ERROR};
     in result hello (string s);
     out void world ();
     behavior
     {
       on hello (s): reply (result.TRUE);
    }
  }
}
```

10.7.1 Namespace Extension

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 space
{
  extern string $std::string$;
  interface ihello { ...  }
}
```

is equivalent to

```
namespace space
{
  extern string $std::string$;
}
namespace space
```

```
{
  interface ihello { ...  }
}
```

10.7.2 Referencing

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

Within its own namespace the short name `string` is also accepted.

In complex cases it may be necessary to refer to the default *global* namespace which has an empty name; this results in a namespace prefix 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();
    behavior {
      on e: {a; reply (Bool.T); }
    }
  }
}
namespace inner {
  namespace foo {
    interface I {
      enum Bool {f,t};
      in Bool e();
      out void a();
      behavior { }
    }
  }
  component space {
    provides foo.I inner;
    provides .foo.I fooi;
    behavior {
      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;
    behavior {
      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.

11 Well-formedness

The syntax as defined in Chapter 10 [Dezyne Language Reference], page 63, leaves room for certain combinations and variations that would lead to Dezyne code that cannot be translated to an mCRL2 process algebra specification. This chapter describes a collection of well-formedness checks that are defined on top of the syntax.

Apart from the syntax checks performed by the parser, five additional categories of checks can be identified:

definition checks
: Upon failure, these produce a `undefined identifier` error,

parameter checks
: Upon failure, these produce a `count mismatch` error,

type checks
: Upon failure, these produce a `type-mismatch` error,

shadowing checks
: Upon failure, these produce a `shadowing` error,

well-formedness checks
: Semantic checks, a.k.a. "well-formedness" checks. Upon failure, these produce a `well-formedness` error.

The first four categories are common programming errors and should not need additional explanation. The last category—the well-formedness checks—are unique to Dezyne and are described in this chapter.

11.1 Well-formedness Checks Categories

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

Top level Interface, event and component definitions.

Directional
: `trigger`s and `action`s are expected at different places depending on the direction of their `event`.

Nesting The imperative part of the language (`assign`s, `action`s, function `call`s) are only allowed in an imperative statement or in a function body,

Mixing The use of statements within `compound`s is restricted,

Reply The usage of `reply`,

Valued Actions and Calls
: The use of non-`void` `action`s and `call`s,

Injection The use of `injected` ports,

Functions A function body should be imperative, and have a well-defined return.

Data Parameters
: The use of data parameters,

Injection The use of `injected` ports,

System All ports should be bound correctly.

> **Note:** A trigger is an event that occurs and is prefixed by `on` in the behavior, an action is an event that is emitted inside the imperative body of a trigger.

11.2 List of Well-formedness Checks

The well-formedness checks in alphabetical order:

See Section 11.7.2 [Must specify provides-port with reply on out-trigger], page 107,
See Section 11.5.4 [Nested blocking used], page 101,
See Section 11.5.3 [Nested on used], page 101,
See Section 11.3.3 [Out-event must be void], page 96,
See Section 11.12.3 [Port is bound more than once], page 118,
See Section 11.12.1 [Port not bound], page 117,
See Section 11.12.2 [Port not bound – of instance], page 117,
See Section 11.12.8 [System composition is recursive], page 123,
See Section 11.11.1 [Type mismatch – parameter expected extern], page 115.

11.3 Well-formedness – Top level

These checks are concerned about interface, event and component definitions.

11.3.1 Interface must define an `event`

Completely "passive" interfaces are not allowed; at least one `in`-event or `out`-event is required:

```
interface interface_without_event
{
  behavior {}
}
```

This results in the following error message:

```
interface-without-event.dzn:1:1: error: interface must define an event
```

11.3.2 Interface must define a `behavior`

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

```
interface interface_without_behavior
{
  in void hello ();
}
```

This results in the following error message:

```
interface-without-behavior.dzn:1:1: error: interface must define a
    behavior
```

11.3.3 `out`-event must be `void`

Only `in`-events can have a non-`void` type.

```
interface valued_out_event
{
  out bool world ();
  behavior {}
}
```

This results in the following error message:

```
valued-out-event.dzn:3:3: error: out-event 'world' must be void, found
    'bool'
```

11.3.4 Component with behavior must have a trigger

Any component with a `behavior` 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 supported.

An example:

```
interface iworld
{
  out void world ();
  behavior {}
}

component component_provides_without_trigger
{
  provides iworld p;
  behavior {}
}
```

This results in the following error message:

```
component-provides-without-trigger.dzn:7:1: error: component with
    behavior must have a trigger
```

Another example:

```
interface ihello
{
  in void hello ();
  behavior {}
}

component component_requires_without_trigger
{
  requires ihello r;
  behavior {}
}
```

This results in the following error messages:

```
component-requires-without-trigger.dzn:7:1: error: component with
    behavior must define a provides port
component-requires-without-trigger.dzn:7:1: error: component with
    behavior must have a trigger
```

11.3.5 Component with behavior must define a provides port

Any component with a `behavior` specification must have a `provides` port through which the component is activated.

An example:

```
interface iworld
{
```

```
  in void hello ();
  behavior {}
}

component component_without_provides
{
  requires iworld r;
  behavior {}
}
```

The examples results in the following error messages:

```
component-without-provides.dzn:7:1: error: component with behavior must
    define a provides port
component-without-provides.dzn:7:1: error: component with behavior must
    have a trigger
```

11.4 Well-formedness – Directional

`triggers` and `actions` are expected at different places, depending on the direction of their `event`.

11.4.1 Cannot use event as action

In an interface this indicates the an `in`-event–that can only be used as a `trigger`—is used as an `action` in the imperative body of an `on`.

```
interface interface_trigger_used_as_action
{
  in void hello ();
  behavior
  {
    on hello: hello;
  }
}
```

This results in the following error message:

```
interface-trigger-used-as-action.dzn:6:15: error: cannot use in-event
    'hello' as action
interface-trigger-used-as-action.dzn:3:3: info: event 'hello' defined
    here
```

in a component this indicates that either it is an `in`-event of a `provides` interface, or an `out`-event of a `requires` interface that is used as an `action` in the imperative body of an `on`.

```
interface ihello
{
  in void hello ();
  out void world ();
  behavior {}
}
```

```
component component_trigger_used_as_action
{
  provides ihello p;
  requires ihello r;
  behavior
  {
    on p.hello ():
    {
      p.hello ();
      r.world ();
    }
  }
}
```

This results in the following error messages:

```
component-trigger-used-as-action.dzn:16:7: error: cannot use provides
    in-event 'hello' as action
component-trigger-used-as-action.dzn:10:3: info: port 'p' defined here
component-trigger-used-as-action.dzn:3:3: info: event 'hello' defined
    here
component-trigger-used-as-action.dzn:17:7: error: cannot use requires
    out-event 'world' as action
component-trigger-used-as-action.dzn:11:3: info: port 'r' defined here
component-trigger-used-as-action.dzn:4:3: info: event 'world' defined
    here
```

11.4.2 Cannot use event as trigger

In an interface this indicates an `out`-event is used as a `trigger`.

```
interface interface_action_used_as_trigger
{
  out void world ();
  behavior
  {
    on world: {}
  }
}
```

This results in the following error message:

```
interface-action-used-as-trigger.dzn:6:8: error: cannot use out-event
    'world' as trigger
interface-action-used-as-trigger.dzn:3:3: info: event 'world' defined
    here
```

in a component this indicates that either it is an `out`-vent of a `provides` interface, or an `in`-event of a **requires** interface that is used as a `trigger`.

```
interface ihello
{
```

```
  in void hello ();
  out void world ();
  behavior {}
}

component component_action_used_as_trigger
{
  provides ihello p;
  requires ihello r;
  behavior
  {
    on p.world (): {}
    on r.hello (): {}
  }
}
```

This results in the following error messages:

```
component-action-used-as-trigger.dzn:14:8: error: cannot use provides
    out-event ‘world’ as trigger
component-action-used-as-trigger.dzn:10:3: info: port ‘p’ defined here
component-action-used-as-trigger.dzn:4:3: info: event ‘world’ defined
    here
component-action-used-as-trigger.dzn:15:8: error: cannot use requires
    in-event ‘hello’ as trigger
component-action-used-as-trigger.dzn:11:3: info: port ‘r’ defined here
component-action-used-as-trigger.dzn:3:3: info: event ‘hello’ defined
    here
```

11.5 Well-formedness – Nesting

Dezyne statements are either *declarative* or *imperative*. One or more declarative statements must be used as a prefix to the imperative statement (See Section 10.4.3 [Declarative Statements], page 71). Imperative statements cannot be used without a “declarative prefix”, and declarative statements cannot be used inside an imperative statement

11.5.1 `assign` outside `on`

An `assign` occurred outside the scope of a declarative context:

```
interface assign_outside_on
{
  in void hello ();
  behavior
  {
    bool b = true;
    [true] b = false;
  }
}
```

This results in the following error message:

```
assign-outside-on.dzn:7:12: error: assign outside on
```

11.5.2 action outside on

An `action` occurred outside the scope of a declarative context:

```
interface action_outside_on
{
  out void world ();
  behavior
  {
    [true] world;
  }
}
```

This results in the following error message:

```
action-outside-on.dzn:6:12: error: action outside on
```

11.5.3 Nested on used

```
interface nested_on
{
  in void hello ();
  in void cruel ();
  out void world ();
  behavior
  {
    on hello: on cruel: world;
  }
}
```

This results in the following error message:

```
nested-on.dzn:8:15: error: nested on used
nested-on.dzn:8:5: info: within on here
```

11.5.4 Nested blocking used

```
interface ihello
{
  in void hello ();
  behavior
  {
    on hello:;
  }
}

component nested_blocking
{
  provides ihello p;
  behavior
  {
```

```
    blocking on p.hello (): [true] blocking p.reply ();
  }
}
```

This results in the following error message:

```
nested-blocking.dzn:15:36: error: nested blocking used
nested-blocking.dzn:15:5: info: within blocking here
```

11.5.5 Cannot use `blocking` in an interface

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

```
interface blocking_in_interface
{
  in void hello ();
  behavior
  {
    blocking on hello:;
  }
}
```

This results in the following error message:

```
blocking-in-interface.dzn:6:5: error: cannot use blocking in an
    interface
```

11.5.6 Cannot use `blocking` with multiple `provides` ports

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

```
interface ihello
{
  in void hello ();
  behavior
  {
    on hello:;
  }
}

component blocking_multiple_provides
{
  provides ihello left;
  provides ihello right;
  behavior
  {
    blocking on left.hello (): left.reply ();
  }
}
```

This results in the following error message:

```
blocking-multiple-provides.dzn:16:5: error: cannot use blocking with
```

```
multiple provide ports
```

11.6 Well-formedness – Mixing

A behavior description introduces a sequence of statements. A statement itself can be a `compound`, 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.

11.6.1 Declarative statement expected

If a `compound` statement starts with a declarative statement, all other statements must be declarative statements.

```
interface mixing_declarative
{
  in void hello ();
  behavior
  {
    [true]
    {
      on hello: {}
      if (true);
    }
  }
}
```

This results in the following error messages:

```
mixing-declarative.dzn:9:7: error: declarative statement expected
mixing-declarative.dzn:9:7: error: if outside on
mixing-declarative.dzn:9:16: error: imperative compound outside on
```

11.6.2 Imperative statement expected

If a `compound` statement starts with an imperative statement, all other statements must be imperative statements.

```
interface mixing_imperative
{
  in void hello ();
  behavior
  {
    bool b = true;
    on hello:
    {
      b = false;
      [b] b = false;
    }
  }
}
```

This results in the following error message:

```
mixing-imperative.dzn:10:7: error: imperative statement expected
```

11.6.3 Cannot use `otherwise` guard more than once

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:

```
interface second_otherwise
{
  in void hello ();
  in void cruel ();
  in void world ();
  behavior
  {
    bool b = true;
    [b] on hello: b = false;
    [otherwise] on world: b = true;
    [otherwise] on cruel: {}
  }
}
```

This results in the following error message:

```
second-otherwise.dzn:11:5: error: cannot use otherwise guard more than
    once
second-otherwise.dzn:10:5: info: first otherwise here
```

11.6.4 Cannot use `otherwise` guard with non-guard statements

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-`guard`. So:

```
interface otherwise_without_guard
{
  in void hello ();
  in void cruel ();
  behavior
  {
    on hello: {}
    [otherwise] on cruel: {}
  }
}
```

This results in the following error message:

```
otherwise-without-guard.dzn:8:5: error: cannot use otherwise guard with
    non-guard statements
otherwise-without-guard.dzn:7:5: info: non-guard statement here
```

11.6.5 Cannot use `illegal` with imperative statements

An `illegal` statement must occur on its own; no other `action`s or `assign`s are allowed. That also applies if the `illegal` occurs in a nested `compound`:

```
interface imperative_illegal
{
  in void hello ();
  behavior
  {
    bool b = false;
    on hello:
    {
      b = true;
      illegal;
    }
  }
}
```

This results in the following error message:

```
imperative-illegal.dzn:10:7: error: cannot use illegal with imperative
    statements
imperative-illegal.dzn:9:7: info: imperative statement here
```

In a component, using an `illegal` within a conditional statement *is* allowed. Also the condition may be accompanied by other `actions`, e.g:

```
interface ihello
{
  in void hello();
  behavior
  {
    on hello: {}
  }
}

component component_if_illegal
{
  provides ihello p;
  behavior
  {
    bool b = true;
    on p.hello():
    {
      b = false;
      if (b)
        illegal;
    }
  }
}
```

11.6.6 Cannot use `illegal` in `if`-statement

In an interface, a `trigger` 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`. An example:

```
interface interface_if_illegal
{
  in void hello ();
  behavior
  {
    bool b = false;
    on hello:
    {
      if (b)
        illegal;
    }
  }
}
```

This results in the following error message:

```
interface-if-illegal.dzn:10:9: error: cannot use illegal in if-statement
```

11.6.7 Cannot use `illegal` in function

In an interface, a `trigger` 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:

```
interface interface_function_illegal
{
  in void hello ();
  behavior
  {
    void f ()
    {
      illegal;
    }
    on hello: f ();
  }
}
```

This results in the following error message:

```
interface-function-illegal.dzn:8:7: error: cannot use illegal in
    function
```

11.7 Well-formedness – Reply

A `reply` is required in the handling of a valued (i.e. non-`void`) trigger. It is also required in case a `trigger` (which in this case 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.

11.7.1 Cannot use `requires` port in `reply`

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

```
interface ihello
{
  in void hello ();
  out void world ();
  behavior
  {
    on hello:;
    on inevitable: world;
  }
}

component requires_port_repyl
{
  provides ihello p;
  requires ihello r;
  behavior
  {
    blocking
    {
      on p.hello (): p.reply ();
      on r.world (): r.reply ();
    }
  }
}
```

This results in the following error message:

```
requires-port-reply.dzn:21:22: error: cannot use requires port 'r'
    in reply
requires-port-reply.dzn:15:3: info: port defined here
```

11.7.2 Must specify `provides`-port with `reply` on out-trigger

When a `reply` is used in the body of a `requires-out` trigger, and the component has multiple provides ports, the `reply` must specify which port it belongs to:

```
interface ihello
{
  in bool hello ();
  behavior
  {
    on hello: reply (true);
    on hello: reply (false);
  }
}
```

```
interface iworld
{
  in void hello ();
  out void world ();
  behavior
  {
    on hello: world;
  }
}

component requires_reply_needs_provides_port
{
  provides ihello left;
  provides ihello right;
  requires iworld r;
  behavior
  {
    on left.hello (): reply (true);
    on right.hello (): reply (false);
    on r.world (): reply ();
  }
}
```

This results in the following error message:

```
requires-reply-needs-provides-port.dzn:30:20: error: must specify a
    provides-port with reply on requires out-trigger: ‘r.world’
```

11.7.3 Must specify `provides`-port with `reply`

When a `reply` is used in the body of a function, and the component has multiple provides ports, the `reply` must specify which port it belongs to:

```
interface ihello
{
  in bool hello ();
  behavior
  {
    on hello: reply (true);
    on hello: reply (false);
  }
}

component function_reply_needs_provides_port
{
  provides ihello left;
  provides ihello right;
  behavior
  {
```

```
    void f (bool b)
    {
      left.reply (b);
    }
    void g (bool b)
    {
      reply (b);
    }
    on left.hello (): f (true);
    on right.hello (): g (false);
  }
}
```

This results in the following error message:

```
function-reply-needs-provides-port.dzn:23:7: error: must specify a
    provides-port with reply
```

11.8 Well-formedness – Valued Actions and Calls

Both `actions` and function `calls` 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 function `calls` (at least the functions that contain `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 or variable definition, or (since version 2.14.0) in simple `if`-expressions and `reply`-expressions.

An extra restriction to this rule is put on the initial value of a global variable in a behavior. Such an expression can not contain `actions` or function `calls` at all, since `actions` are only allowed within an `on`.

11.8.1 `action` in member variable initializer

An `action` is used in the initial value of a member variable.

```
interface ihello
{
  in bool hello ();
  behavior
  {
    on hello: reply (true);
  }
}

component action_in_member_definition
{
  provides ihello p;
  requires ihello r;
  behavior
  {
```

```
    bool b = r.hello ();
  }
}
```

This results in the following error message:

```
action-in-member-definition.dzn:16:14: error: action in member variable
    initializer
```

11.8.2 action used in a complex expression

An action is used in a complex expression.

```
interface ihello
{
  in bool hello ();
  in bool cruel ();
  behavior
  {
    on hello: reply (true);
    on cruel: reply (true);
  }
}

component action_in_complex_expression
{
  provides ihello p;
  requires ihello r;
  behavior
  {
    bool b = true;
    void f (bool b)
    {
      reply (b);
    }
    on p.hello ():
      if (r.hello () && r.cruel ())  // not allowed
        reply (false || r.cruel ()); // not allowed
      else if (r.hello () || true)
        reply (r.hello () == true);
      else
        reply (!r.hello ());
    on p.cruel (): f (r.hello ());   // not allowed
  }
}
```

This results in the following error messages:

```
action-in-complex-expression.dzn:24:25: error: action used in a complex
    expression
action-in-complex-expression.dzn:25:25: error: action used in a complex
    expression
```

```
action-in-complex-expression.dzn:30:23: error: action used in a complex
    expression
```

11.8.3 `call` in member variable initializer

A function `call` is used in the initial value of a member variable.

```
interface ihello
{
  in bool hello ();
  behavior
  {
    on hello: reply (true);
  }
}

component call_in_member_definition
{
  provides ihello p;
  requires ihello r;
  behavior
  {
    bool f () {return false;}
    bool b = f ();
  }
}
```

This results in the following error message:

```
call-in-member-definition.dzn:17:14: error: call in member variable
    initializer
```

11.8.4 `call` used in a complex expression

A function `call` is used in a complex expression.

```
enum result {OK, FAIL};
interface call_in_complex_expression
{
  in result hello ();
  behavior
  {
    result f ()
    {
      return result.OK;
    }

    result g (result r)
    {
      return r;
    }
```

```
    result r = result.OK;
    on hello:
    {
      bool b = f () != result.OK;   // not allowed
      result r = g (f ());          // not allowed
      if (f () == g (result.OK))    // not allowed
        reply (result.OK);
      else if (f () != result.OK)
        reply (f ());
    }
  }
}
```

This results in the following error messages:

```
call-in-complex-expression.dzn:20:16: error: call used in a complex
    expression
call-in-complex-expression.dzn:21:21: error: call used in a complex
    expression
call-in-complex-expression.dzn:22:19: error: call used in a complex
    expression
```

11.8.5 `action` value discarded

A valued Action is called without using its return value.

```
interface ihello
{
  in bool hello ();
  behavior
  {
    on hello: reply (true);
  }
}

component action_discard_value
{
  provides ihello p;
  requires ihello r;
  behavior
  {
    on p.hello (): r.hello ();
  }
}
```

This results in the following error message:

```
action-discard-value.dzn:16:20: error: action value discarded
```

11.8.6 `call` value discarded

A valued function is called without using its return value.

```
interface call_discard_value
```

```
{
  in void hello ();
  behavior
  {
    bool f ()
    {
      return true;
    }

    on hello: f ();
  }
}
```

This results in the following error message:

```
call-discard-value.dzn:11:15: error: call value discarded
```

11.9 Well-formedness – Injection

Not every port can be injected.

11.9.1 `injected` port has `out`-events

When a Component has a `requires injected` port, its interface must not have `out`-events.

```
interface ihello
{
  in bool hello ();
  out void world ();
  behavior
  {
    on hello: world;
  }
}

component injected_with_out_event
{
  provides ihello p;
  requires injected ihello r;
  behavior
  {
  }
}
```

This results in the following error message:

```
injected-with-out-event.dzn:14:3: error: injected port ‘r’ has out
    events: world
injected-with-out-event.dzn:4:3: info: port defined here
```

11.10 Well-formedness – Functions

- A function body can only contain imperative statements, including `action`s. 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.

11.10.1 Missing `return`

A valued function should return a value using the `return` statement. An error is issued when a return is not guaranteed. An example:

```
interface ihello
{
  in void hello ();
  behavior
  {
  }
}

component function_missing_return
{
  provides ihello p;
  behavior
  {
    bool a = true;
    bool b = false;
    bool c = true;
    bool func ()
    {
      if (a && b)
        return true;
      else if (c)
        illegal;
    }
  }
}
```

This results in the following error message:

```
function-missing-return.dzn:21:12: error: missing return
```

11.10.2 Cannot use `return` outside of function

A `return` statement is restricted to function body. So:

```
interface return_outside_function
{
  in void hello ();
  behavior
  {
```

```
    on hello: return;
  }
}
```

This results in the following error message:

```
return-outside-function.dzn:6:15: error: cannot use return outside of
    function
```

11.10.3 Cannot use statement after recursive `call`

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:

```
interface function_not_tail_recursive
{
  in void hello ();
  behavior
  {
    void f ()
    {
      bool b = false;
      if (b)
      {
        f ();
        b = true;
      }
    }
  }
}
```

This results in the following error message:

```
function-not-tail-recursive.dzn:11:9: error: cannot use statement after
    recursive call
function-not-tail-recursive.dzn:12:9: info: statement after call
```

Note: Two functions `f` and `g` that are defined in terms of each other are mutual recursive and are thus also considered to be recursive.

11.11 Well-formedness – Data Parameters

The restrictions on data parameters are summarised here.

11.11.1 Type mismatch: parameter expected `extern`

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

```
extern int $int$;
interface event_with_bool_porameter
{
  in void hello (bool b);
  behavior {}
```

```
}
```

This results in the following error message:

```
event-with-bool-parameter.dzn:4:18: error: type mismatch: parameter ‘b’;
    expected extern, found: ‘bool’
```

11.11.2 Cannot use out-parameter on out-event

An out-event must not have an out-parameter.

```
extern int $int$;
interface out_parameter_on_out_event
{
  out void world (out int value);
  behavior {}
}
```

This results in the following error message:

```
out-parameter-on-out-event.dzn:4:19: error: cannot use out-parameter on
    out-event ‘world’
```

11.11.3 Cannot use inout-parameter on out-event

An out-event must not have an inout-parameter. An example:

```
extern int $int$;
interface inout_parameter_on_out_event
{
  out void world (inout int value);
  behavior {}
}
```

This results in the following error message:

```
inout-parameter-on-out-event.dzn:4:19: error: cannot use inout-parameter
    on out-event ‘world’
```

11.11.4 Formal binding is not a data member variable

Formal binding, which is the binding of a data member variable `data` to an event parameter `p` using the `p <- data` construct, is only allowed in a component, in an `on` context. Using `<-` in any other context is reported as a parse error.

```
extern int $int$;
interface ihello
{
  in void hello (int i);
  in void cruel (int i);
  behavior
  {
    on hello:;
    on cruel:;
  }
}
```

```
component parse_out_binding
{
  provides ihello p;
  behavior
  {
    bool b = false;
    int data;
    blocking on p.hello (i <- data): {}
    blocking on p.cruel (b <- data): {}
    blocking on p.cruel (data <- i): {}
    blocking on p.cruel (k <- b): {}
  }
}
```

This results in the following error messages:

```
out-binding-reversed.dzn:22:26: error: formal binding ‘i’ is not a data
    member variable
out-binding-reversed.dzn:23:26: error: formal binding ‘b’ is not a data
    member variable
```

11.12 Well-formedness – System

In a system, the component's ports and all sub component's ports must be bound correctly.

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

11.12.1 `port` not bound

No binding is specified for a port of a system.

```
interface ihello
{
  in bool hello ();
  behavior {}
}

component port_not_bound
{
  provides ihello p;
  system {}
}
```

This results in the following error message:

```
port-not-bound.dzn:9:3: error: port ‘p’ not bound
```

11.12.2 `port` not bound – of `instance`

No binding is specified for a port of a component `instance`.

```
interface ihello
{
```

```
  in bool hello ();
  behavior {}
}

component hello
{
  provides ihello p;
  behavior {}
}

component instance_port_not_bound
{
  system
  {
    hello h;
  }
}
```

This results in the following error message:

```
instance-port-not-bound.dzn:9:3: error: port ‘p’ not bound
instance-port-not-bound.dzn:17:5: info: of instance: ‘h’
```

11.12.3 port is bound more than once

More than one binding is specified for a port of a system or one of its component instances:

```
interface ihello
{
  in bool hello ();
  behavior {}
}

component hello
{
  provides ihello p;
  behavior {}
}

component instance_port_not_bound
{
  provides ihello p;
  system
  {
    hello h;
    hello i;
    p <=> h.p;
    p <=> i.p;
  }
}
```

This results in the following error messages:

```
port-bound-twice.dzn:20:5: error: port ‘p’ is bound more than once
port-bound-twice.dzn:21:5: error: port ‘p’ is bound more than once
```

11.12.4 Cannot bind port to port

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 system to a port of a component instance, the directions must be the same:
  - **provides** binds to **provides**
  - **requires** binds to **requires**
- When binding a port of the system to another port of the system Component, the directions must be the opposite:
  - **provides** binds to **requires** or vice versa.
- When binding a port of a component instance to a port of another (or the same) component instance, the directions must be the opposite:
  - **provides** binds to **requires** or vice versa.

```
interface ihello
{
  in bool hello ();
  behavior {}
}

component hello
{
  provides ihello p;
  requires ihello r;
  behavior {}
}

component world
{
  provides ihello p;
  behavior {}
}

component instance_port_not_bound
{
  provides ihello p;
  system
  {
    hello h;
    world w;
    p <=> h.r;
    h.p <=> w.p;
```

```
  }
}
```

This results in the following error messages:

```
binding-mismatch-direction.dzn:27:5: error: cannot bind provides port
    ‘p’ to requires port ‘r’
binding-mismatch-direction.dzn:22:3: info: port ‘p’ defined here
binding-mismatch-direction.dzn:10:3: info: port ‘r’ defined here
binding-mismatch-direction.dzn:28:5: error: cannot bind provides port
    ‘p’ to provides port ‘p’
binding-mismatch-direction.dzn:9:3: info: port ‘p’ defined here
binding-mismatch-direction.dzn:16:3: info: port ‘p’ defined here
```

11.12.5 Cannot bind two wildcards

```
interface ihello
{
  in void hello ();
  behavior {}
}

component hello
{
  provides ihello p;
  requires injected ihello r;
  behavior {}
}

component logger
{
  provides ihello p;
  behavior {}
}

component binding_two_wildcards
{
  provides ihello p;
  system
  {
    hello h;
    logger log;

    p <=> h.p;
    log.* <=> *;
  }
}
```

This results in the following error messages:

```
binding-two-wildcards.dzn:29:5: error: cannot bind two wildcards
```

```
binding-two-wildcards.dzn:16:3: error: port ‘p’ not bound
binding-two-wildcards.dzn:26:5: info: of instance: ‘log’
```

11.12.6 `instance` in in a cyclic binding

We can define communication “direction” for bindings as follows:

- For two component instances communicating: the `requires` port directs to the `provides` port in the binding.
- For port forwarding (an external port is forwarded to a component instance port) or vice versa: A `provides` external port directs to a component instance `provides` port, and a component instance `requires` 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:

```
interface ihello
{
  in void hello ();
  behavior {}
}

component hello
{
  provides ihello p;
  requires ihello r;
  behavior {}
}

component binding_cycle
{
  system
  {
    hello h;
    h.p <=> h.r;
  }
}
```

This results in the following error messages:

```
binding-cycle.dzn:18:5: error: instance ‘h’ is in a cyclic binding
```

A more elaborate example creates a cycle over three components:

```
interface ihello
{
  in void hello ();
  behavior {}
}

component hello
```

```
{
  provides ihello p;
  requires ihello r;
  behavior {}
}

component world
{
  provides ihello p_left;
  provides ihello p_right;
  requires ihello r_left;
  requires ihello r_right;
  behavior {}
}

component binding_cycle
{
  provides ihello p_left;
  provides ihello p_right;
  requires ihello r_left;
  requires ihello r_right;
  system
  {
    hello h1;
    hello h2;
    world w1;
    world w2;

    p_left <=> w1.p_left;
    p_right <=> w2.p_left;

    w1.r_left <=> h1.p;
    w1.r_right <=> h2.p;

    w2.r_left <=> w1.p_right;
    w2.r_right <=> r_right;

    h1.r <=> r_left;
    h2.r <=> w2.p_right;
  }
}
```

This results in the following error message:

```
binding-cycle-elaborate.dzn:32:5: error: instance ‘h2’ is in a cyclic
    binding
binding-cycle-elaborate.dzn:33:5: error: instance ‘w1’ is in a cyclic
    binding
```

```
binding-cycle-elaborate.dzn:34:5: error: instance ‘w2’ is in a cyclic
    binding
```

11.12.7 Cannot bind wildcard to `requires` port

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:

```
interface ihello
{
  in void hello ();
  behavior {}
}

component hello
{
  requires injected ihello r;
}

component logger
{
  requires ihello r;
}

component binding_wildcard_requires
{
  system
  {
    hello h;
    logger log;

    log.r <=> *;
  }
}
```

This results in the following error message:

```
binding-wildcard-requires.dzn:24:5: error: cannot bind wildcard to
    requires port ‘r’
binding-wildcard-requires.dzn:14:3: info: port ‘r’ defined here
```

11.12.8 System composition is recursive

A system may instantiate an arbitrary set of components, which in turn can be systems 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 systems are defined that have mutual instances. System `c1` instantiates `c3`, which instantiates `c4`, which instantiates `c1`.

```
component c1 {
  system {
```

```
    c2 ci2;
    c3 ci3;
  }
}

component c2 {
  system {
    c5 ci5;
  }
}

component c3 {
  system {
    c4 ci4;
    c2 ci2;
  }
}

component c4 {
  system {
    c1 ci1;
  }
}

component c5 {
  system { } // an empty system
}
```

This results in the following error messages:

```
recursive-system.dzn:1:1: error: system composition of ‘c1’ is recursive
recursive-system.dzn:14:1: error: system composition of ‘c3’ is
    recursive
recursive-system.dzn:21:1: error: system composition of ‘c4’ is
    recursive
```

11.12.9 Cannot bind `external` port to non-`external` port

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.

```
interface i {
  in void e ();
  behavior {
    on e: {}
  }
}
```

```
component c1 {
  provides i p;
  requires external i r1;
  requires external i r2;
  behavior {
    on p.e (): {}
  }
}

component c2 {
  provides i p;
  behavior {
    on p.e (): {}
  }
}

component s1 {
  provides i p;
  requires i r;
  system {
    c1 ci1;
    c2 ci2;
    p <=> ci1.p;
    ci1.r1 <=> r;
    ci1.r2 <=> ci2.p;
  }
}

component s2 {
  provides i p1;
  provides i p2;
  requires external i r1;
  requires external i r2;
  system {
    s1 si1;
    p1 <=> si1.p;
    p2 <=> r2;
    r1 <=> si1.r;
  }
}
```

This results in the following error message:

```
binding-mismatch-external.dzn:45:5: error: cannot bind external port
    ‘r1’ to non-external port ‘r’
binding-mismatch-external.dzn:39:3: info: port ‘r1’ defined here
binding-mismatch-external.dzn:26:3: info: port ‘r’ defined here
```

12 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 Libera Chat IRC network. We welcome ideas, bug reports (please send your bug reports to `bug-dezyne@nongnu.org`), patches, and anything that may be helpful to the project.

Note: bug reports contain at least descriptions of:

1. Steps to reproduce the bug
2. What you expected to see
3. What you actually saw

You can help us by increasing the signal to noise ratio in your communication including your bug reports.

Before sending your bug report, please check if you found an already known problem first at dezyne bugs at gitlab (`https://gitlab.com/dezyne/dezyne-issues/-/issues`).

12.1 Building from Git

If you want to hack Dezyne yourself, it is recommended to use the latest version from the Git repository:

```
git clone git://git.savannah.nongnu.org/dezyne
```

To setup a development environment, we use GNU Guix (`https://gnu.org/software/guix/`) (see *The GNU Guix Manual*); run

```
guix environment -l guix.scm
```

If you are unable to use Guix when building Dezyne from a Git checkout, the following are the required packages in addition to those mentioned in the installation instructions (see Section 2.2 [Requirements], page 4).

- GNU Autoconf (`https://gnu.org/software/autoconf/`);
- GNU Automake (`https://gnu.org/software/automake/`);
- GNU Gettext (`https://gnu.org/software/gettext/`);
- GNU Texinfo (`https://gnu.org/software/texinfo/`);
- GNU Help2man (optional) (`https://gnu.org/software/help2man/`).

Run `./autogen.sh` to generate the build system infrastructure using Autoconf and Automake.

Then, run `./configure` and `make` as usual.

12.2 Running Dezyne Before It Is Installed

After making changes you will want to test them. To that end, all the command-line tools can be used even if you have not run `make install`. To do that, you first need to have an environment with all the dependencies available (see Section 12.1 [Building from Git], page 126), and then simply prefix each command with `./pre-inst-env`. As an example, here is how you would verify the trivial the `hello` test:

```
$ ./pre-inst-env dzn -v verify test/all/hello/hello.dzn
```

See the file `HACKING` for some developer tips and tricks.

12.3 The Perfect Setup

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`.

... ...

12.4 Coding Style

In general our code follows the GNU Coding Standards (see *GNU Coding Standards*)[1]. However, they do not say much about Scheme, so here are some additional rules.

12.4.1 Programming Paradigm

Scheme code in Dezyne is written in a purely functional style. One exception is code that involves input/output, and procedures that implement low-level concepts, such as memoization.

12.4.2 Data Types and Pattern Matching

The tendency in classical Lisp is to use lists to represent everything, and then to browse them "by hand" using `car`, `cdr`, `cadr`, and co. There are several problems with that style, notably the fact that it is hard to read, error-prone, and a hindrance to proper type error reports.

Dezyne code should define appropriate data types (`AST` or `GOOPS` classes, or using `define-immutable-record-type`) rather than abuse lists. In addition, it should use pattern matching, via Guile's `(ice-9 match)` module, especially when matching lists (see Section "Pattern Matching" in *GNU Guile Reference Manual*).

12.4.3 Formatting Code

When writing Scheme code, we follow common wisdom among Scheme programmers. In general, we follow the Riastradh's Lisp Style Rules (`https://mumble.net/~campbell/scheme/style.txt`). This document happens to describe the conventions mostly used in Guile's code too. It is very thoughtful and well written, so please do read it.

In addition, Dezyne uses the following formatting for `if`

```
(if test? trivial-case
    the-more-elaborate-case)
```

If you do not use Emacs, please make sure to let your editor knows these rules.

12.5 Submitting Patches

Development is done using the Git distributed version control system. Thus, access to the repository is not strictly necessary. We welcome contributions in the form of patches as

---

[1] A notable exception is the `c++` runtime and handwritten code

produced by `git format-patch` sent to the `dezyne-devels@nongnu.org` mailing list (see Section "Submitting patches to a project" in *Git User Manual*).

Please write commit logs in the ChangeLog format (see Section "Change Logs" in *GNU Coding Standards*); you can check the commit history for examples.

Concept Index

U

W

Appendix A GNU Free Documentation License

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 noncommercially. 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.