Code generation and model-based testing in context of OIL

Master Thesis

Mark Frenken

Supervisors:
Tim A. C. Willemse (TU/e), Louis van Gool (Océ)

Tutors:
Olav Bunte (TU/e), Jasper Denkers (TU Delft)

November 2019

Abstract

OIL is a domain-specific language developed at Océ for specifying, analysing, and implementing software components. OIL is to have IDE support, transformations to formal modelling languages for requirement verification, and code generation towards general-purpose languages such as C++. All of which are currently being implemented in the Spoofax language workbench. Our contribution to the OIL architecture is an implementation of the C++ code generator.

In order to validate the correctness of the code generator, we make use of model-based testing. Model-based testing is an approach to test whether the behaviour of an implementation conforms to the behaviour described in a formal model. The benefit of this approach is that test derivation and subsequent execution is automated. Moreover, test cases are derived from models that can be formally verified. We validate the correctness of the code generator by testing whether the behaviour of generated C++ implementations conform to the behaviour described in their original OIL specifications.

In this master thesis, we present our approach to implement the C++ code generator in the OIL architecture, and validate its correctness.

Contents

Chapter 1

Introduction

Model-driven engineering (MDE) is one of the ways to deal with increasing complexity in software engineering. The key is to abstract away from the specifics of an implementation. MDE aims to raise the level of abstraction by looking at software in terms of models.

Creating a domain-specific language (DSL) is an approach to implement MDE. A DSL can make the development of software in a specific domain faster, and better suited for evolution. The software engineer can focus on describing the core process since much of the platform-specific code that is needed when using general-purpose languages is abstracted away.

Correctness and reliability are also becoming increasingly difficult to guarantee due to the increasing complexity of software systems. By describing the behaviour of software components in formal models, engineers can use formal methods to verify requirements on these models.

Océ is a company that applies MDE to its software development. The behaviour of several of their software components is specified in models. Software engineers build the implementations and corresponding test cases based on these models. Next to the benefits that come with describing behaviour in models, this approach also comes with challenges. For proper evolution of the software, the implementation, test cases, and the model must evolve together. It is inevitable that inconsistencies start occurring over longer periods of evolution.

Océ attempts to remedy these problems with the Océ Interaction Language (OIL), originally created by Louis van Gool. OIL is a domain-specific language for specifying, analysing and implementing software components. Using state of the art language workbenches, OIL is to have IDE support, transformations to formal modelling languages for requirement verification, and code generation to implementations in general-purpose languages such as C++.

The OIL architecture is currently being implemented in the Spoofax language workbench by Olav Bunte and Jasper Denkers, two PhD candidates from the Eindhoven University of Technology and Delft University of Technology respectively. The work of Denkers focusses on the development of the DSL, and the work of Bunte focusses on the transformation of OIL to formal models.

In this thesis, we present an approach to generate an implementation in a general-purpose language from an OIL specification. Moreover, we present an approach to validate the correctness of the code generator by means of model-based testing.

In order to develop a code generator for OIL, we perform an in-depth analysis of the semantics of OIL. We then express these semantics in pseudo-code, which serves as a blueprint for expressing OIL semantics in an object-oriented general-purpose language.

We develop a C++ code generator in the existing OIL architecture implemented in Spoofax, and make use of the existing transformations by composing them into our newly defined transformation. We introduce two additional intermediate representations to make these complex transformations more manageable. When defining our transformations from OIL to C++, we also take into account future transformations to other GPLs.

Next, we validate the correctness of the code generator. Formally verifying the correctness of the transformation from a specification in OIL to an implementation in C++ is not feasible. Instead, we present an approach to validate the correctness of the generated C++ implementation with respect to the OIL specification. With access to a verified model in a formal modelling language, model-based testing is a suitable choice. Model-based testing is an approach to test whether the implementation of a software component conforms to a formal model [21]. Instead of manually having to create test cases, model-based testing allows for the algorithmic generation of test cases.

We can use model-based testing to prove that the code generator is incorrect by finding a counterexample, i.e., a generated implementation in C++ exhibiting behaviour that does not conform to the respective OIL specification. We design a number of experiments to find such a counterexample. None of the experiments were able to prove the code generator incorrect. This result validates the correctness of the code generator in the context of the experiments.

1.1 Related Work

In 2007, Visser presented a case study in developing the domain-specific language WebDSL [26]. WebDSL is a language designed specifically for the development of dynamic web applications [9]. The case study by Visser can serve as a guide to creating domain-specific languages. It details this process from its inception all the way through to its implementation. The study applies an incremental approach to its development, and SDF [24] and Stratego/XT [6] are the tools to define the language and the code generation respectively.

For the purpose of generating to multiple object-oriented general-purpose languages (GPL), Hemel and Visser propose a Platform Independent Language (PIL) [12]. The PIL serves as an additional layer of abstraction between the DSL and the target language. It generalizes common constructs found in object-oriented languages to make code generation from PIL to any GPL more simple. The approach was implemented in the code generation for WebDSL, which includes a translation to both Python and Java.

mCRL2 is a formal specification language and toolset for modelling, validation and verification of concurrent systems and protocols [7, 10]. mCRL2 is developed at the Technical University of Eindhoven, and it is broadly used for research in the area of formal systems analysis.

The mCRL2 toolset has been applied in a number of industrial case studies. Among them is the control software of the Compact Muon Solenoid experiment at CERN by Hwong et al. [13]. The control software is programmed in the State Machine Language (SML), and consists of over 27,500 hierarchically organized finite state machines. To formalize SML, Hwong et al. present a mapping to the mCRL2 language. The translation was implemented using the ASF+SDF meta-environment. Hwong et al. selected specific properties of the finite state machines that could be verified in isolation, for which they also built dedicated tooling. Several issues in the control software could be identified using these tools. Later, they were integrated into the development environment.

The thesis by Belinfante from 2014 describes the application of model-based testing with a series of case studies [4]. This work builds on the work of Tretmans' input-output conformance testing (ioco-testing), which is a theory for model-based testing. Belinfante uses JTorX [3], which is a tool that implements model-based testing.

Bouwman et al. present an application of mCRL2 and JTorX in the signalling domain [5]. The mCRL2 toolset is used for modelling the behaviour of an existing signalling system, and JTorX is used to validate the model through model-based testing. The mCRL2 toolset is then used to test high-level safety properties on the validated model. Model-based testing in JTorX revealed several inconsistencies between the model and the software.

Beusekom et al. present a transformation from Dezyne to mCRL2 [22]. Dezyne is a domain-specific language with corresponding toolset for specifying software interfaces and components and checking the compliance between the two. By providing an encoding of Dezyne in the mCRL2 language, Beusekom et al. give Dezyne a formal semantics. The transformation from Dezyne to mCRL2 is implemented in Python. Validation of the transformation is done using a set of test cases, consisting of 168 component models and 224 interface models. The mCRL2 specifications obtained from the transformation were strongly bisimilar to the formal models provided in the test cases, which validated the transformation.

1.2 Research questions

Our research takes place within an existing project which is a collaboration between Océ, Eindhoven University of Technology, and Delft University of Technology. The project revolves around OIL, a domain-specific language developed by Louis van Gool at Océ. The goal of the project is to design and implement an architecture around OIL that includes:

- An IDE environment for OIL

- Formal verification of requirements on OIL specifications
- Code generation (C++, C#, Java, etc.)
- Validation of generated code with respect to the formally verified specification

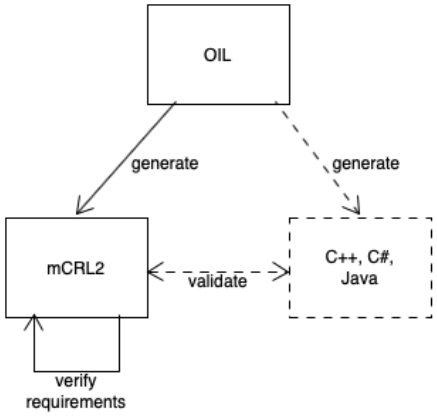


Figure 1.1: Overview of the architecture

Figure 1.1 shows an illustration of the architecture. The implementation of the OIL domain-specific language is currently being worked on by Jasper Denkers (TU Delft) as part of his PhD research. The transformation from OIL to mCRL2, a language and toolset for modelling system behaviour [7], is also in development and part of the PhD research of Olav Bunte (TU Eindhoven).

The research in this master thesis builds on the work of Bunte and Denkers. We will implement the code generator, and investigate how we can validate the generated code with respect to the original specification. We translate this to the following research questions:

- **RQ1**: How can we design an implementation that can generate C++ code from an OIL specification?
- **RQ2**: How can we generalize this code generator to other object-oriented general-purpose languages like C# and Java?
- **RQ3**: How can we validate the correctness of the code generator using model-based testing?

In this report, we investigate how we can generate an implementation in a general-purpose language (GPL) from an OIL specification. Moreover, we investigate how we can validate the correctness of the code generator by means of model-based testing. First, we give an introduction to the OIL language in Chapter 2. This chapter should give the reader an impression about what we can express with OIL, and how we can express it. We continue in Chapter 3 with transformation and code generation in the context of OIL, where we address **RQ1** and **RQ2**. In this chapter, we start with presenting the necessary background on defining transformations, and we continue with the context, implementation, discussion, and lessons learned. Chapter 4 is about model-based testing in the context of OIL. It is in this chapter that we address **RQ3**. Again we start with giving the necessary background on formal models and theories necessary for understanding model-based testing, and we continue with the implementation, experiments, and discussion. Finally, Chapter 5 presents the conclusions of our research and opportunities for future work.

Chapter 2

OIL

In this chapter, we introduce the Océ Interaction Language (OIL). OIL is a domain-specific language for specifying, analysing and implementing software components. The language was created by Louis van Gool at Océ. The company is already applying several model-driven engineering (MDE) principles in its software development. With OIL, Océ hopes to bring its implementation of MDE one step further with the addition of formal verification to code generation. The development of OIL is not considered core business at Océ, hence the company is actively sharing information with other companies and universities about its experience with OIL. The intention is to make OIL open source so that others can also benefit from the language and contribute to its further development.

OIL is a simple and intuitive language, supported by a graphical representation. It was initially designed to specify and analyse communication between software components. We refer to these as *protocol specifications*, as they describe protocols for communication between components. It was later that van Gool realized that the language can also be used to specify the actual behaviour of a component. These specifications are referred to as *component specifications*, which can serve as a source for code generation.

A software system consists of channels and components. Communication between components takes place by sending *events* over the channels. Components subscribe to channels, and can both receive and send events over these channels. An example of an event is a method call between the environment and a component, or from one component to another. An event can also be a reply. The method call is the only event type relevant in the context of this thesis, therefore we will often use method call and event interchangeably.

Interfaces for the software components specified in OIL are defined in the Interface Definition Language (IDL). The IDL captures the static information of an OIL specification, namely the public methods of a component and to which interface they belong.

2.1 Basic specifications in OIL

In this section, we explain the most important concepts of the syntax and semantics of OIL through a series of simple examples. Currently, there exists both an XML (OILXML) and DSL (OILDSL) version of the OIL syntax. In this thesis, we will limit ourselves to OILDSL, as this version of the syntax is more compact and easier to read. We also use OIL diagrams for a graphical representation of the specification. The goal of this section is to give the reader an impression of what OIL is, and what it is capable of expressing.

2.1.1 A power switch in OIL

Figure 2.1 shows an OIL specification for a power switch. The switch can be used to turn the power on or off. Notice that this graphical representation is reminiscent of a finite-state machine (FSM). The more advanced specifications in this section show that OIL features some meaningful extensions on the FSM, that will allow us to more easily define complex behaviour of software components.

In OIL, the *global state* is a collection of variables. A valuation of these variables expresses which states of the component are active. In our example device, `power` is such a variable. Since

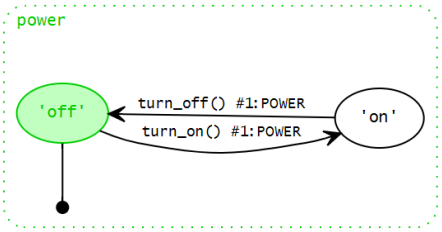


Figure 2.1: A power switch in OIL

there is only one variable present, the value of `power` expresses the global state of our device.

An OIL specification consists of *areas*. There are three types of areas in OIL: *regions*, *states*, and *scopes*. We will explain regions and states in this example. The device in Figure 2.1 shows a dotted box called `power`. This box is an area of type region. Each region refers to one global state variable and contains a collection of states. A state is represented as a square box with rounded corners, or as an ellipse (as shown in Figure 2.1). Each state refers to a value. When we refer to the region of a state, we mean the region which contains that state. When we refer to the variable of a state, we mean the global state variable which is referred to by the region of the state. Our example shows two states in `power`: `'off'`, and `'on'`. A state is considered active when the value of the state is equal to the value of its variable. In our example, this means that if `power` has value `'off'`, then inside region `power`, the state with value `'off'` is active.

Transitions between areas can change the global state of a component. Each transition has one source area and one target area, is labelled with a method name, and is related to one or more concerns. The notation in the diagrams is as follows:

```
method_name() #i:CONCERN
```

We use the combination of the method name and the transition number (`method_name() #i`) to refer to a specific transition. We will elaborate on the notion of concerns in OIL later in this section. The power switch features two transitions: `turn_on() #1` and `turn_off() #1`. Figure 2.1 specifies that `turn_on() #1` is a transition from state `'off'` to state `'on'`. In order for `turn_on() #1` to fire, state `'off'` must be active. When we fire `turn_on() #1`, we update the value of the global state variable `power` to `'on'`. As a result, the state `'on'` is now active, and `'off'` inactive.

We can fire transitions, thus changing the global state of a component, by sending an event over a communication channel. The power switch can react to two events: `turn_on()` and `turn_off()`. An example of a possible sequence of events and state changes is as follows: The device starts with the power set to `'off'`. We can send the `turn_on()` event to set the power to `'on'`. When the device is powered on, we can do `turn_off()` to set power to `'off'`.

```
1 region power {
2   state off
3   state on
4 }
5 concern POWER {
6   in off on turn_on() go on end
7   in on on turn_off() go off end
8 }
```

Listing 2.1: OIL code for the simple device in Figure 2.1

Listing 2.1 shows the OIL code for the specification in Figure 2.1. On lines 1–4 we define the *areas* of our specification. On lines 6 and 7 we define the *transitions* of our specification. The transitions are related to the concern on line 5. Transitions are structured as follows:

```
in <source area> on <method> go <target area> end
```

We take a closer look at the transition defined on line 6. Here we see that the source area is the state `off`, the method is `turn_on()`, and the target area is the state `on`.

2.1.2 A simple printer in OIL using separation of concerns

Figure 2.2 shows how we can extend the power switch to become a simple printer, with the OIL code for this component presented in Listing 2.2. Notice the global state of this component is defined by two variables, `power` and `job`. The variable `power` indicates whether the printer is powered on or off, as was the case in the power switch. The variable `job` indicates whether the printer is currently idle or printing a page. We have effectively separated two different concerns (`power` and `job`) within the specification, making it easier to comprehend.

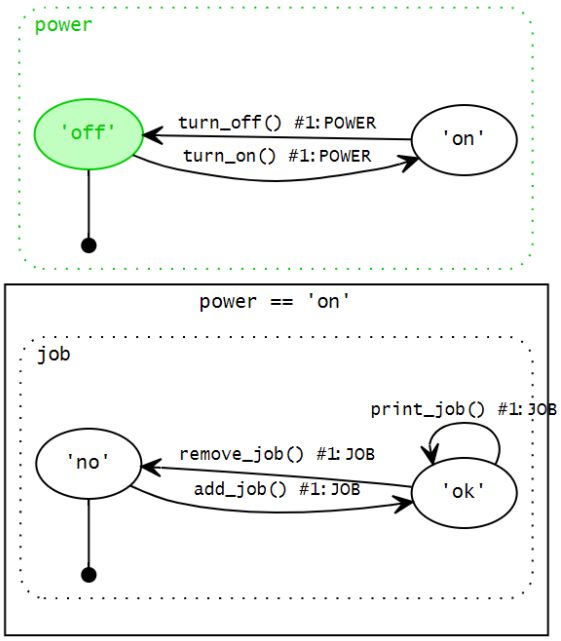


Figure 2.2: A simple printer in OIL

```
region power {
  state off
  state on
}
concern POWER {
  in off on turn_on() go on end
  in on on turn_off() go off end
}
scope power_on[power == 'on'] {
  region job {
    state no
    state ok
  }
}
concern JOB {
  in no on add_job() go ok end
  in ok on print_job() go ok end
  in ok on remove_job() go no end
}
```

Listing 2.2: OIL code for the simple printer in Figure 2.2

For obvious reasons, the printer in Figure 2.2 needs to be powered on in order to function. In order to specify this behaviour, we use the area type *scope*. A scope is represented as a square box. A scope restricts behaviour according to its invariant. Transitions with the source area within a scope can only fire if the invariant of said scope is true. The invariant we use in Figure 2.2 is `power == 'on'`. This results in behaviour such that the transitions `add_job() #1`, `print_job() #1` or

`remove_job() #1` can only fire when region `power` is in state `'on'`. Notice that the transitions `turn_off() #1` and `turn_on() #1` can fire regardless of the value `job` currently has.

Regions, states, and scopes can be nested arbitrarily. A nested area can only become active if the parent area is active. Using this general rule for areas, we can see the specification in Figure 2.3 has the same behaviour as the original simple printer in Figure 2.2. Instead of using a scope with an invariant to limit the behaviour of `job`, we nest the region that determines `job` inside state `'on'`. The result is the same as the solution using scope. The states in region `job` can only become active when state `on` is active, and transitions can only fire if the source area is active.

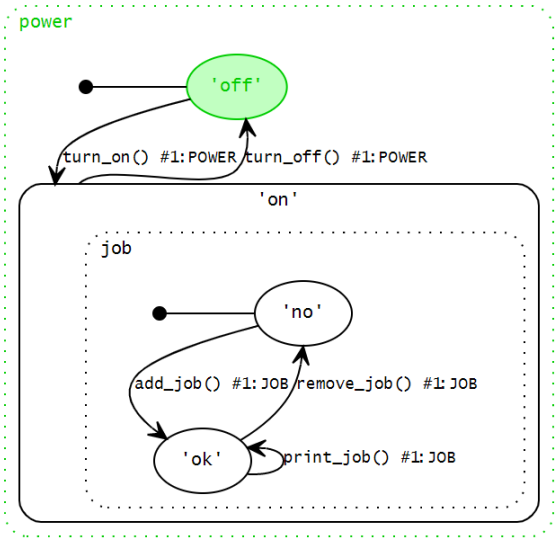


Figure 2.3: A simple printer in OIL, specified using nested areas

2.1.3 Multiplex events and concerns

OIL supports multiple transitions firing simultaneously. When a method call occurs, all transitions labelled with that method which can fire, will fire. We refer to this feature as *multiplex events*.

Figure 2.4 (with the respective OIL code presented in Listing 2.3) shows a specification for a printer with a heat management system. The printer can be turned on and off at most twice before overheating. In order to prevent the overheating, the printer must cool down. This behaviour is specified in region `heat`. Notice that this region features two transitions labelled with `turn_on()`, and these two can be fired one after another. As with the power switch, we should only be able to turn on the printer when it is off. The same applies to turning the device off. However, multiplex events require that, on the occurrence of an event, all transitions labelled with this event that can fire, must fire. This means that we can fire `turn_on()` only twice in a row due to region `heat`.

```
1  scope no_job[job == 'no'] {
2    region power {
3      state off
4      state on
5    }
6  }
7  concern POWER {
8    in off on turn_on() go on end
9    in on on turn_off() go off end
10 }
11 scope power_on[power == 'on' and heat != 'cold'] {
12   region job {
13    state no
14    state ok
15   }
16 }
```

```
17 concern JOB {
18   in no on add_job() go ok end
19   in ok on remove_job() go no end
20   in ok on print_job() go ok end
21 }
22 scope no_job2[job == 'no'] {
23   region heat {
24     state cold
25     state warm
26     state hot
27   }
28 }
29 concern HEAT {
30   in cold on turn_on() go warm end
31   in warm on turn_on() go hot end
32   in warm on cool_down() go cold end
33   in hot on cool_down() go cold end
34 }
```

Listing 2.3: OIL code for the cool-down printer in Figure 2.4

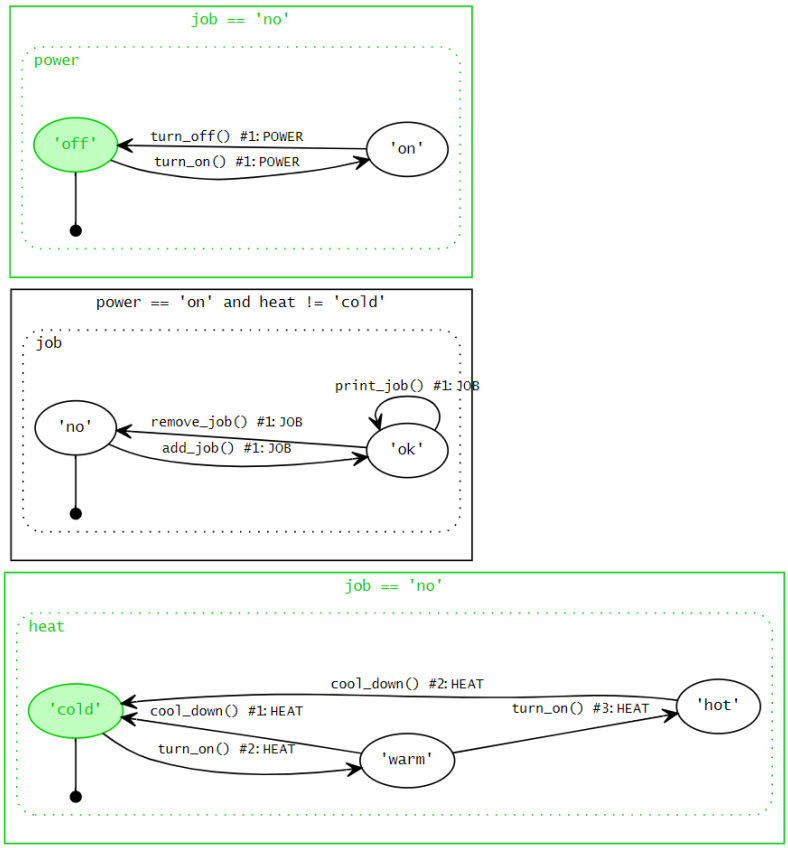


Figure 2.4: The cool-down printer

Concerns allow us to put restrictions on multiplex events. We can relate transitions to concerns. Notice the presence of concerns `POWER`, `JOB`, and `HEAT` in the transition labels in Figure 2.4. An event is related to a concern if one or more of its transitions are related to that concern. If an event related to one or more concerns occurs, then each concern must have at least one transition that can fire. If this is not the case, we have a violation of one or more concerns. The component will terminate with an error if a *concern violation* occurs.

We will show how concerns are used to limit the behaviour of the specification in the following

example. Let the global state of our specification be as follows:

```
power = 'off', job = 'no', heat = 'cold'.
```

The event `turn_on()` occurs, and transitions `turn_on() #1` and `turn_on() #2` can fire. The concerns related to this event are `POWER` and `HEAT`. Concern `JOB` is not related to this event because there does not exist a transition labelled with `turn_on()` and concern `JOB`. By definition of multiplex events the transitions `turn_on() #1` and `turn_on() #2` must fire. Both concerns `POWER` and `HEAT` are satisfied by transitions `turn_on() #1` and `turn_on() #2`, therefore a concern violation does not occur. The global state is now

```
power = 'on', job = 'no', heat = 'warm'.
```

The event `turn_on()` occurs again. In this case, the only transition that can fire is `turn_on() #3`. The transition `turn_on() #3` satisfies concern `HEAT`. Concern `POWER` must also be satisfied since there exists a transition with label `turn_on()` and concern `POWER`. Since region `power` is in state `'on'`, the transition `turn_on() #1` cannot fire. In fact, no transition with label `turn_on()` and concern `POWER` can fire. As a result, the component will terminate with an error due to a concern violation of concern `POWER`.

2.1.4 Arguments, actions, and guards

Some behaviour cannot be expressed in a feasible way using only transitions, states, regions, and scopes. Take for example tracking the number of sheets to be printed. In theory, we could define a region `sheets`, with one state for each number of sheets. We would then have to define transitions between the states to increase or decrease the number. This is not desirable for obvious reasons.

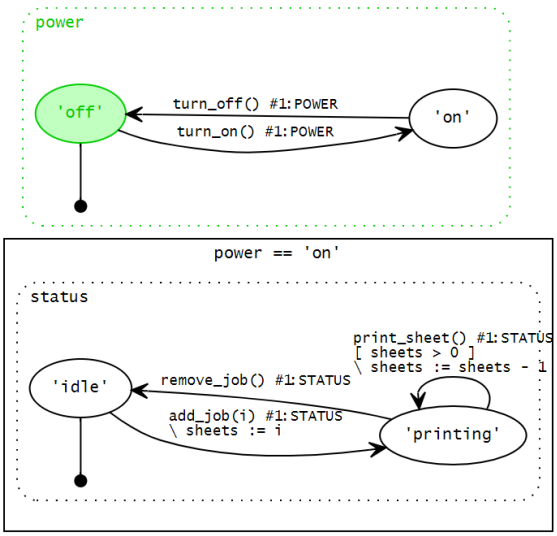


Figure 2.5: Guards and actions in OIL

Parameters, *actions*, and *guards* provide a more practical way of working with data such as numbers. Figure 2.5 (see Listing 2.4 for the OIL code) specifies a printer where we can set the number of sheets we want to print. This is modelled by parameter `i` in `add_job(i)`.

Lines 1–3 in Listing 2.4 show that we have added the new global state variable `sheets` of type `integer` to our specification. This variable is not related to any region or enum type. To set the value `sheets`, the transition `add_job(i) #1` uses an *action* to set `sheets` to `i`, as seen in line 20 in Listing 2.4. In Figure 2.5 the action is denoted as `\ sheets := i`. Actions and state changes occur at the same time. For example, if the event `add_job(3)` occurs and the transition `add_job(3) #1` can fire, the global variables `job` and `sheets` are set to `'printing'` and `'3'` respectively.

Similar to scopes, *guards* offer a way to limit behaviour. The guard `[ sheets > 0 ]` must be satisfied in order for transition `print_sheet() #1` to fire. In combination with the action `\ sheets := sheets - 1` (Listing 2.4 line 24), we have that the event `print_sheet()` can occur at most i times after `add_job(i)`. This means the printer is not able to print more than the specified number of sheets.

```
var {
  sheets : integer
}
region power {
  state off
  state on
}
concern POWER {
  in off on turn_on() go on end
  in on on turn_off() go off end
}
scope power_on[power == 'on'] {
  region status {
    state idle
    state printing
  }
}
concern STATUS {
  in idle on add_job(i)
    do sheets := i
    go printing end
  in printing on print_sheet()
    if sheets > 0
    do sheets := sheets - 1
    go printing end
  in printing on remove_job() go idle end
}
```

Listing 2.4: OIL code for the printer in Figure 2.5

2.2 OIL component specifications for code generation

The specifications described in the previous section should give the reader an intuition on what OIL is, and how OIL can describe the desired behaviour of software components. In this section, we focus on specifications suitable for code generation.

2.2.1 Protocol vs component specifications

The specifications in Section 2.1 describe the desired behaviour of software components. We can use these specifications to evaluate the behaviour of a software system. In case we have a sequence of events that occur in a channel, we can check whether the events adhere to the specifications. We refer to these as protocol specifications.

We can also use these specifications as the definition of a component's behaviour. We refer to these as component specifications. A crucial difference between protocol and component specifications is the perspective from which we view them. In the case of protocol specifications, our view is from the perspective of the system. We look at the communication channel and check which events occur, and in what order. In the case of component specifications, our view is from the perspective of the component. We need to specify which variables belong to this specific instance of the component. We need to specify which events are caused by the environment (*reactive events*), and which events are caused by the component (*proactive events*).

In this study, we only consider the case where there exists one instance of the component. Handling multiple instances of the same component is left to future research.

2.2.2 Run-to-completion semantics

OIL components can have transitions that fire autonomously. From the perspective of the component, these transitions represent method calls made by the component itself. We refer to the occurrence of such method calls as *proactive events*. A component can make method calls to other

components, or to itself. The latter is a special case of proactive events, which we refer to as *silent events*. Silent events represent internal behaviour of the component.

For software components, run-to-completion generally means that the component has to finish performing all tasks, "runs to completion", before it hands control back to the user. In OIL run-to-completion is implemented by giving proactive events priority over reactive events. If a component would have a choice between firing a transition from a reactive event or a transition from a proactive event, it is the latter that must fire. Section 2.2.4 demonstrates how this works in an example.

In order to prevent problematic behaviour due to proactive events, we put two restrictions on the use of these events. First and foremost, loops consisting of only proactive events are not allowed, as these would result in a live-lock. Second, in order to prevent non-deterministic behaviour in the case multiple proactive events are possible, specifications in OIL require confluence of proactive events. This means that if multiple proactive events can fire, the behaviour of the component should be the same irrespective of which event fires first.

2.2.3 Illegal events

In the specification for the power switch in Figure 2.1, notice that the state `'off'` does not have an outgoing transition labelled with `turn_off()`. If we were to send `turn_off()` when the switch is `'off'`, the component is unable to do a transition, and a concern violation will occur. Recall that if a concern violation occurs, the component terminates with an error. In this context, we consider `turn_off()` an *illegal event* because it leads to illegal termination. Figure 2.6 shows all transitions that lead to an illegal termination explicitly. Essentially, the set of illegal events that can occur in a given global state contains all reactive events that lead to termination with an error from that global state.

If we do not model these illegal events, via concerns or otherwise, these events will simply be ignored. Recall that an event in OIL represents a method call to a software component. Software components that have the ability to ignore method calls are not desirable. This can lead to inconsistencies in the system. From this point in this thesis, if an OIL specification is referred to as a component specification, then it is implied that each transition has at least one concern.

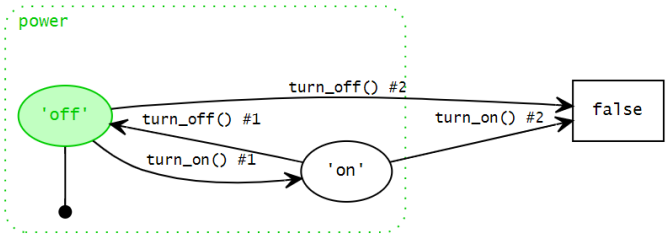


Figure 2.6: All events leading to an illegal state

2.2.4 A component specification in OIL

The printer component in Figure 2.7, with its IDL in Listing 2.5 and the OIL specification in Listing 2.6, is an example of a component specification in OIL. The printer component describes a server where a client may register such that the client can send print jobs and fill the tray. If the client has sent a print job and has filled the tray, the server starts to print (i.e. proactively makes `sheet_printed()` calls on the client) until the tray is empty and/or all sheets have been printed. If a job has been completely printed, the server will proactively transition to state `'ready'`, at which point the client may send the next job.

Notice the use of `this` for global state variables. In case there are multiple instances of the printer component, the use of `this` makes sure we access the correct instance.

We have not explicitly defined any regions. However, given the semantics of OIL, we need regions in order to determine if a state is active. Therefore the existence of some regions is implicit. In this example, we have two regions that exist implicitly. First is the region `root` containing states `'unregistered'` and `'registered'`. Second is the region `registered` containing states `'ready'`

and `'printing'`. These two regions also imply the existence of global state variables `root` and `registered` respectively.

Lines 1 and 2 of Listing 2.6 set the provided and required interfaces. The specification `provides` an implementation for the interface `printer_componenti.printer_server`. Due to the method call `this.c.sheet_printed(sheet_nr=this.i) #1` on the client, this specification `requires` an implementation of the interface `printer_componenti.printer_client`.

Line 13 defines concern `C`. This concern is applied to all transitions. This will make sure illegal events lead to an illegal state. Notice that concern `C` is not mentioned in Figure 2.7. We do not always explicitly show concerns in the diagram when only one concern is present. The existence of such a concern in component specifications is implicit.

Lines 14–18 define the transition `register_client(client)`. The `client` argument contains a pointer to the client instance that is to be registered on the server.

Lines 19–24 define the transition `fill_tray(sheets) #1`, which allows us to fill the printer tray with a specific number of sheets. This transition can fire while state `'registered'` is active.

Lines 25–30 define the transition `print_job(sheets) #1`, which allows us to send a job to the printer to print a specific number of sheets. This transition can fire while states `'registered'` and `'ready'` are active.

Lines 31–37 define the transition `this.c.sheet_printed(sheet_nr=this.i) #1`, which informs the client how many sheets have been printed up until this point. Notice the method of this transition has the interface `printer_componenti.printer_client`. This indicates the transition `this.c.sheet_printed(sheet_nr=this.i) #1` is a method call on another component, i.e. the registered client (`this.c`). Since method calls to other interfaces are considered proactive events, we have that the component will autonomously perform this transition when it can fire, taking priority over transitions caused by reactive events. In practice this means that `this.c.sheet_printed(sheet_nr=this.i) #1` will continue to fire while the job is not completed (`this.i > 0`) and the tray is not empty (`this.f > 0`), and this process cannot be interrupted by method calls to the component.

Lines 38–43 define the transition `all_printed() #1`, which puts the printer back in state `'ready'` when it has finished printing the last job. Line 39 defines `all_printed()` as a silent event, meaning this transition is considered internal behaviour of the component. Since silent events are a special case of proactive events, the run-to-completion semantics apply here as well.

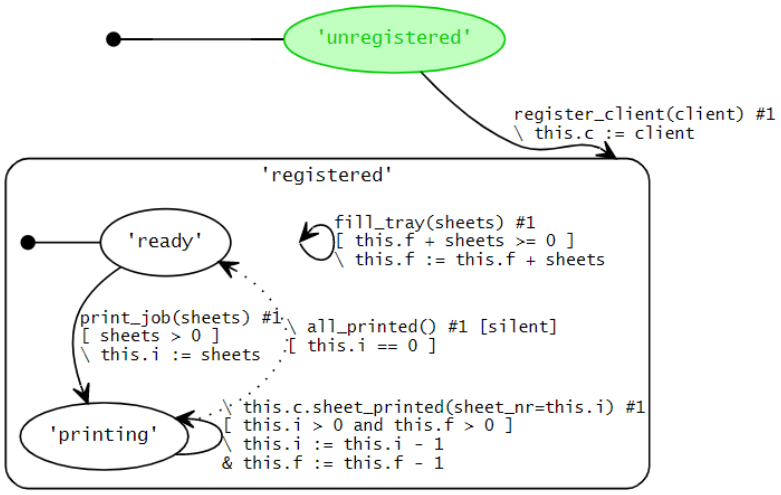


Figure 2.7: printer component in OIL

```
module printer_componenti {
  interface printer_server {
    register_client(client: printer_client)
    fill_tray(sheets: integer)
    print_job(sheets: integer)
  }
  interface printer_client {
    sheet_printed(sheet_nr: integer)
  }
}
```

Listing 2.5: IDL code for the printer component in Figure 2.7

```
provides printer_componenti.printer_server
requires printer_componenti.printer_client

module printer_componenti type call {
var c: printer_componenti.printer_client = 0
var i: integer = 0
var f: integer = 0
state unregistered
state registered {
  state ready
  state printing
}
concern C {
  in unregistered
    interface printer_server
    on register_client(client)
    go registered
    act this.c := client end
  in registered
    interface printer_server
    on fill_tray(sheets)
    go registered
    if this.f + sheets >= 0
    act this.f := this.f + sheets end
  in ready
    interface printer_server
    on print_job(sheets)
    go printing
    if sheets > 0
    act this.i := sheets end
  in printing
    interface printer_client
    do sheet_printed(sheet_nr = this.i)
    go printing
    self this.c
    if this.i > 0 and this.f > 0
    act this.i := this.i - 1 & this.f := this.f - 1 end
  in printing
    interface printer_server
    type silent
    do all_printed()
    go ready
    if this.i == 0 end
}
```

Listing 2.6: OIL code for the printer component in Figure 2.7

Chapter 3

Transformations and code generation in context of OIL

In this chapter, we present our approach for code generation in context of OIL. We describe the transformation from an OIL specification to C++, answering **RQ1**. We provide an approach to implementing other general-purpose languages into the architecture, answering **RQ2**.

3.1 Background

In this section, we discuss the theory and tools we need in order to implement the code generator.

3.1.1 The Spoofax Language Workbench

Language workbenches (LWB) exist for the purpose of developing languages. In Figure 3.1 we see a comparison between modern language workbenches by Erdweg *et al.* from 2015 [8]. The paper presents a feature model which they use for the comparison between different LWBs. With its 97% feature coverage, Spoofax has the highest feature coverage of the LWBs in this comparison. We can conclude Spoofax is the most feature-complete workbench, making it the safest choice for our purposes.

Implemented QL and QLS features per language workbench (●, fully implemented; ◒, partially implemented/limited support).

	Ensō	Más	MetaEdit+	MPS	Onion	Rascal	Spoofax	SugarJ	Whole	Xtext
Syntax	●	●	●	●	●	●	●	●	●	●
Execution										
Rendering	●	●	●	●	●	●	●	●	●	●
Propagation	●	●	●	●	●	●	●	●	●	●
Saving	●		●	●	●	●	●		●	●
Validation										
Names		●	●	●	●	●	●	●	●	●
Types		◒	●	●	●	●	●	●		●
Cycles					●		●			●
Determinism							◒	●		
IDE										
Colouring		●	●	●	●	●	●	●	●	●
Outline			●	●	●	●	●	●	●	●
References		●	●	●		●	●	●	●	●
Marking		●	●	●	●	●	●	●	●	●
QLS										
Sectioning			●		●	●	●		●	●
Pagination			●			●	●			●
Styling			●	◒	●	●	●		●	●
Widgets			●	●	●	●	●			●
Validation			●	●	●	●	●		●	●
Feature coverage (%)	24	44	88	74	82	88	97	59	65	94

Figure 3.1: Comparison between existing LWB by Erdweg et al.

Spoofax is an Eclipse-based language workbench for development of textual domain-specific languages with IDE support [15]. Spoofax includes several meta-DSLs to specify languages. SDF3

[24] is used for syntax specification, Stratego [6] for defining transformation, and NaBL2 [17, 16] for name binding. Spoofax is currently still in development at the Delft University of Technology.

3.1.2 Syntax definition

In Spoofax we define syntax using the syntax definition formalism SDF3 [24]. We can use SDF3 to define both lexical and context-free syntax. Using SDF3, we can define the *concrete syntax* of a language. The concrete syntax is the textual representation of a language. This is a concrete representation of the language. In the context of our research, defining a concrete syntax is not relevant. The concrete syntax of the OIL language is part of the research by Jasper Denkers, and we will not make contributions in this area.

Instead, we will focus on an abstract representation of the syntax, since this format is much more practical for applying transformations. Given a concrete syntax in SDF3, we can derive a corresponding *abstract syntax*. We define an abstract syntax by means of an algebraic *signature*. Spoofax can automatically derive this signature from the syntax definition in SDF3.

Listing 3.2 shows an example of a signature for mathematical expressions in Spoofax. The signature consists of *terms*. The syntax for defining a term is `Term : Sort_1 * ... * Sort_n -> Sort`. A term can be a string, an integer, or a float. A term can also contain other terms or lists of other terms. Each term is of a certain type, which is referred to as a *sort*. In many cases, the term name and sort name are the same, but it can be useful to have multiple terms be of the same sort. For example, when dealing with mathematical expressions we can have a term `Add` for addition and `Sub` for subtraction both be of the same sort `Exp`.

```
1 signature
2   constructors
3     Add : Exp * Exp -> Exp
4     Sub : Exp * Exp -> Exp
5     Num : integer -> Exp
```

Listing 3.1: Signature for mathematical expressions

By parsing the concrete syntax of a language, we obtain the abstract syntax in the form of a data-structure, i.e. the abstract syntax tree (AST). The AST consists of the terms we defined in the signature. The following example shows how parsing a mathematical expression results in an AST that defined with the signature in Listing 3.1:

$$((5 + 3) - 4) \quad \rightarrow \quad \texttt{Sub(Add(Num(5), Num(3)), Num(4)).}$$

Although the AST is more difficult to read, notice that it contains the same information as the original text.

```
1 signature
2   constructors
3     LTS : InitialState * List(Transtion) -> LTS
4     InitialState : Integer -> InitialState
5     Transition : State * Label * State -> Transtion
6     State : integer -> State
7     Label : string -> Label
```

Listing 3.2: Signature for labelled transition system (LTS)

In a second example, we use a DSL that can express labelled transition systems (LTS). The signature for the abstract syntax of the DSL is found in Listing 3.2. Figure 3.2 shows an example of an LTS. Listing 3.3 shows the same LTS expressed in a DSL. Finally, Listing 3.4 shows the AST we obtain by parsing the DSL. Note that the LTS-term contains two terms: the `InitialState` term, and a list that contains `Transition` terms. The textual representation of the AST in Listing 3.4 is expressed in the Annotated Term (ATerm) Format [23].

3.1.3 Transformation and code generation in Stratego

Stratego [6] is a functional language included in Spoofax specifically for program transformations and code generation. Stratego revolves around term rewriting. Given a term pattern from the

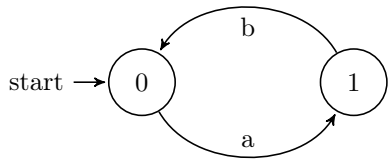


Figure 3.2: LTS from Listing 3.3

```
init: 0
(0, "a", 1)
(1, "b", 0)
```

Listing (3.3) LTS DSL

→

```
LTS( InitialState(0),
  [ Transition(State(0), Label("a"), State(1))
  , Transition(State(1), Label("b"), State(0))
  ]
)
```

Listing (3.4) LTS AST

signature of a language, we apply *rules* to rewrite the source term to another term pattern under a specified condition. The generic structure for a rule is as follows:

```
rule1:  p1 -> p2 where c
```

The rule will attempt to match on the term pattern `p1`. The term pattern can contain a specific term, but it can also contain variables. The term pattern `p2` is the instantiation of a new term pattern which is to replace the terms that match to `p1`. The pattern `p2` can be a term, but also a string. Finally, `c` denotes conditionals that must hold in order for the rule to apply.

In order to transform the source AST to the target AST (or text), we must exhaustively apply a set of rewrite rules. How we traverse the source AST and the order in which to apply the rewrite rules has an effect on the result of the transformation. *Strategies* are a special type of rewrite rules that allow us to define how the rewrite rules are applied. A common strategy is to apply a set of rewrite rules `topdown`. With this strategy, we try to apply a set of rules starting from the top (root) term of the AST, and recursively apply the same set of rules to its children.

```
1 strategies
2   my-strategy = topdown(my-rule)
3 rules
4   my-rule : Sub(a,b) -> Min(a,b)
5   my-rule : Add(a,b) -> Plus(a,b)
```

Listing 3.5: Strategy: change signature

Listing 3.5 shows an example of a strategy that applies a set of rewrite rules top-down. The strategy `my-strategy` attempts to apply `my-rule` to each term in the AST, starting with the root term. Note that `my-rule` is overloaded. The strategy will apply the first rule that matches the term. Applying the strategy will result in the following transformation:

```
Sub(Add(Num(5), Num(3)), Num(4))  →  Min(Plus(Num(5), Num(3)), Num(4)).
```

```
1 strategies
2   to-string = bottomup(to-string-rule)
3 rules
4   to-string-rule : Min(a,b) -> $[([a]-[b])]
5   to-string-rule : Plus(a,b) -> $[([a]+[b])]
6   to-string-rule : Num(a) -> $[[a]]
```

Listing 3.6: Strategy: to string

Listing 3.6 shows an example of a strategy that applies a set of rewrite rules bottom-up. The `to-string` strategy generates a text representation from an AST. Applying the strategy will result in the following transformation:

```
Min(Plus(Num(5), Num(3)), Num(4))  →  ((5+3)-4).
```

```
1 rules
2   my-composed-strategy = my-strategy ; to-string
```

Listing 3.7: Composing strategies

An important feature of Stratego is the ability to compose rules. We can compose rules using the `;`-operator as follows:

```
composed-rule = rule1 ; rule2
```

The `composed-rule` will first apply `rule1` to a term. If `rule1` succeeds, `rule2` is applied to the term. Composition of transformation rules is a very powerful tool when defining complex transformations. We can also apply composition to strategies, as is done in Listing 3.7. Applying `my-composed-strategy` results in the following transformation:

```
Sub(Add(Num(5), Num(3)), Num(4))  →  ((5+3)-4).
```

3.1.4 Transformation patterns

In 2003 Wijngaarden and Visser [25] describe a classification for transformations on programs. They distinguish three different aspects of program transformation: scope, direction, and stages.

The *scope* defines the area of the source and target program affected by the transformation. The area can be local or global. Local refers to a single node in the abstract syntax tree of a program. We can distinguish five cases:

- *Local source to Local target*: A source node is directly translated to the target node.
- *1 to 1*: A special case of Local source to Local target, where the location of the source also corresponds to the location of the target.
- *Global source to Local target*: A single target node is generated by taking information from multiple nodes of the source.
- *Local source to Global target*: Multiple target nodes are formed by taking information from a single source node.
- *Global source to Global target*: Multiple target nodes are formed by taking information from multiple nodes of the source.

The *direction* defines whether the transformation is source driven or target driven. In a source driven transformation, the target is built by traversing the source and applying transformations to the nodes in the source. In a target driven transformation, we traverse the structure of the target and using lookups on the source to find the relevant information.

The *staging* defines the number of steps required to complete the transformation. We define three types of staging:

- *Single-stage*: Transformation to the target is done in a single traversal of the source.
- *Multi-stage modify*: Transformation is done over multiple traversals over the source, each time modifying the source until we have the target.
- *Multi-stage generate*: Each stage generates a piece of the target. The last step in the transformation sequence is to merge all generated pieces to form the target.

3.2 Context

In this section, we perform an analysis of the context. Visser notes in [26] that the first step in designing a DSL is considering *programming patterns* found in the *application domain*. These patterns can be turned into templates, which are essentially programming fragments with holes. The holes are to be filled with *configuration data* to realize different instantiations of the programming patterns. Visser continues to explain that the configuration data can be turned into a DSL.

In our case the application domain is a C++ implementation. We need a reference C++ to identify the programming patterns required for implementing OIL specifications in C++. The creation of a DSL is not relevant in our case since it already exists. How we turn these programming

patterns into templates and configuration data, and how we can transform the OIL DSL into the configuration data is described in Section 3.4.

We describe the state of the OIL architecture in Section 3.2.1. By analysing the current state of the architecture, we gain insight on how to best implement the desired additions. We find a suitable source that we can transform into configuration data and a reference implementation in C++ for analysis of the programming patterns. The source from which we can generate the configuration data is described in Section 3.2.2. Our analysis of the application domain is described in Section 3.2.3.

3.2.1 Existing OIL architecture

There exist two different implementations of the OIL architecture. The first implementation is the *OIL-tooling*. It was developed as a proof of concept using scripting languages like Python and Lua [14]. The OIL-tooling uses OILXML as its source language. It features a graphical user interface (GUI) to visualize OIL specifications, analyze traces of specifications in an interactive way, and generate C++ and mCRL2. This is the most complete implementation available and is used by Océ to generate code C++ for some of their products. However, the code generators are considered ad-hoc implementations that will be difficult to maintain over time.

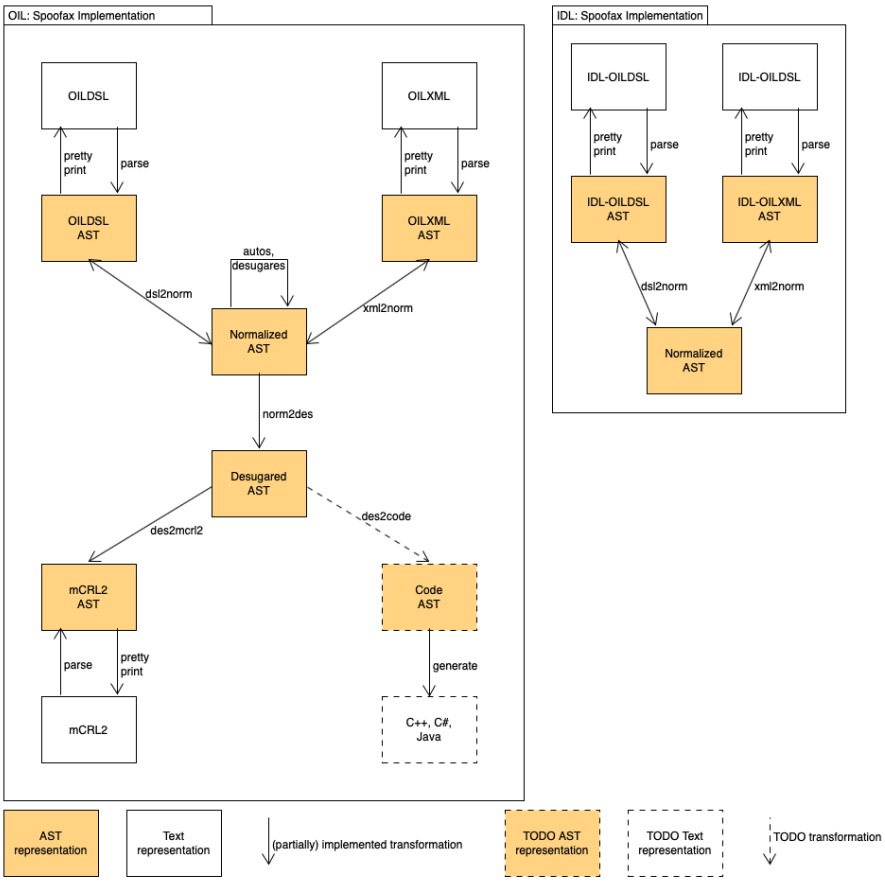


Figure 3.4: Architecture of OIL in Spoofax before our contribution

The second implementation is in Spoofax [15]. The Spoofax implementation is the subject of PhD research by Jasper Denkers and Olav Bunte. Both the development and research are still ongoing at the time of writing.

Figure 3.4 shows the existing OIL architecture implemented in Spoofax. Both the DSL and XML variants of OIL can be parsed and pretty-printed, and both can be transformed to and from the *Normalized AST*. This allows bi-directional transformations between OILXML and OILDSL. The Normalized AST maintains syntactic sugar, i.e. constructs that support expression of functionality that is already provided by the base language in a more compact manner [11]. With the transformation to the *Desugared AST* we remove syntactic sugar, maintaining only the lower-level

constructs of the language. This transformation can only be done in one direction. From the Desugared AST we can transform to the mCRL2 AST, from which we can pretty-print an mCRL2 specification. The transformation from OIL to mCRL2 is relevant for the validation of the code generator discussed in this chapter, we will elaborate on this in Chapter 4.

3.2.2 Transformation source: the Desugared AST

The Desugared AST represents the OIL specification without syntactic sugar. The Desugared AST can have both the DSL and the XML variations of the OIL language as source. Since the code generator does not have to be aware of high-level language constructs, this will serve as a good starting point for the transformation to C++ [11].

3.2.3 Transformation target: C++ implementation

The Python generated C++ implementation is a suitable reference for the analysis of programming patterns. We are interested in how the generated code expresses the semantics of OIL. We analyse each method, starting with the name. The names of methods already contain a lot of valuable information like their purpose, frequency of occurrence, and the OIL term that caused this method to be generated. By analysing the body of the method, we gain insight as to how OIL semantics are expressed in C++. It is also very important to analyse which information from the original OIL specification is needed in order to construct each method.

The following is an example of the analysis of one of the methods found in the reference. The full result of our analysis of the reference can be found in the Appendix in Section A.1.

Example analysis: We analyse the method called `area_condition_`[area name]. One of these methods must be generated for each area defined in the specification. The purpose of this method is to check whether a specific area is active. Depending on the type of area, we must perform different checks. In order for any type of area to be active, its parent area must also be active. Therefore the area condition of the parent area must be checked in all cases. In case the area is of type region, no additional checks are needed. In the case of type state, we must evaluate if the value of the state is equal to the value of the variable of this state. In the case of type scope, we must check whether the invariant holds. Table 3.1 shows the result of the analysis of the `area_condition_`[area name] method.

Element type:	C++ method
Element name:	`area_condition_`[area name]
Occurrence:	Once per area.
Purpose:	Checks if the conditions for a specific area hold. This includes checking if the conditions for the parent area(s) hold.
OIL elements:	Parent area(s), (in case of area type state: corresponding global variable, state value), (in case of area type scope: invariant).

Table 3.1: The analysis of the area condition method in C++

3.3 The big picture: OIL semantics in pseudo-code

In this section, we describe how we can express the semantics of OIL in an object-oriented GPL. Presenting the examples in C++ would make them significantly less readable, therefore we will express the semantics in a pseudo-code that is reminiscent of an object-oriented GPL. The pseudo-code is based on the semantics of OIL in Chapter 2, and the analysis of the reference C++ from the Python implementation in Section 3.2.2.

3.3.1 Global state

The global state object keeps track of the values of the components' global state variables. An instantiation of the global state object defines a valuation over all global state variables. We can access a variable of the global state by `global_state.my_region`, where `global_state` refers to

an instance of the global state, and `my_region` is the name of the variable we want to access. Global state variables can be of several types, e.g. enum types, integers, and interfaces.

```
1 object GlobalState
2   MyRegionType my_region
3   ThatInterface that_interface_ptr
4   int my_int
5   MyRegionType my_var
```

Listing 3.8: Global state object

3.3.2 Enum types

In OIL, regions refer to global state variables. We use enum types to express the type of these variables. For example, we have a region named `my_region`. The region contains the states `my_state` and `my_other_state`, and refers to the global state variable `global_state.my_region`. The type of this variable is `MyRegion`. The values this variable type can hold are `MY_REGION_MY_STATE` and `MY_REGION_MY_OTHER_STATE`. Listing 3.9 shows the enum type `MyRegion`.

```
1 enum MyRegion { MY_REGION_MY_STATE, MY_REGION_MY_OTHER_STATE }
```

Listing 3.9: Enum type for a generic region

3.3.3 Area condition

The area condition checks whether a specific area is currently active. Each area in the model needs its own corresponding area condition function. This is a boolean function that returns true iff the area is active. For an area of any type to be active, the parent of the area must also be active. The parent of an area can be another area, or the root. The root is always considered to be active. States and scopes have additional requirements. Listing 3.10 shows the method to check the area conditions of the root. This method returns `true` as the root is always considered to be active. Next, Listing 3.11 shows the method to check the area conditions of a region. This method returns `area_condition_my_parent()`, which in turn returns a boolean value indicating whether or not the parent area of `my_region` is active. An example of the area condition method for a state is shown in Listing 3.12. In addition to checking whether the parent is active, we also need to check whether the global state holds the correct value. Finally, Listing 3.13 shows the method to check the area conditions of a scope. Next to the area condition of the parent, the invariant of the scope must also hold.

```
1 private bool area_condition_root()
2   return true
```

Listing 3.10: Area condition for the root

```
1 private bool area_condition_my_region()
2   return area_condition_my_parent()
```

Listing 3.11: Region condition

```
1 private bool area_condition_my_state()
2   return area_condition_my_parent()
3          and global_state.my_region == MY_REGION_MY_STATE
```

Listing 3.12: State condition

```
1 private bool area_condition_my_scope()
2   return area_condition_my_parent() and my_invariant
```

Listing 3.13: scope condition

3.3.4 Area update

The area update occurs when a transition fires. When a transition fires, the area update function of the target area is called to update the global state. Listing 3.15, Listing 3.16, and Listing 3.17 show the update functions regions, states and scopes respectively. When an area is updated, the parent area must also be updated. This is shown on line 2 of Listing 3.15, Listing 3.16, and Listing 3.17. In case the parent of an area is the root, no further updates are required. Listing 3.14 shows the area update for the root, which effectively takes no further action. The area update for state areas can assign a new value to the global state variables. An example is shown on line 3 of Listing 3.16, where we assign the value `MY_REGION_MY_STATE` to the global state variable `global_state.my_region`.

```
1 private void area_update_root()
2   // empty
```

Listing 3.14: Update root

```
1 private void area_update_my_region()
2   area_update_my_parent()
```

Listing 3.15: Update region

```
1 private void area_update_my_state()
2   area_update_my_parent()
3   global_state.my_region := MY_REGION_MY_STATE
```

Listing 3.16: Update state

```
1 private void area_update_my_scope()
2   area_update_my_parent()
```

Listing 3.17: Update scope

3.3.5 Preconditions

A transition can fire if its precondition holds. The precondition always includes a check if the source area of a transition is active. A number of additional guards can also be specified. Both the area condition and the guards must evaluate to true for the precondition to hold. Listing 3.18 shows an example of checking the precondition for transition `my_event() #1`. These methods are generated per transition. Note that the `1` and the end of the method name refers to the `#1` in the name of the transition.

```
1 private bool pre_my_event_1()
2   return area_condition_my_event_1_source() and guards
```

Listing 3.18: Transition precondition

3.3.6 Concerns

Next to the precondition, the concerns related to a transition should not be violated. Listing 3.19 shows an example of how to check concerns for an event. Concerns are checked per event, not per transition. In the example, we have two concerns related to the event `my_event`. The concern `MY_CONCERN_1` is related to the transition `my_event() #1`, and `MY_CONCERN_3` is related to transitions `my_event() #2` and `my_event() #3`. Line 2 shows that `MY_CONCERN_1` holds if the precondition for transition `my_event() #1` holds. Line 3 shows that `MY_CONCERN_2` holds if the preconditions for transitions `my_event() #2` or `my_event() #3` hold. Line 5 shows that all concerns related to the event must hold. If a transition has no concern, the concern check evaluates to `true`.

```
1 private bool concerns_my_event()
2   bool MY_CONCERN_1 := pre_my_event_1()
3   bool MY_CONCERN_3 := pre_my_event_2() or pre_my_event_3()
```

```
4   ...
5   return MY_CONCERN_1 and MY_CONCERN_3 and ...
```

Listing 3.19: Method concerns

3.3.7 Process event

The occurrence of an event can lead to one or more transitions firing. One *process event* method is generated for each event in the specification. The process event determines which of the transitions can fire, fires the respective transitions, and checks whether the global state was updated correctly. Listing 3.20 shows an example of such a method for the event `my_event()`.

On line 3 we check whether or not the concerns related to `my_event()` hold. By using an assert, we require that the `concerns_my_event()` method evaluates to `true`, otherwise the component will terminate with an `error`.

On lines 5 and 6 we evaluate the preconditions for each transition labelled with the event `my_event`. We store the results of the evaluations in the corresponding boolean variables.

In OIL, all updates to the global state caused by a single event are performed simultaneously. Area updates are assignments that always have constants on the right-hand side. Actions can use global state variables in the right-hand side of the assignment, meaning they can use values from the current global state to determine values from the next global state. This means that the order in which we perform actions and area updates can influence the result of the update to the global state.

On lines 8–10, we create a collection of variables that will allow us to simulate a simultaneous update to the global state. On lines 12–15, each variable is assigned the result of one action. The variables are named `temp_i_j`, where `i` stands for the i-th transition, and `j` stands for the j-th action of that transition. Note that these temporary variables are created and assigned their values before any update to the global state occurs.

On lines 17–21, we perform the updates for each transition that fires. We first perform each action by assigning the value in the temporary variable to the global state variable. Then we perform the area updates.

On lines 23–28, we check whether each transition that should have fired, did fire. This is determined by checking the postcondition. The postcondition can consist of several asserts. The area update was performed correctly iff the area conditions of the target area hold. An action was performed correctly if the value of the temporary variable equals the value of the global state variable it was assigned to. Additional asserts defined on the transition must also hold. If the postcondition does not hold, the component will terminate with an `error`.

```
1   private void process_my_event()
2     // concerns
3     assert concerns_my_event()
4     // transition fired indicator
5     bool fired_my_event_1 := pre_my_event_1()
6     ...
7     // declare temps
8     int temp_1_1
9     MyRegion temp_1_2
10    ...
11    // assign temps
12    if fired_my_event_1 then
13      temp_1_1 := global_state.my_int + 1
14      temp_1_2 := MY_REGION_MY_STATE
15    ...
16    // update
17    if fired_my_event_1 then
18      global_state.my_int := temp_1_1
19      global_state.my_var := temp_1_2
20      area_update_my_event_1_target()
21    ...
22    // postcondition
```

```
23   if fired_my_event_1 then
24     assert ( global_state.my_int == temp_1_1
25          and global_state.my_var == temp_1_2
26          and asserts
27          and area_condition_my_event_1_target())
28   ...
```

Listing 3.20: Process event

3.3.8 Run-to-completion

The run-to-completion semantics in OIL require that all proactive and silent events that can fire, must fire. If multiple events can fire from a single state, the order in which these events fire is of no consequence due to the confluence of proactive events. Infinite sequences of proactive events are not allowed, which prevents the occurrence infinite loops due to the run-to-completion semantics. Listing 3.21 shows the implementation in pseudo-code. The while-loop will start another iteration iff a proactive event has fired. The `try_my_event()` function returns true iff the event `my_event()` has fired.

```
1  private void run_to_completion()
2    bool fired := true
3    while fired do
4      fired := false
5      // try proactive events
6      fired := try_my_event() or fired
7      ...
```

Listing 3.21: Run-to-completion

3.3.9 Proactive and silent events

The *try event* checks whether a proactive or silent event is possible and calls the event if this is the case. In order to prevent a concern violation, we first check whether the concerns of the proactive event `my_event()` hold on line 3.

On lines 4–9, we call the event if we find a transition labelled with `my_event()` that can fire. In the case of a silent event, the event is part of this component. In the case of a proactive event, the event is part of another component, thus an other object. The pointer to this object is stored in the global state. In the OIL specification, the `self` specifies which global state variable stores the pointer. After calling the event, the try event method returns `true`, informing the `run_to_completion()` method a proactive event was called. In case no event was possible, the method returns `false`.

```
1  private bool try_my_event()
2    // precondition and concerns
3    if concern_my_event() then
4      if pre_my_event_i then
5        ThatInterface self := global_state.that_interface_ptr
6        self.my_event()
7        process_my_event()
8        return true
9    ...
10   return false
```

Listing 3.22: Try event

3.3.10 The public method called by a reactive event

The occurrence of a reactive event is a method call on a component. Listing 3.23 shows an example of such a method. In this case, the event would need to call `my_event()`. Observe that this method is public, so it can be called by the environment. On line 2 we make a call to `process_my_event()`,

this is a private method that will try to perform all transitions related to the event. If it has performed at least one transition successfully, the `run_to_completion()` is called to perform all possible proactive events until the component reaches a stable state at which point it can handle new reactive events.

```
1 public void my_event()
2   process_my_event()
3   run_to_completion()
```

Listing 3.23: The public method that can be called by the environment

3.4 Implementation

In this section, we will explain how the C++ code generator is implemented in the Spoofax language workbench. In order to tackle a complex problem, it is often practical to divide the problem into smaller, less complex problems and solve those instead. When all the smaller problems are solved, we have also solved the complex one. We can apply this divide-and-conquer strategy to code-generation as well. Figure 3.5 shows the transformation pipeline to generate mCRL2 from OILDSL. We can see the *composition* of multiple transformations to create a single complex transformation.

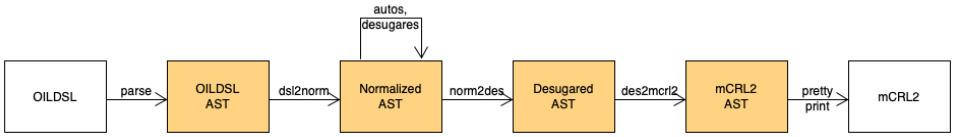


Figure 3.5: The transformation from OILDSL to mCRL2

The transformation from OIL to mCRL2 makes use of four *intermediate representations* in the form of ASTs. These ASTs serve as an interface between transformations. The existence of these ASTs allows us to reuse parts of this pipeline in other transformations. In the existing architecture depicted in Figure 3.4, we can see that both the OILXML and OILDSL can be transformed into the Normalized AST. The Normalized AST serves as an interface between these two DSL representations of OIL. The result is that the `norm2des` transformation can be used in both the transformation from OILDSL to the Desugared AST, and the transformation from OILXML to the Desugared AST.

For the implementation of the transformation from OIL to C++, we will also make use of existing pipelines. The Desugared AST will serve as our interface. This allows the use of existing transformations from both the OILDSL and OILXML to the Desugared AST, as depicted in Figure 3.6.

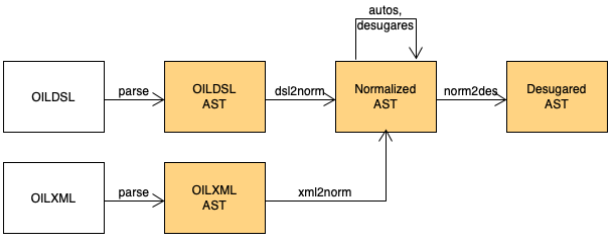


Figure 3.6: The transformation from OIL to the Desugared AST

The implementation of the transformation from the Desugared AST to C++ is shown in Figure 3.7. We require data from two sources: the OIL Desugared AST, and the IDL Normalized AST. The OIL Desugared AST was discussed earlier in this section and contains a normalized, desugared version of an OIL specification. The IDL Normalized AST contains a normalized version of an IDL file that describes interface information for the corresponding OIL specification.

From the Desugared AST we define *des2gpl*, a transformation from the Desugared AST to the GPL AST. We introduce GPL AST as a new intermediate representation. The GPL AST is a

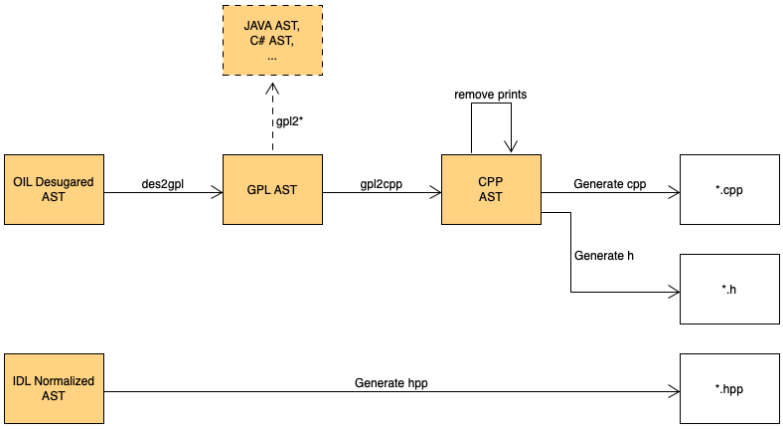


Figure 3.7: Implemented transformations

merger and restructuring of the data from the source. The resulting data-structure is useful for generating object-oriented general-purpose languages. By creating an explicit representation of this data-structure, we can reuse the transformations leading to the GPL AST for code-generation of not only C++, but other object-oriented GPLs as well. We describe the GPL AST and the transformation in more detail in Section 3.4.1 and Section 3.4.2 respectively.

From the GPL AST we define the transformation to the CPP AST, we refer to this transformation as *gpl2cpp*. The CPP AST captures a very small subset of the C++ language in an AST representation. Details about the CPP AST and the transformation can be found in Section 3.4.3 and Section 3.4.4 respectively. The final step is generating the various files containing the C++ code, i.e. the main C++ file (*.cpp), the header file (*.h), and the interface file (*.hpp). This is described in Section 3.4.5.

To help explain the transformation and intermediate representations more clearly, we make use of the *generic area*. We define the generic area as a new type of area that exists only to serve as an example for the following subsections. The generic area has a name, and for the generic area to be active, its parent must be active. It is semantically the same as an area of type scope with its invariant set to true.

Finally, in Section 3.4.6 we provide a guide how to introduce a transformation to a new object-oriented GPL to the code generator.

3.4.1 Intermediate representation: GPL AST

The GPL AST contains the configuration data needed for generating an implementation of an OIL specification in an object-oriented general-purpose language. It is the result of a restructuring of the Desugared AST. Listing 3.24 shows the signature of the `GPLSpec`, the root term of the GPL AST. Most of the terms contained in the `GPLSpec` can be transformed into (lists of) methods in the C++ implementation. Each term contains all information needed to generate a method. These terms are structured according to the programming patterns found in the reference C++. For reference, the complete Stratego definitions of the GLP AST can be found in Appendix A.3.

```
1 GPLSpec : NAME * MODUL * List(GPLProvides) * List(GPLRequires)
2   * List(GPLInterface) * List(GPLEnum) * GPLGlobalState
3   * List(GPLAreaCondition) * List(GPLAreaUpdate)
4   * List(GPLTransitionCondition) * List(GPLConcernCondition)
5   * List(GPLProcess) * List(GPLTryTransition) * GPLOilRun
6   * List(GPLHandleCall) * List(GPLMethod) -> GPLSpec
```

Listing 3.24: Signature of GPL specification from Appendix Section A.3 lines 6–11

We use the generic area as an example to illustrate how we can use the GPL AST as configuration data. Listing 3.25 shows the signature of `GPLAreaCondition`, a term in the GPL AST for the generic area. The term contains an identifier for the area (`AREA-NAME`) and the parent area (`AREA-PARENT`). We can create a template for the method that checks whether an area is

active or not. Listing 3.26 is such a template in pseudo-code. By combining the template and the configuration data in `GPLAreaCondition`, we can generate an instantiation of this method.

```
1 GPLAreaCondition : AREA-NAME * AREA-PARENT -> GPLAreaCondition
```

Listing 3.25: Signature of area condition for the generic area

```
1 private bool area_condition_[AREA-NAME]()
2   return area_condition_[AREA-PARENT]()
```

Listing 3.26: Generic area condition

3.4.2 Transformation: Desugared AST to GPL AST

The transformation from the Desugared AST to the GPL AST (des2gpl) is primarily a restructuring of the terms found in the source AST. For reference, the complete Stratego definitions of the Desugared AST and the GLP AST can be found in Appendix Section A.2 and Section A.3 respectively.

The `des-to-gpl` transformation rule transforms the root term of the Desugared AST (`DESOILModule`) into the root term of GPL AST (`GPLSpec`). Listing 3.27 shows the definition of this rule. The basic idea of this transformation rule is straightforward. It takes a sub-term from `DESOILModule`, applies a number of transformations on the sub-term, and inserts the transformed sub-term into `GPLSpec`.

```
1  des-to-gpl : DESOILModule(interfaces, enums, variables, areas, transitions)
2    -> GPLSpec(name, _, provides, requires, _
3               , enums', globalstate, areaconds'
4               , areaupdates', transitionconds
5               , concernconds, oilprocess, trytransitions
6               , oilrun, handlecalls, methods)
7  where
8    name := $[printer]
9  ; provides := <des2gpl-provides>provreq
10 ; requires := <des2gpl-requires>provreq
11 ; enums':= <des2gpl-enums>enums
12 ; globalstate:= GPLGlobalState(<des2gpl-variables>variables)
13 ; areaconds':= <des2gpl-area-conditions>areas
14 ; areaupdates':= <des2gpl-area-updates>areas
15 ; events := <make-set> <map(extract-event-name(PROCESSID))> transitions
16 ; processdata := <des2gpl-nested-indexed-processes> (events, transitions)
17 ; transitionconds := <des2gpl-transtion-conds>processdata
18 ; concernconds := <des2gpl-concern-conds>processdata
19 ; oilprocess := <des2gpl-processes>processdata
20 ; trytransitions := <des2gpl-try-transitions>processdata
21 ; oilrun := <des2gpl-oilrun>processdata
22 ; handlecalls := <des2gpl-handlecalls>processdata
23 ; methods := <des2gpl-methods>processdata
```

Listing 3.27: The main rule `des-to-gpl` to transform the Desugared AST into the GPL AST

A simple example is the transformation to the `GPLGlobalState` term. The `GPLGlobalState` contains a list of `GPLVariable` terms (see Appendix A.3 line 27, 28). Aside from the name, the `GPLVariable` term is the same as the `DESVariable` term from the Desugared AST. The `des2gpl-variables`-rule in Listing 3.28 transforms a list of `DESVariable` terms into a list of `GPLVariable` terms using the `map()` function from Stratego. In Listing 3.27 on line 12, we apply the `des2gpl-variables` rule on the list of `DESVariable` terms from the Desugared AST to create the list of `GPLVariable` terms, which we then contain in the `GPLGlobalState` term.

```
1 des2gpl-variables = map(\ DESVariable(name, type, exp)
2                       -> GPLVariable(name, type, exp) \)
```

Listing 3.28: Stratego rule to transform the list of global state variables

The transformation becomes more complex when dealing with areas. Lines 13 and 14 of Listing 3.27 show that we use a single rule to transform the list of `DESArea` terms from the Desugared AST to the list of `GPLAreaCondition` and the list of `GPLAreaUpdate`. To illustrate the transformation of areas, we will use the transformation of a generic area term in the Desugared AST (`DESArea`) to an area conditions term in the GPL AST (`GPLAreaCondition`). The signatures are shown in Listing 3.29 and Listing 3.30 respectively. In OIL, areas are stored in a nested structure. This nested structuring of areas is also present in the Desugared AST. The `DESArea` contains a list of areas that stores the areas nested within. On the other hand, `GPLAreaCondition` contains a reference to its parent area. This way the information about the nested structure is maintained, while the terms are stored in a flat list.

```
1 DESArea : NAME * List(DESArea) -> DESArea
```

Listing 3.29: Signature of a generic area from the Desugared AST

```
1 GPLAreaCondition : AREA-NAME * AREA-PARENT -> GPLAreaCondition
```

Listing 3.30: Signature of a generic area from the GPL AST

To transform the nested structure of areas in the Desugared AST into a flat list, we traverse the nested area structure using recursive rules and build the list with each recursive step. Listing 3.31 shows the rules used for creating an area condition for the generic area. For the implementation of specific area types, we can overload `des2gpl-ac-rec-step` on line 5 and define a rule for each type.

```
1 des2gpl-area-conditions = des2gpl-ac-rec(|"root");flatten-list
2
3 des2gpl-ac-rec(|parent) = map( des2gpl-ac-rec-step(|parent) )
4
5 des2gpl-ac-rec-step(|parent) : DESArea(name, childr)
6   -> <conc>( GPLAreaCondition(name, parent), <des2gpl-ac-rec(|name)>childr )
```

Listing 3.31: Stratego rules to create the list of area conditions for a generic area

The relation between transitions, events, and concerns presents another restructuring challenge. In the Desugared AST transitions are stored in a flat list, and each transition contains one event and one or more concerns. The `GPLTransitionCondition` and the `GPLTryTransition` terms are generated once per transition. However, the terms `GPLMethod`, `GPLHandleCall`, `GPLConcernCondition`, and `GPLProcess` are generated once per event. Moreover, the `GPLProcess` and `GPLConcernCondition` terms require a list of all related transitions.

In order to facilitate the generation of these terms, we define an intermediate representation that only exists within the des2gpl transformation. This is the `GPLProcessList` shown in Listing 3.32. The `GPLProcessList` is a list of tuples that consist of an identifier for the event (`GPLProcessDataID`) and a list of transitions from the Desugared AST that are related to the method (`List(DESTransition)`).

```
1 GPLProcessList : List(GPLProcessData) -> GPLProcessList
2 GPLProcessData : GPLProcessDataID * List(DESTransition) -> GPLProcessData
3 GPLProcessDataID : MODUL * INTERFACE * METHODNAME * EVENTTYPE -> GPLProcessDataID
```

Listing 3.32: Signature of the process data term

The `GPLProcessList` term is obtained by a series of transformations on transitions, which is the list of `DESTransition` terms. On line 16 of Listing 3.27 we extract the list of unique events from the transitions, expressed as a list of `GPLProcessDataID` terms. On line 17 we use a rule that takes the list of `GPLProcessDataID` terms and the list of `DESTransition` terms as input, and outputs the `GPLProcessList` term from Listing 3.32. On lines 17–23 we use the `GPLProcessList` term as a source for the remaining terms we need to build the GPL AST.

We consider the transformation from the Desugared AST to the GPL AST to have high complexity. This is evident by the many different transformation patterns that occur in the des2gpl transformation. We can identify several different scopes:

- *1 to 1*: The transformation of global state variables Listing 3.28. The rule transforms one source term directly into one target term, without altering the location in the list.

- *Global source to Local target*: The transformation of areas Listing 3.31. The source is considered global because we need information from the parent term of the `DESArea` term in order to generate the `GPLAreaCondition` term. The target is local because each `DESArea` term generates one `GPLAreaCondition` term. Although we have not gone into detail how we obtain the `GPLProcessList` in this section, the transformation is also a global to local transformation.
- *Local source to Global target*: The des-to-gpl rule in Listing 3.27. The `processdata` is a list of `GPLProcessData` terms. We use this list on lines 17–23 to generate multiple different terms and list of terms, all of which are inserted into the `GPLSpec` term.
- *Global source to Global target*: The `des-to-gpl` rule in Listing 3.27. On line 15 and 16 we have the global to local rules to create the `GPLProcessList` term. Then on lines 17–23 we use the `GPLProcessList` term in a local to global transformation. By composition we have that lines 15–23 define a global to global transformation.

We can identify both directions:

- *Source driven*: The transformation of global state variables Listing 3.28. The direction of the rule is source driven, as we are simply iterating over the source list.
- *Target driven*: The `des-to-gpl` rule in Listing 3.27. On lines 17–23 we query the source term and insert the data into the `GPLSpec` term. We refer to this as a query as we want to extract data from `processdata`, but do not want to transform it as we need the same data for generating other terms. We perform a query by applying a transformation rule on a term and storing the result of the transformation in a different variable. This way we do not alter the source term.

Staging:

- *Single-stage*: The transformation of global state variables Listing 3.28. The direction of the rule is source driven, as we are simply iterating over the source list. Another example is the transformation of areas Listing 3.31, as we make one recursive walk over the source.
- *Multi-stage modify*: The `des-to-gpl` rule in Listing 3.27. The transformation on lines 15–23 was already discussed as a global to global transformation that is defined by the composition of several transformations. This is a transformation where we obtain the target by modifying the source over multiple transformation stages.
- *Multi-stage generate*: The `des-to-gpl` rule in Listing 3.27. On lines 8–23 we create the intermediate terms that are then merged into the `GPLSpec` term.

3.4.3 Intermediate representation: CPP AST

The CPP AST is a representation of the generated C++ code in the form of an AST. The CPP AST represents only a very small subset of the C++ language, including several terms specifically for the generation of C++ in the context of this project. This makes the CPP AST domain-specific. The Stratego definition of the CPP AST is shown in the Appendix A.4.

3.4.4 Transformation: GPL AST to CPP AST

In the transformation from the GPL AST to the CPP AST (gpl2cpp), we generate an AST representation of C++ code based on the configuration data in the GLP AST. This transformation is significantly less complex than the des2gpl, as most of the complex restructuring is done during the transformation to the GPL AST. The gpl2cpp transformation only needs a single top-down traversal of the GPL AST to generate the CPP AST.

Listing 3.33 shows the strategy we apply to traverse the GPL AST. By applying the `gpl-to-cpp` rule on the root term of the GPL AST, we traverse this term and all sub-terms in a top-down order, applying the `gpl2cpp-term` rule to each (sub-)term. The `gpl2cpp-term` rule is overloaded. In fact, most of the rules defined in the gpl2cpp transformation are overloads of the `gpl2cpp-term` rule. Each of these rules matches to a specific term pattern found in the GPL AST.

```
1 gpl-to-cpp = topdown( try( gpl2cpp-term ) )
```

Listing 3.33: gpl2cpp strategy

An example of the `gpl2cpp-term` rule is shown in Listing 3.34. This rule defines a transformation for a generic area condition term from the GPL AST (`GPLAreaCondition`), to a method term in the CPP AST (`CPPDeclMethod`). The signature of the `CPPDeclMethod` term is shown in Listing 3.35. We can define the entire C++ method using the information in `GPLAreaCondition`. The fact that this is a `GPLAreaCondition` term determines the general structure of the method. We know the method is private (line 4), has no parameters (line 6), and returns a boolean (line 7). The method name is a combination of the fact that it is an area condition method and the name given to this instantiation (line 5). This also holds for the body of the method. We know the body consists only of a return statement which makes a method call to the area condition method of the parent (line 8). Listing 3.36 shows an instantiation of `GPLAreaCondition`, and Listing 3.37 shows the results of applying the transformation.

```
1 gpl2cpp-term : GPLAreaCondition(name, parent)
2   -> CPPDeclMethod(access, methodname, parameters, methodtype, body)
3 where
4   access := CPPAccessPrivate()
5 ; methodname := $[oil_area_condition_[name]]
6 ; parameters := []
7 ; methodtype := CPPPrimitive(CPPTypeBoolean())
8 ; body := [CPPReturn(CPPCallMethod($[oil_area_condition_[parent]],[]))]
```

Listing 3.34: Transformation generic area condition

```
1 CPPDeclMethod : CPPAccess * ID * List(CPPArg) * CPPType * List(CPPStat)
2   -> CPPStat
3 CPPReturn : CPPExp -> CPPStat
4 CPPPrint : STRING -> CPPStat
5 CPPAccessPrivate : CPPAccess
6 CPPCallMethod : ID * List(CPPArg) -> CPPExp
```

Listing 3.35: Terms from the CPP AST signature

```
GPLAreaCondition(
  "my_area"
, "my_parent"
)
```

Listing (3.36) GPL AST-term

→

```
CPPDeclMethod(
  Some(CPPAccessPrivate())
, "oil_area_condition_my_area"
, []
, CPPPrimitive(CPPTypeBoolean())
, [
    CPPReturn(
      CPPCallMethod("oil_area_condition_my_parent",[])
    )
  ]
)
```

Listing (3.37) CPP AST-term

The rule in Listing 3.34 will only be applied to a `GPLAreaCondition` term, other terms are not affected by this rule. If we were to apply the `gpl-to-cpp` strategy to an instantiation of the GPL AST, with only defining the rule in Listing 3.34, the resulting AST would have every occurrence of `GPLAreaCondition` replaced by `CPPDeclMethod`. But the rest of the AST would remain the same. For the full transformation it is important we define a `gpl2cpp-term` rule for every term in the GPL AST.

We consider the transformation from the GPL AST to the CPP AST to be considerably less complex then the des2gpl transformation. We analyse which transformation patterns are used. We can identify two scopes:

- *Local source to local target*: The transformation of the area condition shown in Listing 3.34. The rule transforms one source term into one target term, and generates multiple sub-terms in the target term. Since all additional terms are sub-terms of the original target term, we still consider the target to be local. This same classification applies to all variants of the `gpl2cpp-term` rule, as they all have the same structure.
- *1 to 1*: Although no specific examples are given in this section, there also exist 1 to 1 transformations in the glp2cpp.

We can identify transformations in both directions:

- *Source driven*: The main transformation strategy shown in Listing 3.33. The direction of the strategy is source driven, as we are simply traversing the source in a top-down order, applying transformations in place.
- *Target driven*: The transformation of the area condition shown in Listing 3.34. Essentially the target term is a template which we are filling with the configuration data in the source term.

We found two types of staging:

- *Single-stage*: The main transformation strategy shown in Listing 3.33. We apply the transformation in a single traversal of the source term.
- *Multi-stage generate*: The `des-to-gpl` rule in Listing 3.34. On lines 4–6 we create the intermediate terms that are then merged into the `CPPDeclMethod` term.

3.4.5 Code generation: C++

The implementation in C++ is generated from the CPP AST. For reference, the Stratego definitions of the CPP AST can be found in Appendix Section A.4. The scope of gpl2cpp is 1-to-1, as we generate a single string from each term. The direction is source driven, as we build the text by applying transformation rules in place as we traverse the source. The staging is single-stage, as we only need to traverse the source once. We also make use of templates for this transformation.

Listing 3.38 shows a simplified version of the C++ template that is used. The C++ generated from the CPP AST is inserted into line 5 of the template.

```
1  cpp-to-text : CODESpec(a) ->
2  $[#include "Printer.h"
3  namespace tst {
4  ...
5  [statements]
6  ...
7  PrinterPtr Printer::New()
8  {
9    return New(State());
10 }
11 }]
12 where
13   statements := <list-statement> a
```

Listing 3.38: C++ template example

To illustrate how the code generation works, we will continue to use the generic area condition example from Listing 3.37. Listing 3.39 shows some of the transformation rules we need to generate the C++ code in Listing 3.40. The signatures of the terms used in the transformation are shown in Listing 3.35. The `list-statement` rule on line 1 of Listing 3.39 attempts to apply the `build-statement` rule to every term in a list using `filter()` function of Stratego. The `build-statement` rules on lines 3, 7, and 10 define how each term from the CPP AST is transformed into a string. The list of strings is then joined into a single list by `join-strings(|"\r")`, where a return is inserted between each pair of strings.

Notice that the `build-statement` rule on line 3 that matches on the `CPPDeclMethod` term, also calls the `list-statement` on the body of the method. We also use the `comma-sepp-list` rule to join the arguments into a comma separated string.

```
list-statement = filter(build-statement) ; join-strings(|"\r")

build-statement : CPPDeclMethod(_,name,args,type,body)
  -> $[[<build-type>type] Printer::[name]([<comma-sepp-list>args]) {
       ["\r"][<list-statement>body]["\r"]}]

build-statement : CPPReturn(a)
  -> $[return [<build-statement>a];]

build-statement : CPPCallMethod(a,b)
  -> $[[a]([<comma-sepp-list>b])]
```

Listing 3.39: C++ code generation rules

```
bool Printer::oil_area_condition_my_area() {
  return oil_area_condition_my_parent();
}
```

Listing 3.40: C++ code

The CPP AST is only a small subset of the C++ language, and it includes some domain-specific abstractions. A good example of this is the `CPPPrint` term used for logging. In the context of using OIL at Océ, we do not use the standard `cout` to output a string. Rather, Listing 3.41 shows an example of logging that is domain-specific to Océ. The corresponding `build-statement` rule to generate the domain-specific logging statement from the `CPPPrint` term is shown in Listing 3.42. This is one example of several domain-specific abstractions implemented in the CPP AST.

```
OCEprint("Hello world!", LEVEL2);
```

Listing 3.41: C++ code for logging, we use an altered version to prevent IP violation

```
build-statement : CPPPrint(a) -> $[OCEprint([a], LEVEL2);]
```

Listing 3.42: Transformation rule for logging statement

We consider the transformation from the GPL AST to the CPP AST to be considerably less complex then the des2gpl transformation. We analyse which transformation patterns are used. We can identify one scope:

- *Local source to local target*: All of the transformations presented in Listing 3.38 and Listing 3.39. In each rule we take one term and transform that term (and its sub-terms) into a textual representation.

We can identify transformations in both directions:

- *Source driven*: The `list-statement` rule shown in Listing 3.39. The direction of the rule is source driven, as we are simply iterating over the source list, applying transformations in place.
- *Target driven*: The `cpp-to-text` and `build-statement` transformation presented in Listing 3.38 and Listing 3.39 respectively. Essentially the target term is a template which we are filling with the configuration data in the source term.

We found two types of staging:

- *Single-stage*: The `cpp-to-text` and `build-statement` transformation presented in Listing 3.38 and Listing 3.39 respectively. We apply the transformation in a single traversal of the source term.
- *Multi-stage modify*: The `list-statement` rule shown in Listing 3.39. First, we apply the `build-statement` rule to each term in the list. Then we apply the `join-strings(|"\r")` rule to merge all items in the list into a single string.

3.4.6 Guide to introducing a new GPL

In order to obtain a transformation to a new GPL, e.g. Java, we advise the following steps to be taken:

1. *Express programming patterns in Java.* Much like the reference used for C++, we need an example of how to express the programming patterns in Java. This time we will not need it to analyse which programming patterns are present, but rather how the programming patterns are best expressed in Java. Moreover, in the context of Océ we need to establish which formatting rules apply to programs in Java, to apply these to the code generation. We advise this to be done with the help of a domain expert, i.e., a software engineer who is familiar with creating Java implementations for software components in the context of Océ.
2. *Define the JAVA AST.* This will be a small subset of Java. This subset is based on which constructs we need to express the programming patterns (in the context of Océ).
3. *Define the gpl2java transformation.* The structure of this transformation and the patterns used will be very close to the gpl2cpp transformation.
4. *Define the gjava2text transformation.* The structure of this transformation and the patterns used will be very close to the cpp2text transformation.

3.5 Discussion

In this section, we discuss some of our observations, and why certain decisions were made during the development of the Spoofax implementation.

3.5.1 Complexity of transformations

We consider the transformation from the Desugared AST to the GPL AST significantly more complex than the other transformations. By analysing the transformation according to the transformation patterns by Wijngaarden and Visser [25], we wanted to gain insight into why we made this observation. An overview of the patterns we found is presented in Table 3.2.

	des2gpl	gpl2cpp	cpp2text
Scope:			
Local to local	•	•	•
1 to 1	•	•	•
Global to local	•		
Local to global	•		
Global to global	•		
Direction:			
Source driven	•	•	•
Target driven	•	•	•
Staging:			
Single-stage	•	•	•
Multi-stage modify	•		•
Multi-stage generate	•	•	

Table 3.2: Overview of the patterns found in the des2gpl, gpl2cpp, and cpp2text

The first thing that is obvious from the table is that the des2gpl is the only transformation that features all transformation patterns. This is a clear indication of its superior complexity over the gpl2cpp and cpp2text transformations. The main difference between the des2gpl and the other two transformations is found in the scope. All three transformations make use of local source to local target and 1 to 1 transformations. These patterns occur frequently in the transformation of sub-terms. These transformations often have low complexity. The global to local, local to global, and global to global transformations occur exclusively in the des2gpl. This is a logical consequence of the fact that the des2gpl is a complex restructuring of data.

Another difference is the lack of use of overloaded rules in the des2gpl. We argue this is a direct consequence of the global target pattern in the transformation. Overloaded rules can be used if the source term in the transformation dictates the specific transformation rule that must be applied, irrespective of the context. In both the gpl2cpp and cpp2text this is the case. In the des2gpl we have the same source terms occur in multiple different rules. Which rule is to be applied depends on the context.

3.5.2 Implementation the transformation from the CPP AST to text

The current implementation of the transformation from the CPP AST to C++ code is defined manually and uses templates. The advantage of this approach is that it allows us to customize the transformation in great detail. We can change the transformation of a term based on the context. These customisation options can come in useful when the generated code has to adhere to specific formatting rules.

A different approach is to develop a syntax definition for C++ in Spoofax and automatically derive a C++ parser and pretty-printer. This makes the implementation of the transformation better suited for evolution, as only the syntax definition needs to be maintained. The downside is that we rely on the automatically generated pretty-printer, which is significantly more difficult to customize.

The second approach makes a tempting case due to the benefits for evolution, and the fact that we also obtain a parse for free. However, Océ enforces formatting rules for its C++ codebase. The current cpp2text transformation is also relatively low in complexity, making evolution less of a concern. Therefore we currently prefer the flexibility of the manual definition.

3.5.3 Additional intermediate representations vs reuse of transformation rules

Interestingly, the transformation from the Desugared AST to the GLP AST requires some of the same transformations as the transformation from the Desugared AST to mCRL2. One is the transformation of the nested structure of areas to a flat list of areas. Another is the grouping of transitions per event. These two are also the most complex transformations in the GPL AST. For the grouping of transformations per event, we chose to create a variant that can be used by both transformations. Note that this variant is still specific to the Desugared AST as its source. However, for the transformation of areas, we chose to define a new rule specifically for the des2gpl transformation.

We found that making transformation rules generic is not always viable. The problem is that rules are defined from a specific source pattern to a specific target pattern. If the source and target patterns are generic patterns that occur in any signature, e.g. a list or a string, it possible to define generic rules. A good example is a rule that joins a list of strings into a single string and inserts a separator character such as a comma between each of the original strings. In contrast, for the transformation of areas our source pattern is a term defined in the signature of the Desugared AST, and our target patterns is a list of terms from the GPL AST. In case of the transformation to mCRL2, the target pattern is from the mCRL2 AST. Aside from the different target patterns, the transformation rules for areas are very similar in both the des2gpl and the transformation to mCRL2.

Since generalizing rules is difficult, and we still want to avoid duplication, another option is to introduce a new intermediate representation with its own signature directly after the Desugared AST. This AST would structure areas in a flat list, and group transitions per method. An additional representation and transformation will require more artefacts to be maintained. However, this addition also reduces the complexity of both the transformation to the GPL AST and the transformation to mCRL2. This also has a positive effect on maintainability.

Between the two approaches, the introduction of a new intermediate representation has our preference. However, the definition of this new AST is by no means trivial. We must leave the implementation of this new representation to future work.

3.5.4 The OIL to C++ transformation in Python

The Python script that defines the transformation from OIL to C++ was not used for the development of the transformation in Spoofax. This was done because the information extracted from

the reference C++ in combination with the knowledge on OIL semantics was sufficient to construct the transformation in Spoofax. Analysing the Python script would have taken considerable time. The transformation in Python is also implemented in a different context. The script is a direct transformation from OIL to C++. Recall that the implementation we defined in Spoofax is from the Desugared AST to C++. Lastly, Stratego is a functional language designed specifically for transformations, whereas Python is a GPL. The two languages likely require a very different approach to implementing the transformation.

However, the Python implementation does contain valuable information as the transformations it performs appear to be correct. The two implementations can be compared in future work to check if the C++ each generates has the same behaviour.

3.5.5 Definition of the GPL AST

We have chosen to construct the GPL AST such that its structure resembles that of the code we want to generate. However, even though it is called the GPL AST, it contains no constructs generally found in general-purpose languages. The result is that the GPL AST dictates much of the structure of classes and methods in the generated code, but the exact implementation of the methods is left to the transformation to a specific language. One benefit of this approach is that the des2gpl transformation is focussed only on the restructuring of data. Adding control-flow or other terms from a programming language is left to the transformation to a specific language. Another benefit is that it allows a lot of freedom in the transformation to a specific language. For example, the transformation can take into account specific formatting rules for the language. These formatting rules may differ between languages. The drawback is that transformations to different languages will likely be very similar, as they must all express the same behaviour.

A different approach is to define a GPL AST that contains constructs found in every object-oriented language. This would make the transformation to a specific language significantly easier, bringing us even closer to the ability to generate implementations in other GPLs. But it would also make the des2gpl transformation significantly more complex. Given that the GPL AST is already considered the most complex transformation, adding event more complexity is not desirable.

Yet another approach is to keep the current GPL AST and add a representation for a generic object-oriented language between the GPL AST and the CPP AST. For future work, when transformations to different languages are introduced, it will be necessary to revaluate the GPL AST. It may be beneficial to introduce an additional AST between the current GPL AST and the ASTs of specific languages to prevent duplication of transformation rules.

For the moment, without the transformations to other GPLs, we consider the current form of the GPL AST the most viable. This approach keeps the complexity of the des2gpl transformation manageable, while still being useful for generating implementations in other GPLs.

3.6 Lessons learned

In this section, we describe the lessons we have learned from implementing OIL in Spoofax.

3.6.1 Stratego patterns for common transformations

In this section, we give formatting patterns for the most common rules we encountered. We will start with the most common rule pattern we have in the transformations we have defined. Listing 3.43 shows a rule that matches to a specific term and transforms it into another term. The transformation can construct the target term using only the data of the source term. This makes the rule in Listing 3.43 a local to local, and target driven. In many cases, the source term contains other terms, which need to be transformed before they can be placed in the target term. The complexity of the rules applied to the sub-terms can vary greatly, as will be shown in other examples.

```
1 my-rule : Source(a,b) -> Target(a,b')
2 where
3   b' := <my-other-rule>b
```

Listing 3.43: The most common rule pattern

Lists are commonly used throughout the signatures in our context. Listing 3.44 shows the pattern we use to traverse lists. The `map` rule from Stratego traverses the list, and applies `my-rule` to each term is the list. This `my-rule` is generally a rule similar to the rule in Listing 3.43.

```
1 my-list-rule = map( my-rule )
```

Listing 3.44: Traversing lists

Some transformations allow us the use of a top-down (or bottom-up) traversal in combination with an overloaded rule. The pattern is shown in Listing 3.45. We apply this strategy in the gpl2cpp transformation. We can only do this when no restructuring is needed. This can only be done if the transformation is local to local, and source driven.

```
1 my-strategy = topdown( try( my-overloaded-rule ) )
```

Listing 3.45: Local to local, source driven, single-stage

The following patterns illustrate how we can compose complex transformations using the pattern shown in Listing 3.43. Listing 3.46 shows a global to local transformation, where we merge multiple terms in the source to obtain a single term in the target. In Listing 3.47 we query a single term in the source multiple times to obtain multiple different terms in the target. Finally, we can compose the two previous patterns to obtain a global to global transformation. This pattern is shown in Listing 3.48. This last pattern for global to global transformation assumes that the terms of the global target are of a similar structure. An example of this can be seen in the des2gpl transformation.

```
1 my-rule : Source(a,b) -> Target(c)
2 where
3   c := <merge>(a,b)
```

Listing 3.46: Global source to local target

```
1 my-rule : Source(a) -> Target(b,c)
2 where
3   b := <query-b>a
4 ; c := <query-c>a
```

Listing 3.47: Local source to global target

```
1 my-rule : Source(a,b) -> Target(c,d)
2 where
3   x := <merge>(a,b)
4 ; c := <query-c>x
5 ; d := <query-d>x
```

Listing 3.48: Global source to global target

Chapter 4

Model based testing in context of OIL

In this chapter, we investigate how we can validate the correctness of implementations generated with the architecture from Chapter 3, with respect to the corresponding OIL specifications (**RQ3**).

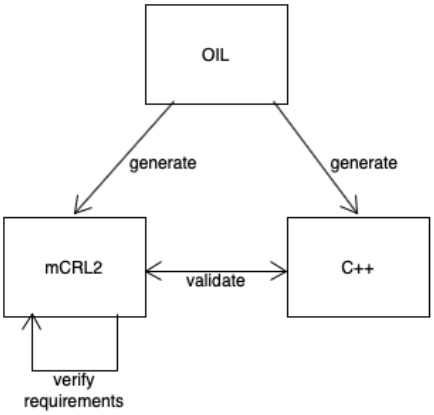


Figure 4.1: Relation between OIL, C++, and mCRL2

Figure 4.1 shows an overview of the transformations that are available to us. The transformation from OIL to mCRL2 is the subject of the promotion research by Olav Bunte. One of the goals of his research is to create a formal semantics for OIL. This research is still ongoing at the time of writing. In the context of this thesis, we assume the transformation from OIL to mCRL2 is correct. With the representation of OIL specifications in mCRL2, the mCRL2 toolset allows access to additional transformations to formal models like linear process specifications (LPS) and labelled transition systems (LTS). Furthermore, we can verify requirements using formal methods. The subject of formal requirement verification is beyond the scope of our research.

There are multiple ways to tackle the validation of transformations (**RQ3**). One initial proposal was to define a transformation from the generated implementation to mCRL2. If this mCRL2 model describes the same behaviour as the mCRL2 model generated directly from OIL, we can validate the correctness of the implementation. However, transforming C++ code directly to mCRL2 is not feasible. We would have to abstract from the C++ code, which is essentially another unverified transformation. Even if the two mCRL2 models turn out to describe the same behaviour, it is unclear what this says about the implementation.

Another approach is to compare our generated C++ to the Python generated C++. This code generator has been in use for some time and has shown to be reliable. It is, therefore, reasonable to assume the transformation in Python is correct. However, it is still considered an ad-hoc implementation that has never been formally validated.

Finally, model-based testing is the approach we have chosen to validate our transformations. Model-based testing revolves around testing the behaviour of an implementation against the be-

haviour described in a formal model. Thanks to the work of Olav Bunte we have a reliable transformation from OIL to mCRL2, which gives us access to verified formal models. And with model-based testing, we investigate the behaviour of the actual C++ implementation.

However, we cannot use model-based testing to verify the correctness of the transformation directly. Instead, we can use it to prove the transformation is incorrect. This can be done by finding an implementation that does not conform to its respective specification.

4.1 Background

In this section, we provide the background for model-based testing. In Section 4.1.1 we describe the framework for model-based testing. We describe the formal models and the properties we need in Section 4.1.2. How we determine the conformance of an implementation with respect to the specification is defined in Section 4.1.3. Finally, Section 4.1.4 and Section 4.1.5 describe the process of deriving and executing tests.

4.1.1 Model-based testing

In this section, we describe the framework used for model-based testing, based on the work by Jan Tretmans [21]. Figure 4.2 shows the framework for model-based testing.

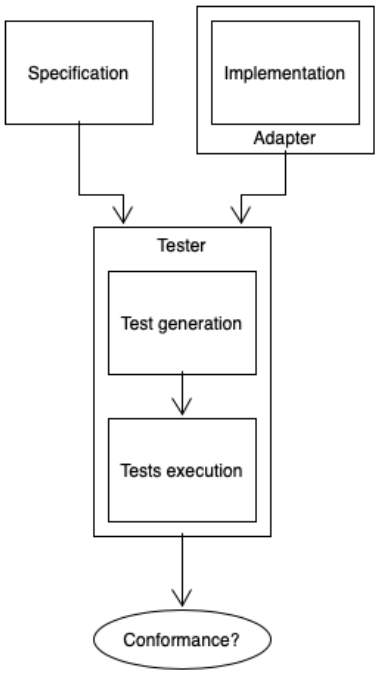


Figure 4.2: Framework for model-based testing

The *Implementation Under Test* (IUT) is the software component being tested. In the context of our research, the IUT is an implementation generated from an OIL specification. We can communicate with the IUT over its interfaces. The IUT is considered a black-box, i.e. we are interested only in the interactions between the IUT and its environment.

The *specification* models the desired behaviour of the IUT. The specification must be expressed in a formal model, e.g. a labelled transition system. Thanks to the work of Olav Bunte we can transform OIL specifications to mCRL2. The mCRL2 toolset allows us to generate formal models from an mCRL2 specification.

We have an IUT and an OIL specification s_{OIL} that models the desired behaviour of the IUT. We want to investigate whether the behaviour of the IUT conforms to the behaviour described by s_{OIL}. However, the IUT is a piece of software. We cannot directly compare executable code to an OIL specification. What we can do is investigate the behaviour of the IUT and compare this to the desired behaviour described in s_{OIL}. In order to do this, we must assume there exists some

OIL model that can capture the behaviour of the IUT. This assumption is referred to as the *test assumption*.

Using the test assumption, we can now investigate whether the behaviour of the IUT conforms to s_{OIL}. The exact meaning of conformance depends on the specific conformance relation that is used. In our case, we use *input-output conformance* (**ioco**) [21], which we describe in further detail in Section 4.1.3.

We assess **ioco** by means of testing. In the context of OIL, interaction between a component and its environment is possible by sending reactive events to the component (input) and observing the resulting proactive events from the component (output). We can derive all valid sequences of events from s_{OIL}. Applying a sequence of inputs to the IUT is referred to as a *test case*. A test case can either pass or fail, with respect to the conformance relation. If the behaviour of the IUT conforms to the behaviour described in s_{OIL}, all test cases will pass.

Model-based testing allows automated generation of test cases using an algorithm. With respect to the implementation relation, we can derive all possible test cases from the specification [21]. However, running all possible test cases is not feasible. In this study, we will limit ourselves to random on-line testing. Informally this means that a tester tool randomly traverses the specification, providing input to the IUT and observing output accordingly. Further details on this topic can be found in Section 4.1.4.

The tester is a software tool that implements model-based testing. It interprets a formal model of the specification, and generates and executes test cases on the IUT with respect to the implementation relation. In order to allow the tester to communicate with the IUT, an adapter is needed. In this study, we use JTorX [3] as our tester. JTorX can communicate with the IUT over the standard input-output streams. An adapter is needed to translate the strings sent over the input stream to function calls, and the function calls by the IUT to strings and send them over the output stream.

4.1.2 Labelled transition systems

In this section, we describe the formal models we need for model-based testing. We use a variant of the labelled transition systems (LTS) to formally describe the behaviour of our software components. The LTS is a commonly used model in the formal analysis of systems, which includes model-based testing. In Definition 4.1.1 we give the formal definition for an LTS. In the context of software components, it can be more natural to refer to a label of a transition as an action. Going forward, the terms label and action both refer to an element of L in an LTS.

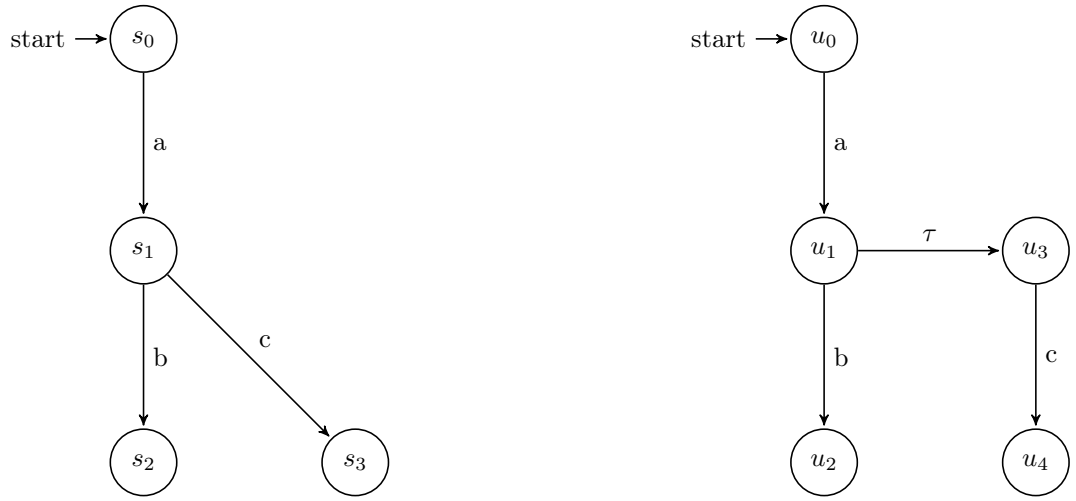
Definition 4.1.1. (*Labelled transition system*, LTS). A *labelled transition system* (LTS) is a 4-tuple $\langle Q, L, T, q_0 \rangle$ where
– Q is a non-empty, countable set of states
– L is a countable set of labels (can also be referred to as the set of actions)
– $T \subseteq Q \times L \times Q$ is the transition relation
– q_0 is the initial state

An LTS consists of states, actions, transitions, and the initial state. Let $(q, \mu, q') \in T$ be a transition in an LTS, such that $q, q' \in Q$ and $\mu \in L$. An alternative notation for this transition is $q \xrightarrow{\mu} q'$. Figure 4.3 shows an example of an LTS. The first LTS S in figure 4.3a consists of three transitions: $s_0 \xrightarrow{a} s_1$, $s_1 \xrightarrow{b} s_2$, and $s_1 \xrightarrow{c} s_3$.

An LTS can accept a sequence of actions in L. A sequence of actions is an element of L^*, where L^* is the set of all finite sequences over the set of actions L. The transitions of an LTS specify which sequence the LTS can accept. The sequences S can accept are: the empty sequence ϵ, a, ab, and ac.

We represent the internal action by τ, such that $\tau \notin L$. The second LTS U in Figure 4.3b features this internal action τ in one of its transitions. A sequence that includes the internal action works as follows. After LTS U is given an a, it can accept a b or a c by making the τ step. This behaviour appears the same as that of LTS S. However, LTS U can internally decide to move from u_1 to u_3 before a b or a c is given. Once in u_3, the LTS can only accept c. The sequences that U can accept are: the empty sequence ϵ, a, ab, $a\tau$, and $a\tau c$.

Depending on the context, the occurrence of an internal action in a sequence may not be of interest. A transition relation with a double arrow ($\Rightarrow$) indicates τ-abstracted transition sequence, meaning that only the observable actions are noted. The formal definition for the τ-abstracted



(a) LTS S (b) LTS U

Figure 4.3: Example LTS

transition relation is given in Definition 4.1.2. The τ-abstracted sequences U can accept are: the empty sequence ϵ, a, ab, and ac. Notice that if we abstract away the τ action, LTS S and LTS U accept the same sequences.

Definition 4.1.2. (τ-*abstracted transition sequence relation*, $\Rightarrow$). Let the τ-abstracted transition relation be denoted by $\Rightarrow \subseteq Q \times L^* \times Q$, where L^* denotes the set of all finite sequences over the set of labels L, and $\sigma \in L^*$. We use the Plotkin-style deduction rules to describe the relation $\Rightarrow$ [20]:

$$\frac{}{q \overset{\epsilon}{\Rightarrow} q} \qquad \frac{q \overset{\sigma}{\Rightarrow} q'' \quad q'' \xrightarrow{x} q' \quad x \neq \tau}{q \overset{\sigma x}{\Longrightarrow} q'} \qquad \frac{q \overset{\sigma}{\Rightarrow} q'' \quad q'' \xrightarrow{\tau} q'}{q \overset{\sigma}{\Rightarrow} q'}$$

Definition 4.1.3 describes a set of functions that help us define the more complex concepts later in this section. The *enabled actions* of a state q ($\texttt{init}(q)$), is the set of actions including the internal action τ that is allowed in that state. The *weakly enabled actions* ($\texttt{Sinit}(q)$) differ from the enabled actions in that the actions that can be reached by one or more τ-steps are also included. However, the set of weakly enabled actions does not include the τ-step itself. The *traces* of a state q ($\texttt{traces}(q)$) is the set of all sequences that are accepted by q. Finally, we have q **after** σ, which denotes the set of states that is reachable from state q with sequence σ.

Definition 4.1.3. Let q, q' be states of an LTS, and $\sigma \in L^*$, where L^* is the set of all finite sequences over the set of labels L.

$$\begin{aligned}
\texttt{init}(q) &=_{def} \{\ \mu \in L \cup \{\tau\} \mid \exists q' : q \xrightarrow{\mu} q'\ \} \\
\texttt{Sinit}(q) &=_{def} \{\ \mu \in L \mid \exists q' : q \overset{\mu}{\Rightarrow} q'\ \} \\
\texttt{traces}(q) &=_{def} \{\ \sigma \in L^* \mid \exists q' : q \overset{\sigma}{\Rightarrow} q'\ \} \\
q\ \texttt{after}\ \sigma &=_{def} \{\ q' \mid q \overset{\sigma}{\Rightarrow} q'\ \}
\end{aligned}$$

We extend the LTS with the notion of input and output labels. We refer to such a system as an *input-output labelled transition system* or IOLTS. The input and output labels are two disjoint sets of labels, meaning that a label is either part of one set or the other. The IOLTS is commonly used to model software components that communicate via input and output messages.

Definition 4.1.4. (*input-output labelled transition system*, IOLTS). An *input-output labelled transition system* (IOLTS) is a 5-tuple $\langle Q, L_I, L_U, T, q_0 \rangle$ where:
– $\langle Q, L_I \cup L_U, T, q_0 \rangle$ is an LTS
– L_I and L_U are countable sets of input labels and output labels respectively, which are disjoint: $L_I \cap L_U = \emptyset$

Software components often require input to perform actions. Once the input is given, the component proceeds to process the input, give the respective output, and wait for new input. We can model such states with an IOLTS. A state that is unable to change autonomously unless new input is given, is referred to as a *quiescent state*.

Definition 4.1.5. (*Quiescent state*, $\delta(q)$). Let $\langle Q, L_I, L_U, T, q_0 \rangle$ be an IOLTS. A state $q \in Q$ is quiescent, denoted by $\delta(q)$, iff $\texttt{init}(q) \subseteq L_I$.

A state is considered *input enabled* when it can accept all inputs from the set of input labels. Input enabledness is an important notion for model-based testing, specifically for the IUT. The formal definition is found in Definition 4.1.6.

Definition 4.1.6. (*Input enabled*). Let $\langle Q, L_I, L_U, T, q_0 \rangle$ be an IOLTS. A state $q \in Q$ is *input enabled* iff $L_I \subseteq \texttt{Sinit}(q)$.

The *internal choice input-output labelled transition system* is an IOLTS in which only quiescent states accept inputs. This means that we can only feed the IOLTS$^{\sqcap}$ input when it cannot change state autonomously. Definition 4.1.7 gives the formal definition of the IOLTS$^{\sqcap}$. [18]

Definition 4.1.7. (*Internal choice input-output labelled transition system*, IOLTS$^{\sqcap}$). An IOLTS $\langle Q, L_I, L_U, T, q_0 \rangle$ is an *internal choice input-output labelled transition system* IOLTS$^{\sqcap}$ iff only quiescent states may accept inputs, i.e.,

$$\forall q \in Q : \texttt{init}(q) \cap L_I \neq \emptyset \implies \delta(q)$$

The *Internal choice input-output transition system* (IOTS$^{\sqcap}$) is the final, and most important model we present in this section. The IOTS$^{\sqcap}$ is an IOLTS$^{\sqcap}$ in which every quiescent state is input enabled. The formal definition is given in Definition 4.1.8. [18]

Definition 4.1.8. (*Internal choice input-output transition system*, IOTS$^{\sqcap}$). An IOLTS$^{\sqcap}$ $\langle Q, L_I, L_U, T, q_0 \rangle$ is an *internal choice input-output transition system* IOTS$^{\sqcap}$ iff every quiescent state is input enabled, i.e.,

$$\forall q \in Q : \delta(q) \implies L_I \subseteq \texttt{Sinit}(q)$$

Informally, the IOTS$^{\sqcap}$ is a transition system with input and output labels and two very specific properties. The first property it inherits from the IOLTS$^{\sqcap}$, and it requires that only quiescent states may accept inputs. The second property states that every quiescent state must be input enabled. This means that if the IOTS$^{\sqcap}$ accepts inputs, it can accept any input from the set of input labels. These two properties are essential for applying model-based testing in our context, as will be explained in Section 4.1.3.

4.1.3 The implementation relation ioco for asynchronous communication

In this section, we give a definition of the implementation relation for input-output conformance for asynchronous communication (**ioco**$^{\sqcap,\sqcap}$), and how we can implement **ioco**$^{\sqcap,\sqcap}$ in practice.

In the original definition by Tretmans [21], **ioco**-testing assumes synchronous communication between tester and implementation. However, in our test environment, we have asynchronous communication between the tester and the IUT. The tester and the IUT communicate over the standard input and output streams, which effectively function as FIFO buffers. Asynchronous communication can lead to problems in classic **ioco**. The problem occurs when we can do both input and output actions from the same state. In an asynchronous setting, the tester can decide to do an input at the same time the IUT can decide to do an output. In Section 4.3.1 we perform an experiment that explains the issue in more detail. Examples can also be found in the literature [2].

In her doctoral thesis and related article from 2014, Noroozi presents a solution for the asynchronous setting [18, 19]. Noroozi presents a formal proof that the **ioco** relation is valid independent of synchronous or asynchronous communication when the specification and the IUT can be modelled as an IOLTS$^{\sqcap}$ (Definition 4.1.7) and an IOTS$^{\sqcap}$ (Definition 4.1.8) respectively. Using the proof by Noroozi we can now apply model-based testing in an asynchronous setting by using the **ioco**$^{\sqcap,\sqcap}$ relation.

Before we proceed with the definition of the implementation relation, we augment the transition sequence relation $\Rightarrow$ with the observations of quiescence. The observation of quiescence in a sequence is denoted by $\delta \notin L \cup \{\tau\}$. Much like abstracting away the internal action τ in $\Rightarrow$, we need a transition relation that also abstracts from the observation of quiescence in a sequence. We give the formal definition of this τ-δ-abstracted relation in Definition 4.1.9.

Definition 4.1.9. (*τ-δ-abstracted transition sequence relation*, $\Rightarrow_\delta$). Let the τ-δ-abstracted transition relation be denoted by $\Rightarrow_\delta \subseteq Q \times L^* \times Q$, where L^* denotes the set of all finite sequences over the set of labels L, and $\sigma, \rho \in L^*$. We use the Plotkin-style deduction rules to describe the relation $\Rightarrow_\delta$ [20]:

$$\frac{q \overset{\sigma}{\Rightarrow} q'}{q \overset{\sigma}{\Rightarrow}_\delta q'} \qquad \frac{\delta(q)}{q \overset{\delta}{\Rightarrow}_\delta q} \qquad \frac{q \overset{\sigma}{\Rightarrow}_\delta q'' \quad q'' \overset{\rho}{\Rightarrow}_\delta q'}{q \overset{\sigma\rho}{\Longrightarrow}_\delta q'}$$

Using the transition relation $\Rightarrow_\delta$, we present two more concepts: the *suspension traces* and the *set of possible outputs*. The suspension traces (Definition 4.1.10) are sequences of actions where both the internal action τ and the observation of quiescence δ are abstracted away. The set of possible outputs gives all possible outputs in a specific state or set of states (Definition 4.1.11).

Definition 4.1.10. (*Suspension traces*, `Straces`). Let q, q' be states in an IOLTS, and let L be the set of labels, then

$$\texttt{Straces}(q) =_{def} \{\ \sigma \in L^* \mid \exists q' : q \overset{\sigma}{\Rightarrow}_\delta q'\ \}$$

Definition 4.1.11. (*Set of possible outputs*, `out`). Let q, q' be states in an IOLTS, and let Q be a set of states, then

$$\begin{aligned} \texttt{out}(q) &=_{def} \{\ x \in (L_U \cup \{\delta\}) \mid \exists q' : q \overset{x}{\Rightarrow}_\delta q'\ \} \\ \texttt{out}(Q) &=_{def} \bigcup \{\ \texttt{out}(q) \mid q \in Q\ \} \end{aligned}$$

We present the formal definition of the implementation relation for *input-output conformance for asynchronous communication* (**ioco**$^{\sqcap,\sqcap}$) in Definition 4.1.12.

Definition 4.1.12. (*input-output conformance for asynchronous communication*, **ioco**$^{\sqcap,\sqcap}$). Let IUT be an existing implementation. Let specification IOLTS$^{\sqcap}$ s model the desired behaviour of the IUT. Using the test assumption, we assume there exists an IOTS$^{\sqcap}$ i such that i models the behaviour of the IUT. Then **ioco**$^{\sqcap,\sqcap}$ is defined as follows [18, 19]:

$$i\ \textbf{ioco}^{\sqcap,\sqcap}\ s =_{def} \forall \sigma \in \texttt{Straces}(s) :\ \texttt{out}(i\ \texttt{after}\ \sigma) \subseteq \texttt{out}(s\ \texttt{after}\ \sigma)$$

Informally, the implementation relation **ioco**$^{\sqcap,\sqcap}$ means that the tester must be able to predict the behaviour of the IUT based on the information in the model. If the IUT exhibits behaviour that the tester could not predict with respect to the given model, **ioco**$^{\sqcap,\sqcap}$ does not hold. The $^{\sqcap,\sqcap}$ superscript refers to the model of the specification and the model that describes the behaviour of the IUT. Both models must be internal choice, meaning that only quiescent states may accept inputs. This is crucial for applying model-based testing in a setting with asynchronous communication.

4.1.4 Random on-line testing

Tretmans proposes an algorithm for automated testing to check for **ioco**. Tretmans describes this in **Algorithm 1** [21]. This algorithm serves as a basis for random on-line testing. Essentially the algorithm presents three choices at each recursive step:

Let t denote the test case.

1. Terminate t, and return **pass**.
2. Test the implementation by supplying an input $a \in \textbf{init}(s)$, with s being the current set of states of the specification. The algorithm is then recursively applied. Additionally, t is prepared to accept any output the implementation supplies before supplying a. If the output is not allowed, the test case terminates with **fail**.
3. Check the next output of the implementation. If no output arrives, it observes quiescence. If the output is not allowed with respect to the specification, the test case terminates with **fail**.

4.1.5 Model coverage

Model coverage denotes how many of the transitions and states of the specification have been covered during a test. A state or transition is considered covered if it was visited at least once. It is important to note that full coverage of a model does not imply all possible execution paths have been taken. Coverage is a much weaker notion.

Model coverage can also be used during on-line testing in order to obtain full coverage. During a test run, the algorithm can use the coverage information obtained thus far to derive the following step [4].

4.2 Implementation of model-based testing for OIL

In this section, we describe how we can apply **ioco**-testing in the context of OIL. Furthermore, we describe how we can apply model-based testing on OIL specifications and implementations using JTorX.

4.2.1 OIL as internal choice input-output transition system

In this section, we show that we can apply model-based testing in the context of OIL. The **ioco**$^{\sqcap,\sqcap}$ theory requires the IUT to behave as an IOTS$^{\sqcap}$. We first present a mapping between OIL and an IOLTS. This mapping is part of the work by Olav Bunte, who is working on the formal semantics of OIL. Next, we show that OIL specifications can also be expressed as IOTS$^{\sqcap}$, which is essentially an IOLTS with additional properties.

Thanks to the work of Olav Bunte, mapping OIL to an IOLTS is straightforward. With the transformation from OIL to mCRL2, we can generate an LTS using the mCRL2 toolset. Using event information captured in the labels, we can convert the LTS to an IOLTS. The mapping from OIL to an IOLTS is as follows. Given an OIL specification in the form of an LTS $\langle Q, L, T, q_0 \rangle$, we construct the IOLTS $\langle Q, L_I, L_U, T', q_0 \rangle$ such that:

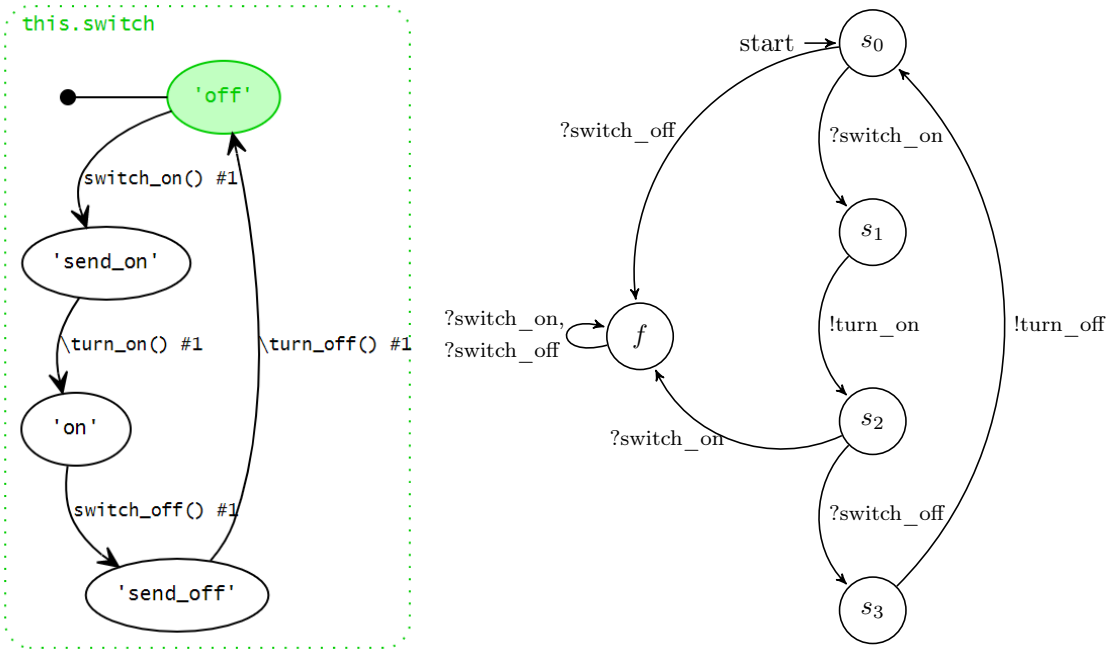
- $L_I : \mu \in L_I$ iff $\mu \in L$ and μ is a reactive event
- $L_U : \mu \in L_U$ iff $\mu \in L$ and μ is a proactive event that is not silent
- $T' : (q, \mu, q') \in T'$ iff $(q, \mu, q') \in T$ and μ is not silent, and $(q, \tau, q') \in T'$ iff $(q, \mu, q') \in T$ and μ is silent

In the mapping presented above, we consider silent events to be the internal actions of the component, hence we map them to τ. Events that implicitly exist due to concerns, i.e. illegal events that lead to the failure state, are also present in the resulting IOLTS.

By using an example we show how we can construct an IOLTS from an OIL specification. Then we will show that the IOLTS we have constructed is also an IOTS$^{\sqcap}$, and we will argue that we can generalize this to all OIL component specifications.

For our example, we will use the OIL component specification in Figure 4.4a and the corresponding IOLTS in Figure 4.4b. Figure 4.4a shows a simple OIL component specification of an on/off switch. The switch can turn a generic device on or off. We can call the event `switch_on`, and the switch will proactively call `turn_on` on the generic device. The `switch_off` event works in a similar manner. Recall that as this is a component specification, each transition is related to a generic concern.

Figure 4.4b shows the IOLTS that models the same behaviour as the on/off switch in OIL. Reactive events and proactive events are translated to input and output labels, denoted with the prefixes "?" and "!" respectively. The states of the OIL specification `off`, `send_on`, `on`, and `send_off`, are mapped to s_0, s_1, s_2, and s_3 respectively. Illegal termination of the component is represented by the failure state f. Notice the events that lead to f, these are the illegal events. While the switch is on, `?switch_on` will lead to f. The same holds for `?switch_off` while the switch is off.



(a) (b)

Figure 4.4: An on/off switch in OIL (a) and as an IOLTS (b)

Now we will argue that the IOLTS of any OIL specification is also an IOTS$^{\sqcap}$, and we will use the IOLTS in Figure 4.4b as an example. By Definition 4.1.8 the following two requirements must hold:

Requirement 1. Only quiescent states may accept inputs.

Requirement 2. All quiescent states must be input enabled.

Requirement 1: We must show non-quiescent states do not accept input. We argue that due to the run-to-completion semantics (Section 2.2.2), there cannot exist a state that has both reactive events (input) and proactive events (output or τ) enabled. Take any non-quiescent state from an IOLTS of an OIL component specification. A state is non-quiescent iff there exists an outgoing proactive event. If this state has an outgoing proactive event, then this event has priority over any outgoing reactive events that exist in this state due to the run-to-completion semantics of OIL. Hence, an outgoing reactive event cannot exist in any non-quiescent state.

In the example, we have the quiescent states s_0, s_2, and f. Notice that only inputs are enabled in these states, no outputs are available. The remaining states s_1 and s_3 only have output labels enabled.

Requirement 2: We must show the quiescent states are input enabled. By requirement 1 we have that if a state has an outgoing reactive event, the state is quiescent. If an OIL specification is a component specification, this implies that all events have at least one concern. This implies the existence of illegal events for all quiescent states. The set of outgoing illegal events possible in a quiescent state contains all reactive events that occur in the model, except the outgoing reactive events for which there exists a corresponding transition in the OIL specification (Section 2.2.3). For any given quiescent state, if we combine the set of outgoing illegal events with the set of outgoing reactive events, we obtain the set of all reactive events that occur in the model. This makes all quiescent states in an OIL model input enabled.

Note that it is only for the quiescent states that we model the implied illegal events that lead to a failure state. This is due to the run-to-completion semantics, which prioritizes proactive events over reactive events. Illegal events are defined as a subset of the reactive events.

We can see the illegal events in Figure 4.4b. Recall that states s_0, s_2, and f of Figure 4.4b are quiescent states. While the switch is off (s_0), `?switch_off` will lead to f. The same holds for `?switch_on` while the switch is on (s_2). Due to illegal events, all quiescent states in Figure 4.4b

are input enabled.

With the argument presented above for both requirements, we conclude it is reasonable to assume that all OIL component specifications express behaviour that can be modelled as an IOTS$^{\sqcap}$.

4.2.2 Partial specifications for experiments

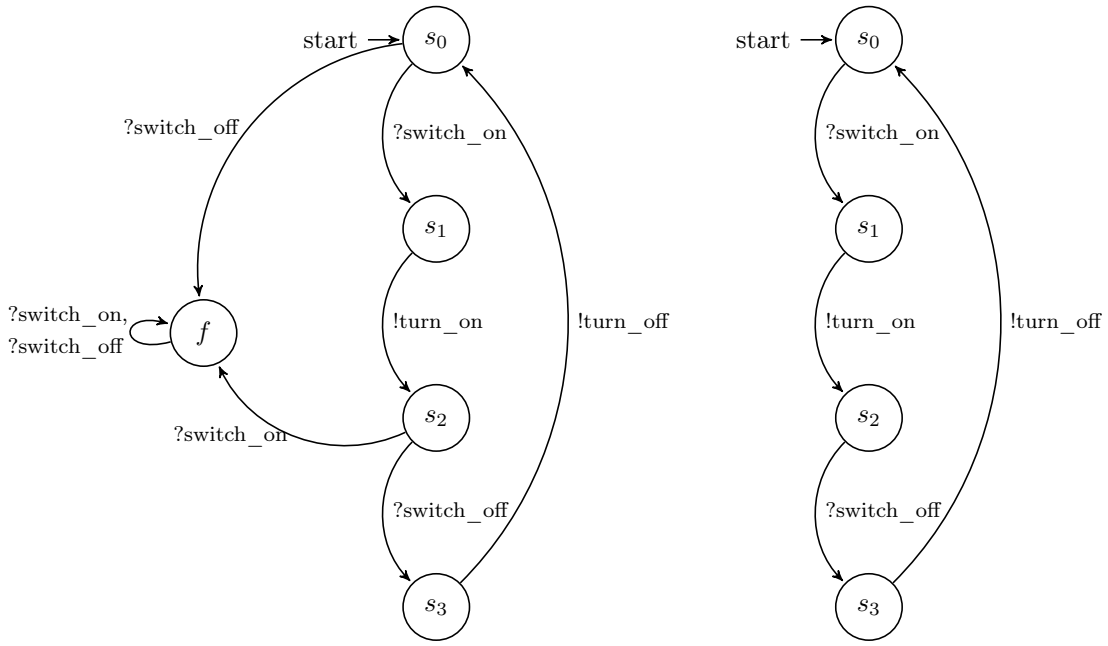


Figure 4.5: Full specification (a) and partial specification (b)

The definition for **ioco**$^{\sqcap,\sqcap}$ (Definition 4.1.12) does not require the specification to be input enabled. The **ioco**$^{\sqcap,\sqcap}$ relation holds when, for all traces in the specification, the tester is able to predict the response given by the IUT after each trace. If a trace is not in the specification, this trace will not be tested. Hence the behaviour of the implementation after an unspecified trace is allowed to be anything. This allows the use of partial specifications in **ioco**$^{\sqcap,\sqcap}$.

The **ioco**$^{\sqcap,\sqcap}$ relation requires specification s in IOLTS$^{\sqcap}$, an OIL specification without illegal events is also in IOLTS$^{\sqcap}$. Such a specification shows only good weather behaviour. Figure 4.5b shows an example of such a partial specification that only shows good weather behaviour. Figure 4.5a shows the full specification.

Testing exclusively good weather behaviour has several practical benefits:

- We avoid the illegal events, which lead to a sink state from which the implementation cannot recover without a hard reset. Sink states are very impractical for random on-line testing in JTorX, as it does not feature automated resets of the IUT.
- We have fewer transitions to cover in a test, potentially making testing significantly faster.

4.2.3 The sentinel

In model-based testing we approach the IUT as a black-box, i.e., the tester (JTorX) has and needs no knowledge of the internal working of the IUT. In order to test the behaviour of the IUT, we must feed the IUT input and observe the output the IUT returns. In other words, JTorX needs output to determine the IUT is working correctly.

In OIL, we can define a component specification that has no output. In this case, the only output the IUT can give is quiescence. Model-based testing cannot differentiate between actual quiescence and a livelock, e.g., a situation where the IUT stuck in an infinite loop and is unable to process new events. In both cases, the IUT returns quiescence. However, in the case of an

infinite loop, JTorX will continue to flood the standard input queue. In a practical setting, a component that has no observable behaviour is unlikely. However, the problem remains when there exists a sequence of exclusively reactive events. If at some point during this sequence the IUT gives unexpected output, e.g. quiescence when a proactive event is expected, JTorX will correctly conclude non-conformance. However, it is not clear which reactive event in the sequence caused the test to fail. The inability to test specifications that have no output, lack of precision in non-conformance detection, and possible flooding of the standard input queue are undesirable properties.

In an attempt to remedy these issues, we introduce the *sentinel*. We modify the IUT to output a sentinel when it has finished processing and is ready to receive new input. We translate this to the following requirements for the IOLTS$^{\sqcap}$ model of the specification:

1. When the IUT is initialised and the tester has not taken any other action, the following sequence of events must occur: zero or more proactive events, followed by one sentinel.
2. When the tester sends a reactive event, the following sequence of events must occur: zero or more proactive events, followed by one sentinel.
3. In any sequence, a reactive event is always directly preceded by a sentinel.
4. In any sequence, a sentinel can never be directly followed by a proactive event.

Requirements 1,2, and 3 prevent flooding of the input queue by disallowing any sequence of more than one reactive event. Requirement 4 makes sure the IUT is quiescent, and by property of IOTS$^{\sqcap}$ therefore also input enabled, after the sentinel is observed. This means that the input queue should be empty at this point, and any input given can be directly processed by the IUT. Any failures will be the direct result of the last input. When testing specifications that have no proactive events, the sentinel will inform the tester that the IUT has at least accepted and processed the last input. Still, the sentinel gives no insight into the specific internal behaviour of the IUT, so the value of testing this type of specification remains questionable.

4.2.4 Specifications for JTorX

In order to prepare our specification for testing in JTorX, we perform a series of transformations during post-processing. We do this using a Python script on the LTS in the Aldebaran file format. We make the following changes to the LTS in post-processing:

- Rename silent events to `tau`
- Add "?" and "!" markers to the labels of reactive and proactive events, so JTorX can recognize them as input or output respectively
- Add the sentinel to the specification

The first two transformations are trivial since we can identify reactive and (silent) proactive events by their labels. The addition of the sentinel is more involved. Figure 4.6 shows an IOLTS representing a partial specification before the sentinel was added. Figure 4.7 shows the same specification with the sentinel. Listing 4.1 shows the algorithm for adding the sentinel to the model in pseudo-code. Line 3 shows the algorithm visits each state in the original IOLTS once. Line 4 requires that the state has an outgoing reactive event. With lines 5–8 we create a new state and a new sentinel transition from the original state to the new state. Lines 9 and 10 find all reactive events with the original state as a source, and change the source of each event to the new state. Lines 11–14 add all new states and transitions to the original IOLTS.

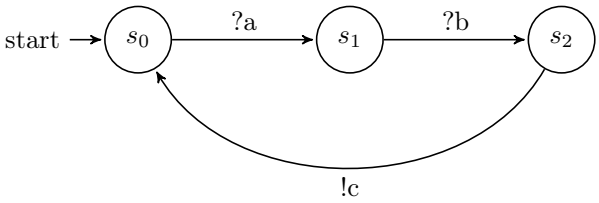


Figure 4.6: LTS s without sentinel

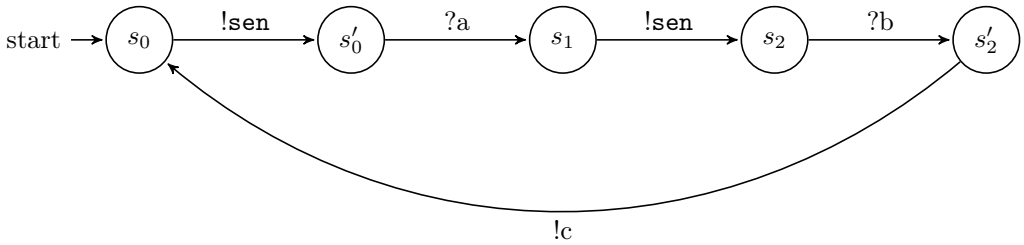


Figure 4.7: LTS s with sentinel `sen`

```
1  Given an IOLTS (Q, Li, Lu, T, q0) as input
2  Let Qnew and Tnew be empty sets
3  for each q ∈ Q do
4    if exists (q,l,r) ∈ T with l ∈ Li and r ∈ Q then
5      qnew := new state not in Q ∪ Qnew
6      Qnew := Qnew ∪ {qnew}
7      tnew := (q,"!sen",qnew)
8      Tnew := Tnew ∪ {tnew}
9      for each (q,l,r) ∈ T with l ∈ Li and r ∈ Q do
10       T :=  T \ {(q,l,r)}
11       Tnew := Tnew ∪ {(qnew,l,r)}
12 Q' := Q ∪ Qnew
13 Lu' := Lu ∪ {"!sen"}
14 T' := T ∪ Tnew
15 return (Q', Li, Lu', T', q0)
```

Listing 4.1: Algorithm to add sentinel to specification

4.2.5 Adapters for JTorX

This section describes the implementation of the adapter. The adapter allows communication between the tester and the IUT. JTorX, the tester used for our research, can communicate with the IUT over the standard input and output streams. We require an adapter that can translate reactive events sent over the standard input stream to method calls on the IUT, and method calls made by the IUT to proactive events on the standard output stream.

A generic example of an adapter for OIL implementations is shown in Listing 4.2. On line 1 we create a new instance of the IUT. It is important to note that execution must take place in sequence. Allowing parallel execution, e.g. by running the IUT on a different thread, will result in illegal behaviour.

In order for the **ioco**$^{\sqcap,\sqcap}$ relation to apply, we assume the behaviour of the IUT can be expressed as an IOTS$^{\sqcap}$. This means the IUT must be input enabled in all quiescent states. This input enabledness is implemented with the `while`-loop on line 3. The loop continues to monitor the input stream (`std.in`) for new inputs. If a new input is sent by JTorX, the adapter will attempt to match it to one of the reactive events of the model (lines 6 and 10). If a match is found, the adapter will call the corresponding method on the IUT (lines 8 and 12), and continue the `while`-loop. We construct such an `if`-statement in the adapter to include all labels in the set of input labels. This makes the IUT input enabled in the quiescent state. If no match could be made, the

adapter will exit the `while`-loop and send `"fail"` over the standard output (line 14). This output should never occur as it would imply the IUT is not input enabled.

We implement the sentinel for the IUT in the adapter. Recall that we require single-threaded execution. We output `"sen"` over the standard output after initialisation of the IUT, and after each method call to the IUT. First, we elaborate on the `"sen"` output after initialisation. The run-to-completion semantics apply directly after the IUT is initialized, so it is possible for proactive events to occur directly after initialisation. On line 1 we create a new instance of the IUT. At this point, all proactive events that are allowed will also occur. After the IUT has completed all executions, line 2 outputs `"sen"` on the standard output stream. This sentinel output will not occur before the IUT has finished execution because we require the implementation to be single-threaded. The same applies to the sentinels on lines 9 and 13, that we output after their respective method calls.

```
1  iut := new IUT()
2  std.out << "!sen"
3  while (b):
4    b := false
5    std.in >> input
6    if input == "?re_event1":
7      b := true
8      iut.event1()
9      std.out << "!sen"
10   if input == "?re_event2":
11     b := true
12     iut.event2()
13     std.out << "!sen"
14 std.out << "!fail"
```

Listing 4.2: Pseudo-code for adapter

Proactive events are essentially method calls by the IUT on other interfaces. In OIL component specifications these are referred to as the required interfaces. In order for the JTorX to interpret these proactive events, the method calls made by the IUT must be translated to output for the standard output stream. This is also done in the adapter. We create a custom implementation of such a required interface. An example of such an implementation is given in Listing 4.3. Lines 2 and 4 show implementations of two methods on the required interface `Interface`. Each method outputs the corresponding proactive event on the standard output stream. By defining custom implementations of required interfaces in the adapter, JTorX is able to interpret proactive events caused by the IUT.

```
1 class myInterface implements Interface
2   void proEvent1():
3     std.out << "!pro_event1"
4   void proEvent2():
5     std.out << "!pro_event2"
```

Listing 4.3: Pseudo-code for implementing an interface

4.3 Experiments

In this section, we describe a series of experiments to test model-based testing in the context of OIL. With the exception of experiment E9 and E10, all experiments are performed on components specifically designed for each respective experiment. Experiment E1 (Section 4.3.1) illustrates the problem that can occur with asynchronous communication in the classic **ioco** relation by Tretmans [21]. In experiments E2 and E3 (Section 4.3.2, Section 4.3.3) we show how the addition of the sentinel is useful for accurately detecting the cause of a test failure. With experiments E4–E8 (Section 4.3.4–Section 4.3.8) we test the validity of our code generator. During these experiments, we assume that the implementation of the code generator is correct. Given that these experiments are performed correctly, an unexpected result in one of them contradicts the validity of the code generator. Finally, experiments E9 and E10 (Section 4.3.9, Section 4.3.10) show how we can apply model-based testing in a different context.

Quiescence is observed using a *timeout*. If the IUT does not give any output before the timeout ends, we assume the IUT is in a quiescent state. This is an approximation to quiescence. Recall that the definition of quiescence (Definition 4.1.5) requires that the IUT is quiescent iff no autonomous action is possible from the current state. Obviously, a finite timeout cannot observe true quiescence. If the IUT returns an output after the timeout ends, JTorX will have assumed quiescence, but this is clearly not the case.

JTorX also features a form of guided on-line testing which aims to achieve full coverage. Full coverage of a model means that every state and transition has been visited during the test. We refer to this test as a *coverage test*. We use it to test if we can reach full coverage of a model, and report how many steps it takes with the selected seed.

The *coverage tables* report statistics on coverage of the model during a test. Figure 4.14 is an example of such a coverage table. *#Nodes* and *#Edges* correspond to the number of states and transitions respectively. *#QLoops* and *#NQLoops* correspond to the number of quiescent and non-quiescent loops. *#QTrans* and *#NQTrans* correspond to the number of quiescent and non-quiescent transitions. Quiescent loops are not explicitly modelled, but they do occur during the test. Hence quiescent loops are found in the model of the IUT, this can result in a greater *#Edges* in the model for the IUT. The *mdl* denotes the model of the specification, the *sa* denotes the model expressing the behaviour of the IUT. The results reported next to the *mdl* and *sa* are the expected results derived from the given specification. The *sum* denotes the summation of results found in all test runs, and *mdl-i* and *sa-i* denote results of test *i*.

4.3.1 E1: Problems in asynchronous ioco

In this experiment, we will show the problem that occurs in classic **ioco** with asynchronous communication. Figure 4.8 shows an IOLTS that describes the problem. The environment can make a request to the IUT with the sequence `?a?b`. The IUT needs time to process the request, after which the IUT will take the τ-step and reply with `!d`. While the IUT is processing the request, the environment can also cancel the request using `?c`. In an asynchronous setting, the timing of the cancellation request cannot be guaranteed. The environment may send `?c` before the IUT has finished processing, but it may reach the IUT after processing is done and `!d` has been sent. This will result in the observation of illegal behaviour when the IUT was, in fact, behaving correctly.

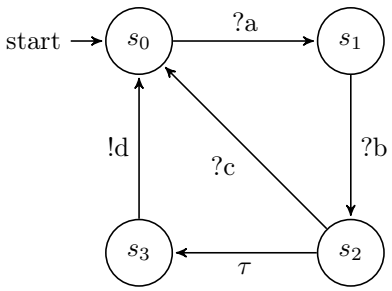


Figure 4.8: Problem case for **ioco** with asynchronous testing

For this experiment, we have created a custom C++ implementation. We insert a timeout of 300 ms after which `!d` will be sent. To simulate a delay in the communication channels, the implementation will not process new input before `!d` occurs. We expect the tester will now be able to call `?c` before the tester is able to receive `!d`.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	500 ms
Random seed:	941875214
Steps:	1000
#States LTS:	4
#Transitions LTS:	5
Coverage:	Full coverage
Verdict:	Failure
Coverage test:	–

As expected, the test fails. Before `!d` has occurred, the tester decides to send `?c` and then test for quiescence. Due to a delay in the communication channels, the IUT does not receive `?c` before it has already replied with `!d`. The tester will now receive `!d` when it expects quiescence, failing the test. The sequence diagram is shown in Figure 4.9.

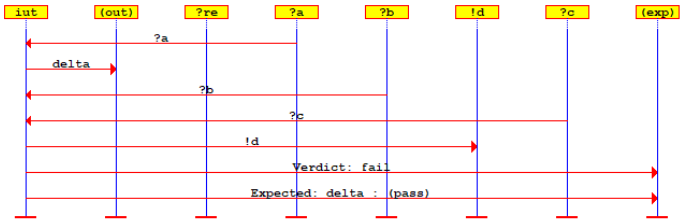


Figure 4.9: Sequence diagram from experiment E1

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	4	5	0	0	0	5	sa	4	8	3	0	0	5
sum	4	5	0	0	0	5	sum	4	8	3	0	0	5
mdl-1	4	5	0	0	0	5	sa-1	4	8	3	0	0	5

Figure 4.10: Coverage table for experiment E1

Noroozi refers to this as a race situation in **ioco** [18]. The problem occurs when there exists a state from which both input and output (via τ-steps) are possible. This is allowed in classic **ioco**. Noroozi solves this problem by introducing the **ioco**$^{\sqcap,\sqcap}$ relation, which requires both the specification and the IUT to be internal choice. See section Section 4.1.3 for more details on **ioco**$^{\sqcap,\sqcap}$, and Section 4.2.1 for how this applies to OIL.

Conclusion: Classic **ioco** does not work in a setting with asynchronous communication. Hence, we have to use the **ioco**$^{\sqcap,\sqcap}$ relation as proposed by Noroozi.

4.3.2 E2: Test failure detection without the sentinel

In this experiment, we will attempt to determine the exact event that causes test failure. We will show that JTorX is unable to detect this accurately if the specification includes a sequence that consists exclusively of reactive events. In experiment E3 (Section 4.3.3) we show how we have solved this problem by introducing the sentinel.

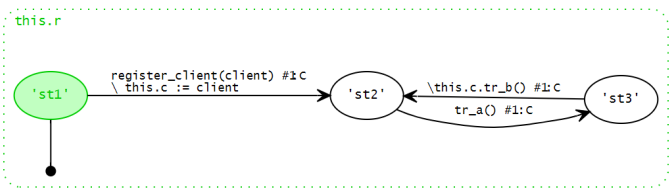


Figure 4.11: OIL specification modelling the IUT from experiment E2

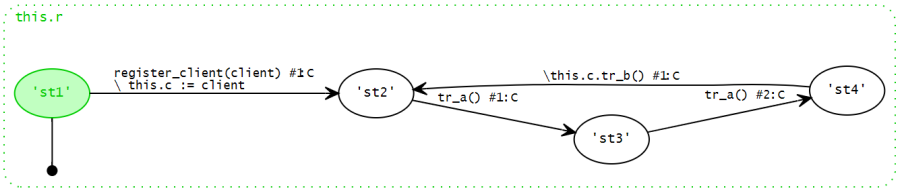


Figure 4.12: OIL specification modelling the desired behaviour from experiment E2

Figure 4.11 and Figure 4.12 show the OIL specifications we use for the IUT and the desired behaviour respectively. The IUT can infinitely repeat sequences of `tr_a()`, `tr_b()`. However, the desired behaviour specifies that sequences of `tr_a()`, `tr_a()`, `tr_b()` can be infinitely repeated. Clearly, the IUT will not conform to the desired behaviour. In order to emphasise the fact that the communication between JTorX and the IUT is asynchronous, we have implemented a delay in the IUT. The IUT waits for 1000 ms before sending the `tr_b()` event.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	1500 ms
Random seed:	2008614735
Steps:	1000
#States LTS:	4
#Transitions LTS:	4
Coverage:	Incomplete coverage, see Figure 4.14
Verdict:	Failure in 11 steps, see Figure 4.13
Coverage test:	–

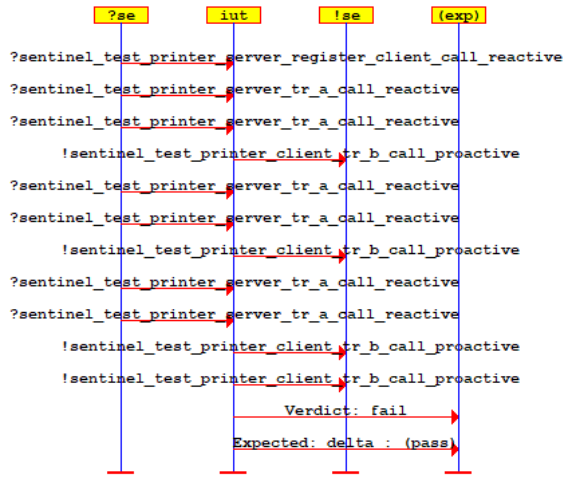


Figure 4.13: Sequence diagram from experiment E2

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	4	4	0	0	0	4	sa	4	7	3	0	0	4
sum	4	4	0	0	0	4	sum	4	4	0	0	0	4
mdl-1	4	4	0	0	0	4	sa-1	4	4	0	0	0	4

Figure 4.14: Coverage table from experiment E2

As expected, the test fails. Figure 4.13 shows the sequence diagram of the test. Non-conformance is detected after 11 steps. These results are misleading, as they seem to suggest the problem occurs after multiple successful iterations. The tester appears to successfully complete the `tr_a()`, `tr_a()`, `tr_b()` sequence at least two times. This is not the case. The real failure occurs on the

third step since the IUT returns `tr_b()` after every `tr_a()`.

Conclusion: In the context of random on-line testing with asynchronous communication, it can be difficult to determine the exact point of failure in a test.

4.3.3 E3: Test failure detection with the sentinel

In this experiment, we show that we can determine the exact event that caused failure within a sequence of reactive events by implementing the sentinel. See Section 4.2.3 for more details on the sentinel. We use the same specifications as the previous experiment (Section 4.3.2), but in this case, we implement the sentinel. The delay of 1000 ms before sending `tr_b()` is still present in the IUT.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	1500 ms
Random seed:	2008614735
Steps:	1000
#States LTS:	7
#Transitions LTS:	7
Coverage:	Incomplete coverage, see Figure 4.16
Verdict:	Failure in 5 steps, see Figure 4.15
Coverage test:	–

The test fails as expected. Figure 4.15 shows the sequence diagram of the test. Due to the implementation of the sentinel, the tester is forced to wait for the IUT to complete its execution. Instead of the expected `!sen`, the tester receives a `tr_b()` event. The test failure is now detected at the correct event in the sequence.

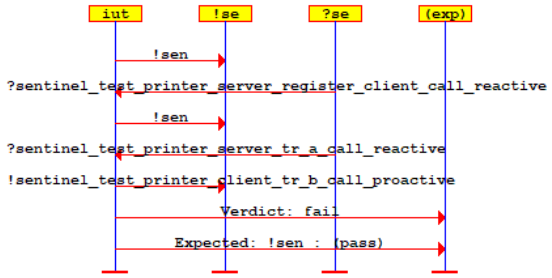


Figure 4.15: Sequence diagram from experiment E3

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	7	7	0	0	0	7	sa	7	10	3	0	0	7
sum	5	4	0	0	0	4	sum	5	4	0	0	0	4
mdl-1	5	4	0	0	0	4	sa-1	5	4	0	0	0	4

Figure 4.16: Coverage table from experiment E3

Conclusion: In the context of random on-line testing with asynchronous communication, the sentinel can help us to determine the exact point of failure in a test.

4.3.4 E4: Run-to-completion semantics in OIL

In this experiment, our goal is to test the run-to-completion semantics (Section 2.2.2) of OIL in the generated code. Test failure implies the implementation of the code generator is incorrect.

The run-to-completion semantics demand that proactive events have priority over reactive events. Figure 4.17 shows the OIL component specification we will be using for this experiment.

The IUT for this experiment is generated from the specification using the transformation implemented in Spoofax. Important are the two events that cause the component to transition from `st4` to `st2`. The event `tr_c()` is a reactive event, and `tr_d()` is a proactive event. After the component has transitioned to state `st4`, `tr_d()` should occur autonomously, and therefore `tr_c` should never happen.

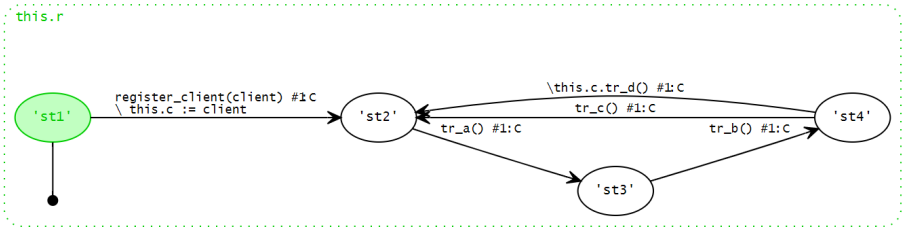


Figure 4.17: OIL specification for experiment E4

Section 4.1.4 explains the algorithm JTorX uses for random on-line testing. An important detail is that JTorX will always check the standard output stream for new output from the IUT before supplying new input. If output is found, this will have priority over new input. For the specification in Figure 4.17 this means that, if the IUT is sufficiently fast, JTorX finds `tr_d()` on the output stream, forcing it to perform this proactive event over `tr_c()`. In the case of this experiment, the IUT is significantly faster than JTorX.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	100 ms
Random seed:	941875214
Steps:	1000
#States LTS:	4
#Transitions LTS:	5
Coverage:	1 transition missing, see Figure 4.18
Verdict:	Success without full coverage
Coverage test:	No full coverage in 1000 steps

The coverage table in Figure 4.18 shows we are unable to reach one transition. The `tr_c()` has not occurred in the test due to `tr_d()` having priority. This shows the run-to-completion semantics function as intended.

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	4	5	0	0	0	5	sa	4	8	3	0	0	5
sum	4	4	0	0	0	4	sum	4	7	3	0	0	4
mdl-1	4	4	0	0	0	4	sa-1	4	7	3	0	0	4

Figure 4.18: Coverage table for experiment E4

Conclusion: The experiment is unable to find mistakes in the generated code.

4.3.5 E5: Concerns

In this experiment, our primary goal is to test the implementation of concerns. For more details on concerns in OIL, see Section 2.1.3. Other notable OIL constructs that appear in this test are scopes and guards. Figure 4.19 shows the OIL component specification used for this experiment. The IUT for this experiment is generated from the specification using the transformation implemented in Spoofax. Test failure implies the implementation of the code generator is incorrect.

The component specification in Figure 4.19 models a printer which we can turn on and off, register and unregister a client, add and remove jobs, and print an added job to the client. The printer also features a cool-down mechanism to prevent overheating. We can add a new job at most twice before the printer needs a moment to cool down. We have implemented this cool-down

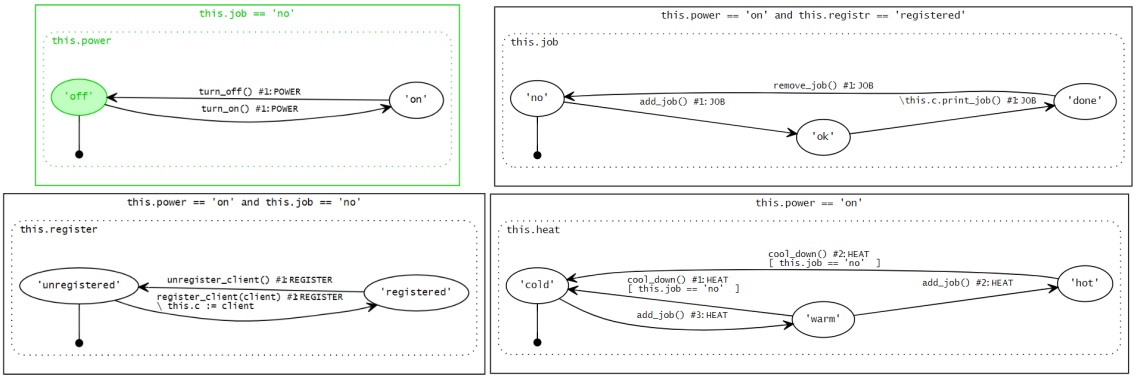


Figure 4.19: OIL specification for experiment E5

mechanism in OIL using concerns.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	100 ms
Random seed:	1680315128
Steps:	1000
#States LTS:	30
#Transitions LTS:	42
Coverage:	Full coverage, see Figure 4.20
Verdict:	Success
Coverage test:	Full coverage reached in 148 steps

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	30	42	0	0	0	42	sa	30	56	14	0	0	42
sum	30	42	0	0	0	42	sum	30	56	14	0	0	42
mdl-1	30	42	0	0	0	42	sa-1	30	56	14	0	0	42

Figure 4.20: Coverage table for experiment E5

The test succeeds and we obtain full coverage. We conclude that, given the correct input, the component behaves as expected.

Conclusion: The experiment is unable to find mistakes in the generated code.

4.3.6 E6: Concern failure

In this experiment, we test whether the IUT generated by the component specification from E5 (Figure 4.19) conforms to the component specification in Figure 4.21. Notice the difference between the two specifications. Figure 4.21 is identical to Figure 4.19 except for the concerns. In practice, this means the cool-down mechanism does not limit the behaviour in the specification from this experiment. However, the cool down mechanism should still be present in the IUT. As a result, the IUT must reject the input from JTorX if it is in conflict with the cooldown mechanism. Test success implies the implementation of the code generator is incorrect.

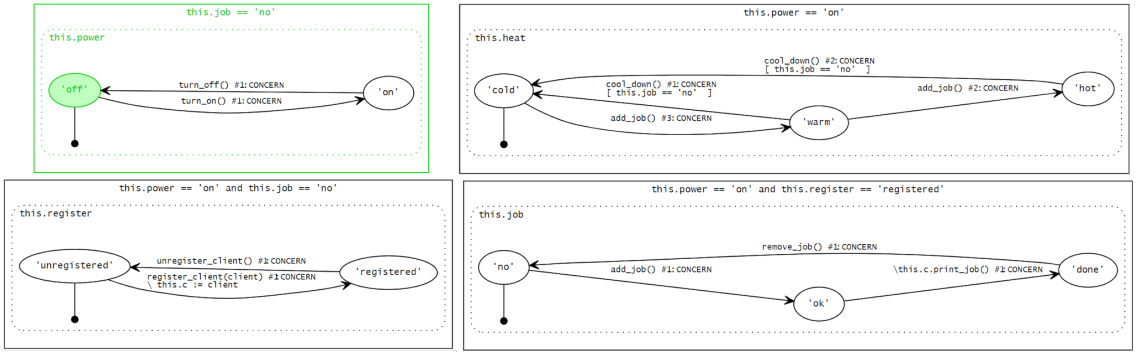


Figure 4.21: OIL specification for experiment 2 (*spec2*)

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	100 ms
Random seed:	1680315128
Steps:	1000
#States LTS:	30
#Transitions LTS:	46
Coverage:	Incomplete coverage, see Figure 4.23
Verdict:	Failure in 65 steps
Coverage test:	Failure in 13 steps

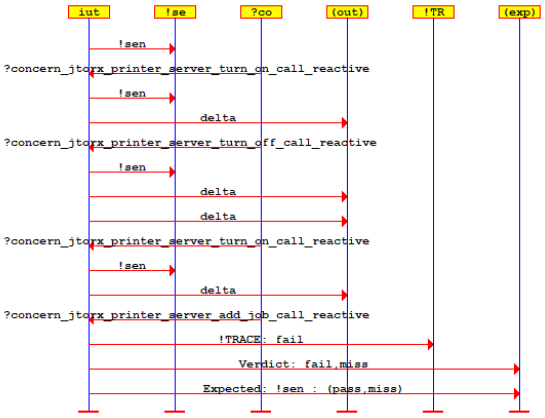


Figure 4.22: Sequence diagram from experiment E6

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	30	46	0	0	0	46	sa	30	60	14	0	0	46
sum	23	27	0	0	0	27	sum	23	35	8	0	0	27
mdl-1	23	27	0	0	0	27	sa-1	23	35	8	0	0	27

Figure 4.23: Coverage table for experiment E6

As expected, the test fails. The IUT does not conform to the behaviour described in Figure 4.21. The sequence diagram in Figure 4.22 gives a clearer indication were the test fails. The tester calls `add_job` event due to the availability of `add_job() #3` in the `heat` region. The only requirement this transition has is that `power` is turned on. However, in the IUT `add_job() #3` is part of concern `HEAT`. The result is that `add_job() #1` must also fire. The transition `add_job() #1` is not allowed while there is no registered client. Hence, the IUT returns `fail`.

Conclusion: The experiment is unable to find mistakes in the generated code.

4.3.7 E7: Guards, actions, and silent events

In this experiment, we test guards, actions, and silent events. The specification in this experiment is inspired by the component specification presented in Section 2.2.4. We use a simplified version due to limitations of the translation from OIL to mCRL2. The use of parameters is not yet supported. Figure 4.24 shows the OIL component specification used for this experiment. The IUT for this experiment is generated from the specification using the transformation implemented in Spoofax. Test failure implies the implementation of the code generator is incorrect.

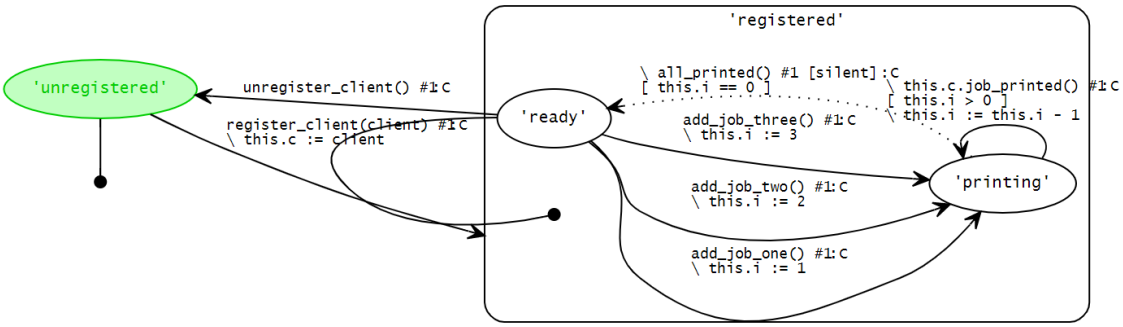


Figure 4.24: OIL specification for experiment E7

The component specification in Figure 4.24 models a printer which can register and unregister a client, and add a job which can be printed one, two, or three times. The job is then printed by the client. When the job is done printing, the printer will autonomously ready itself to accept a new job.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	100 ms
Random seed:	6768
Steps:	1000
#States LTS:	8
#Transitions LTS:	11
Coverage:	Full coverage, see Figure 4.25
Verdict:	Success
Coverage test:	Full coverage in 23 steps

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	8	11	0	0	0	11	sa	8	13	2	0	0	11
sum	8	10	0	0	0	10	sum	8	13	2	0	0	11
mdl-1	8	10	0	0	0	10	sa-1	8	13	2	0	0	11

Figure 4.25: Coverage table for experiment E7

The test succeeds. Notice in Figure 4.25 that we are missing one edge in the model. This is the τ-step. Since the τ-step is an internal action of the IUT, JTorX is unable to observe this transition. Hence we can still conclude full coverage of the IUT. We conclude that, given the correct input, the component behaves as expected.

Conclusion: The experiment is unable to find mistakes in the generated code.

4.3.8 E8: Action failure

In this experiment, we test whether the IUT generated by the component specification from E7 (Figure 4.24) conforms to the component specification in Figure 4.26. Notice the difference between the two specifications. Figure 4.26 is identical to Figure 4.19 except for the transition `add_job_three() #1`. Instead of setting `this.i` to 3, we set it to 4. As a result JTorX will expect the IUT to output the event `job_printed()` four times after sending the `add_job_three()` event.

Test success implies the implementation of the code generator is incorrect.

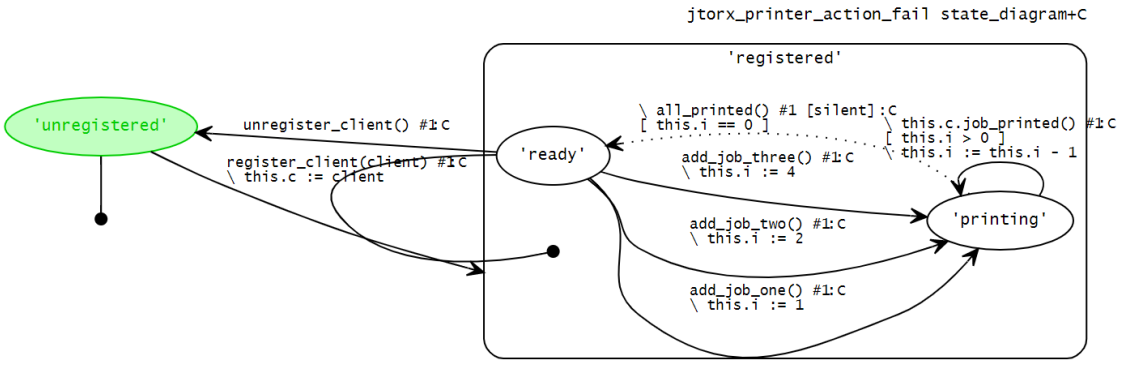


Figure 4.26: OIL specification for experiment E8

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	100 ms
Random seed:	6768
Steps:	1000
#States LTS:	9
#Transitions LTS:	12
Coverage:	Incomplete coverage, see Figure 4.28
Verdict:	Failure in 25 steps
Coverage test:	fail

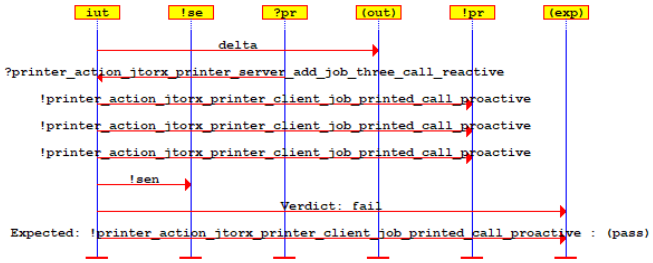


Figure 4.27: Sequence diagram from experiment E8

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	9	12	0	0	0	12	sa	9	14	2	0	0	12
sum	9	11	0	0	0	11	sum	9	13	1	0	0	12
mdl-1	9	11	0	0	0	11	sa-1	9	13	1	0	0	12

Figure 4.28: Coverage table for experiment E8

As expected, the test fails. The sequence diagram in Figure 4.27 shows what happened after JTorX gave the reactive event `add_job_three()` as input. The IUT outputs the proactive event `job_printed()` three times and then outputs the sentinel. Instead of the sentinel, JTorX expected a fourth `job_printed()`. Hence, the test fails.

Conclusion: The experiment is unable to find mistakes in the generated code.

4.3.9 E9: Boost case, part 1

In this experiment, we apply model-based testing in a different context. Océ currently implements many of its software components as state-machines using the Boost C++ libraries [1].

For this experiment, we are given an IUT implemented in Boost. The specification we will be using is an OIL component specification that should describe the desired behaviour of the IUT in Boost. We will use model-based testing to test if the behaviour of the Boost implementation conforms to the behaviour described in the OIL component specification. Test failure implies the OIL component specification does not capture the behaviour of the Boost implementation.

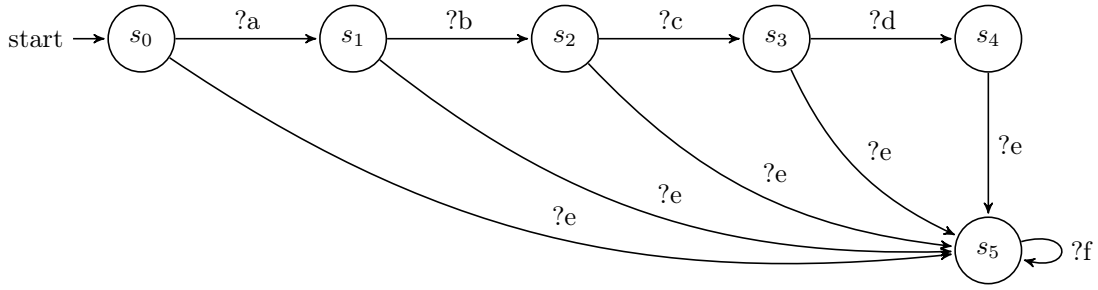


Figure 4.29: Abstract visualisation of the boost model used for experiment E9

We will not go into details on the models of the implementation or the specification. These are confidential. The IOLTS depicted in Figure 4.29 gives an impression of the shape of the specification for this experiment. In the real specification, outputs exist between every two inputs. These have been left out to keep the visualisation in Figure 4.29 simple. Notice that state s_5 in the model is *sink* state, i.e., once the IUT enters this state, it is unable to exit this state. Moreover, the IUT is unable to return to its initial state during normal execution. Eventually, all sequences will end up in the sink state s_5.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	500 ms
Random seed:	456
Steps:	1000
#States LTS:	83
#Transitions LTS:	121
Coverage:	Incomplete coverage, see Figure 4.31
Verdict:	Failure in 10 steps
Coverage test:	–

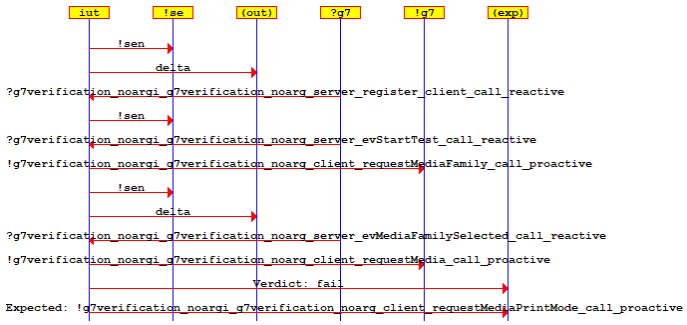


Figure 4.30: Sequence diagram from experiment E9

Our first attempt at this experiment ends in a test failure. The sequence diagram and coverage table for this run is shown in Figure 4.30 and Figure 4.31. The test failed after a very small number of steps. Inspection of the sequence diagram reveals the event that leads to the failure of the test. The IUT responded with the wrong proactive event. Further inspection of the IUT and the specification revealed that we were in fact using a different implementation as IUT than originally intended. This mix up was most likely caused by the strong resemblance between the two implementations.

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	83	121	0	0	0	121	sa	83	148	27	0	0	121
sum	8	7	0	0	0	7	sum	8	9	2	0	0	7
mdl-1	8	7	0	0	0	7	sa-1	8	9	2	0	0	7

Figure 4.31: Coverage table for experiment E9

Conclusion: The test failed because the wrong OIL specification was used in the test.

4.3.10 E10: Boost case, part 2

This experiment has the same setup as E9 (Section 4.3.9), except that we are using the correct implementation as the IUT. Test failure implies the OIL component specification does not capture the behaviour of the Boost implementation.

Experiment details:

Test tool:	JTorX
Test type:	random on-line testing
Timeout:	500 ms
Random seed:	456
Steps:	1000
#States LTS:	83
#Transitions LTS:	121
Coverage:	Incomplete coverage, see Figure 4.32 and Figure 4.33
Verdict:	Success
Coverage test:	–

JTorX concludes the test is successful. However, inspection of the coverage table in Figure 4.32 reveals the limitations of testing in JTorX. JTorX has taken 1000 steps during this test. However, the vast majority of these steps were taken in the sink state (state s_5 in Figure 4.29). The run ends in the sink state, where no further coverage can be obtained.

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	83	121	0	0	0	121	sa	83	148	27	0	0	121
sum	12	13	0	0	0	13	sum	12	16	3	0	0	13
mdl-1	12	13	0	0	0	13	sa-1	12	16	3	0	0	13

Figure 4.32: Coverage table for experiment E10, without use of coverage

We attempt to remedy this problem by running consecutive coverage runs. Each consecutive run will take advantage of the coverage information acquired during the previous run. Figure 4.33 shows the coverage table from the coverage runs. On the first run, the run ends in 21 steps with a pass. Repeating the coverage run yields some additional coverage, but clearly not enough for full coverage. After several coverage runs on the same random seed, JTorX seems unable to obtain additional coverage.

run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans	run#	#Nodes	#Edges	#QLoops	#NQLoops	#QTrans	#NQTrans
mdl	83	121	0	0	0	121	sa	83	148	27	0	0	121
sum	18	21	0	0	0	21	sum	18	26	5	0	0	21
mdl-1	15	16	0	0	0	16	sa-1	15	19	3	0	0	16
mdl-1	2	3	0	0	0	3	sa-1	2	4	1	0	0	3
mdl-1	1	1	0	0	0	1	sa-1	1	1	0	0	0	1
mdl-1	0	1	0	0	0	1	sa-1	0	2	1	0	0	1
mdl-1	0	0	0	0	0	0	sa-1	0	0	0	0	0	0
mdl-1	0	0	0	0	0	0	sa-1	0	0	0	0	0	0
mdl-1	0	0	0	0	0	0	sa-1	0	0	0	0	0	0
mdl-1	0	0	0	0	0	0	sa-1	0	0	0	0	0	0
mdl-1	0	0	0	0	0	0	sa-1	0	0	0	0	0	0

Figure 4.33: Coverage table for experiment E10, with use of coverage

Although the experiment did not yield inconsistencies between the specification in OIL and the implementation in Boost, we remain unconvinced due to the poor coverage of the test. It is clear the existence of a sink state is a problem for the current test setup.

Conclusion: The experiment is unable to find inconsistencies between the OIL component specification and the Boost implementation.

4.4 Limitations

Our goal in this chapter is to validate the correctness of the transformation presented in Chapter 3. There are notable limitations to the use of model-based testing in this context.

The set of OIL component specification we tested is not exhaustive, i.e., we have not tested the generated code of every possible specification in OIL. This is impossible as there exists an infinite number of specifications. The specifications for our test cases were designed to test specific constructs in the OIL language. Certain combinations of OIL constructs may lead to incorrect transformations.

Moreover, the sequences tested during our experiments were not exhaustive. We cannot exclude the possibility that a test failure will occur beyond the number of steps tested in our experiments. This is a problem inherent to model-based testing, as exhaustively testing every possible sequence is generally not feasible.

4.5 Discussion

In this section, we discuss some of our observations, and why certain decisions were made during the process of implementing model-based testing in the context of OIL.

4.5.1 Partial specifications and sink states

The main reason we only test good weather behaviour is because bad weather behaviour, i.e. illegal events, lead to the failure state. Since no recovery is possible from the failure state, this is effectively a sink state. As we showed in the experiments with the Boost case (Section 4.3.10), sink states form a problem for random on-line testing in JTorX. As JTorX is unable to infinitely loop through a model featuring a sink state, obtaining full coverage becomes difficult. The coverage run, a guided online test in JTorX that makes use of previously obtained coverage information, is also unable to obtain full coverage.

A possible solution to this problem is to add an additional event that can return the specification and the IUT from the sink state back to the initial state. The event can easily be added to the LTS alongside the sentinel. In the IUT this reset event can be implemented in the adapter by simply destructing the current instance of the component and creating a new instance.

The solution proposed above is effective in case the sink is a single state. However, in the more general case where the occurrence of one event excludes other events from occurring in the rest of the sequence, this solution falls short. In the latter case it is unclear from which state the reset event should be allowed. This is why we argue that this should be resolved at the level of the testing tool, and not by making adjustments to the model.

4.5.2 Purpose of the sentinel

Originally, the sentinel was intended to replace the timeout as indication of quiescence in JTorX. By removing the need for a timeout we can significantly speed up testing.

Theoretically, quiescence means no output will be given autonomously from this state, ever. The timeout as indication of quiescence is an approximation of this phenomenon. Due to our knowledge of the IUT, we can claim that the implementation is quiescent after the sentinel is given as output. Unfortunately, JTorX does not allow us to specify an output as quiescent.

We decided to include the sentinel in our implementation because of its value during testing. When applying random on-line testing to an implementation that features a sequence of reactive events, the sentinel allows us to determine the exact event in the sequence that caused the test case to fail, as shown in the experiments in Section 4.3.2 and Section 4.3.3.

4.5.3 The value of model coverage

Model coverage is considered a heuristic, and it is difficult to determine the value of full coverage in a test case. Although all transitions have been used at least once, and all states have been visited at least once, full coverage does not imply all possible sequences have been tested. It may also be the case an implementation shows unintended behaviour after a certain number of loops.

We argue that model coverage has significant value in our context. The main reason is that the OIL specifications describe the workings of the IUT in great detail. Although we did not test for code coverage in our experiments, we suspect full coverage of the model will result in a high code coverage. We leave these experiments for future work.

Chapter 5

Conclusions and future work

In this thesis, we have shown how we can generate an implementation from a component specification in OIL, and validate the correctness by means of model-based testing. The research in this thesis was guided by three research questions.

Our first research question asks how we can implement a transformation from an OIL component specification to C++ executable code (**RQ1**). We started with an analysis of the semantics of OIL. Based on this analysis, we have created a pseudo-code that expresses these semantics. The pseudo-code uses the basic concepts from object-oriented general-purpose languages. As such, the pseudo-code serves as a blueprint for expressing the semantics of OIL in an object-oriented general-purpose language.

We have implemented the C++ code generator for component specification in OIL with the Spoofax language workbench. Defining a single transformation from OIL to C++ is difficult to implement due to its high complexity. Hence, we have made use of the existing transformation in the OIL architecture that was already implemented in Spoofax. The architecture contained several transformations that could be of use to our code generator. Among them is a transformation from the OIL language to the Desugared AST, an intermediate representation of the OIL language in the form of an abstract syntax tree. The Desugared AST is a representation of the OIL language that does not include syntactic sugar, making it an ideal starting point for our code generator. We were now able to reuse the transformations to the Desugared AST by composing into our own transformation.

From the Desugared AST, we have introduced another two intermediate representations before the transformation to a textual representation of the C++ code. The first is the GPL AST. The transformation from the Desugared AST to the GPL AST is a restructuring of OIL terms, to create a data-structure that is suitable for generating object-oriented general-purpose languages. Next is the CPP AST, which is an AST representation of the generated C++ code. The CPP AST represents only a small subset of the C++ language, and it is domain-specific for OIL at Océ. From the CPP AST, we have defined the final transformation to C++ executable code.

We found that the transformation from the Desugared AST to the GPL AST was by far the most challenging to define. In an attempt to find the cause, we have analysed the transformation patterns that occur in the three transformations defined by us. We concluded that in our case the scope has the greatest effect on the complexity of a transformation. Especially the global source to global target transformations were challenging to define, which frequently occur in the transformation from the Desugared AST to the GPL AST. We managed to alleviate some of this complexity by composing global to global transformation from smaller, less complex transformations of which either the target or the source were local.

Our second research question asks how we can generalize this code generator to other object-oriented general-purpose languages like C# and Java (**RQ2**). Although we have not created code generators for languages other than C++, we have made considerable progress towards this goal. The semantics expressed in the pseudo-code can be translated to an object-oriented GPL. Moreover, the GPL AST is not language-specific to C++. It contains configuration data that can be used for generating code in any object-oriented general-purpose language.

Our third research question asks how we can validate the correctness of the code generator using model-based testing (**RQ3**). In order to answer this question, we have shown how we can use model-based testing to validate the correctness of the generated implementations with respect

to the original OIL component specification. If we can find an implementation that does not conform to its respective specification, we have proven the code generator to be incorrect.

Our implementation of model-based testing with JTorX communicates with the implementation over the standard input-output steams. Communication in this setting is asynchronous. In order to use model-based testing in an asynchronous setting, we must use a specific variant of the input-output conformance relation, namely $\mathbf{ioco}^{\sqcap,\sqcap}$. In order to use the $\mathbf{ioco}^{\sqcap,\sqcap}$ relation, we have shown that OIL component specification can be modelled as an $\text{IOLTS}^{\sqcap}$, and that the respective implementations can be modelled as an $\text{IOTS}^{\sqcap}$. This allows us to test for input-output conformance in a setting with asynchronous communication.

We introduced the sentinel to assist in debugging. The sentinel allows us to determine the exact input that leads to test failure. The sentinel indicates that the implementation is ready to process new input. In the context of OIL component specifications, the sentinel indicates that the implementation has finished processing the previous input, and will not perform any further action autonomously. It will not take internal actions, nor give any further output.

In order to prove the code generator incorrect, we have performed a number of experiments using random on-line testing with JTorX. The experiments consist of checking whether the $\mathbf{ioco}^{\sqcap,\sqcap}$ relation holds between a specification in OIL and the implementation generated with our code generator. During the experiments we made use of the model coverage heuristic. This heuristic was useful in our context since the OIL specification describe the behaviour of the implementation in great detail. None of the experiments were able to prove the code generator to be incorrect.

There are clear limitations to validating the correctness of the code generator using our approach. It is important to stress the fact that the code generator has not been formally verified. There may exist OIL specifications that generate incorrect implementations. Although our approach does not ensure the correctness of every possible implementation, it does provide a correctness test for every generated implementation. Moreover, these test cases are algorithmically derived from formal models on which requirements can be formally verified, and the implementation and execution of these test cases can be almost fully automated. By applying model-based testing to every implementation generated from the code generator, we can make a strong argument that these implementations are correct. And with each new implementation generated and tested, the claim that our transformation is correct also becomes stronger.

5.1 Future work

The transformation presented in this thesis does not support the full OIL language yet. The most notable omissions are the more complex data-types such as mappings. New constructs may be added in the future as the OIL language is still in development.

Another omission in the current code generator are the adapters for JTorX. The adapters used during our research were created manually. However, creating a code generator for an adapter is straightforward. This would be a valuable addition to the code generator.

Inspired by the commonalities found in the transformation to C++ and the transformation to mCRL2, Bunte created the Semantic AST. In the transformation pipeline, the Semantic AST is placed after the Desugared AST. We can use the Semantic AST as an intermediate step between the Desugared AST and the GPL AST. We suspect this will alleviate a significant part of the complexity of the current transformation.

JTorX is a simple and straightforward tool that is easy to use and quick to implement, but its features are somewhat limited. For example, the presence of sink states in a model presents a problem. The tester works best when every path that can be taken through the model leads back to the initial state. This requirement severely limits its applicability in an industrial setting. Moreover, support for JTorX is limited. The tool has not been updated since 2014. We are interested in a comparison between different tools that implement model-based testing. There are also two specific features we are interested in. First is the ability to set a specific output as indication of quiescence, instead of the timeout. In our context, this would be the sentinel output. Another is a guided on-line testing algorithm that makes smarter use of the coverage statistics. This algorithm in JTorX is limited in use, as we have shown in our experiments on the Boost implementation.

Océ is a company that already applies model-driven engineering in its production process. This means that Océ has made an effort to also create formal models that describe the behaviour of their software components. It would be valuable for the company to use model-based testing to

check whether these existing components indeed conform to the desired behaviour described in the corresponding models.

Bibliography

[1] Boost C++ libraries. https://www.boost.org/. [Online; accessed 17-Nov-2019].

[2] Hamid Reza Asaadi, Ramtin Khosravi, Mohammad Reza Mousavi, and Neda Noroozi. Towards model-based testing of electronic funds transfer systems. In *Fundamentals of Software Engineering - 4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011, Revised Selected Papers*, pages 253–267, 2011.

[3] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 266–270, 2010.

[4] Axel Belinfante. JTorX: exploring model-based testing. PhD Thesis. University of Twente. 2014.

[5] Mark Bouwman, Bob Janssen, and Bas Luttik. Formal modelling and verification of an interlocking using mCRL2. In *Formal Methods for Industrial Critical Systems - 24th International Conference, FMICS 2019, Amsterdam, The Netherlands, August 30-31, 2019, Proceedings*, pages 22–39, 2019.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.

[7] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, pages 21–39, 2019.

[8] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[9] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. WebDSL: a domain-specific language for dynamic web applications. In *OOPSLA Companion*, pages 779–780. ACM, 2008.

[10] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.

[11] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software and System Modeling*, 9(3):375–402, 2010.

[12] Zef Hemel and Eelco Visser. PIL: A platform independent language for retargetable DSLs. In *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, pages 224–243, 2009.

[13] Yi-Ling Hwong, Jeroen J. A. Keiren, Vincent J. J. Kusters, Sander J. J. Leemans, and Tim A. C. Willemse. Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider. *Sci. Comput. Program.*, 78(12):2435–2452, 2013.

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. The evolution of lua. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–26, 2007.

[15] Lennart C. L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *OOPSLA*, pages 444–463. ACM, 2010.

[16] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Bootstrapping domain-specific meta-languages in language workbenches. In *GPCE*, pages 47–58. ACM, 2016.

[17] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *SLE*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012.

[18] Neda Noroozi. Improving input-output conformance testing theories. PhD thesis. Eindhoven University of Technology. 2014.

[19] Neda Noroozi, Ramtin Khosravi, Mohammad Reza Mousavi, and Tim A. C. Willemse. Synchrony and asynchrony in conformance testing. *Software and Systems Modeling*, 14(1):149–172, 2015.

[20] Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

[21] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[22] Rutger van Beusekom, Jan Friso Groote, Paul F. Hoogendijk, Robert Howe, Wieger Wesselink, Rob Wieringa, and Tim A. C. Willemse. Formalising the Dezyne modelling language in mCRL2. In *FMICS-AVoCS*, volume 10471 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2017.

[23] Mark van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Softw., Pract. Exper.*, 30(3):259–291, 2000.

[24] Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: a component-based language development environment. *Electr. Notes Theor. Comput. Sci.*, 44(2):3–8, 2001.

[25] Jonne van Wijngaarden and Eelco Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. 2003.

[26] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer, 2007.

Appendix A

Implementation

A.1 C++ analysis

Enums

Element type:	C++ enum
Element name:	[enum name]
Occurrence:	Once per enum.
Purpose:	Define required enums.
OIL elements:	Enum name and enum values.

Connect

Element type:	C++ method
Element name:	`Connect`
Occurrence:	Once.
Purpose:	Returns a new instance of the component.
OIL elements:	–

State

Element type:	C++ struct
Element name:	`State`
Occurrence:	Once.
Purpose:	Object that stores all global state variables.
OIL elements:	Global state variables.

Area condition

Element type:	C++ method
Element name:	`area_condition_`[area name]
Occurrence:	Once per area.
Purpose:	Checks if the conditions for a specific area hold. This includes checking if the conditions for the parent area(s) hold.
OIL elements:	Parent area(s), (in case of area type state: corresponding global variable, state value), (in case of area type scope: invariant).

Area update

Element type:	C++ method
Element name:	`area_update_`[area name]
Occurrence:	Once per area.
Purpose:	Updates the global state based on the specific area. This includes updating the parent area(s).
OIL elements:	Parent area(s), (in case of area type state: corresponding global variable, state value).

Transition precondition

Element type:	C++ method
Element name:	`oil_transition_condition_`[event]`_`[transtion nr]
Parameters:	self, OIL parameters
Occurrence:	Once per transtion.
Purpose:	Checks if the transition is allowed to fire based on the pre-conditions.
OIL elements:	Message, Transition number, Pre-conditions, Parameters.

Process

Element type:	C++ method
Element name:	`oil_process_`[event]
Parameters:	self, OIL parameters
Occurrence:	Once per reactive event.
Purpose:	Performs all actions related to a call. For each transition related to the event: - checks transition condition, - sets the temporary variables based on the actions of the transition, - performs the state updates based on the actions of the transition, - checks the post-conditions of the transition.
OIL elements:	Message, list of transition numbers of all transitions with this Message, Parameters, actions per transition, post-conditions per transition.

Try transition

Element type:	C++ method
Element name:	`oil_try_transition_`[event]`_`[transtion nr]
Parameters:	self, OIL arguments
Occurrence:	Once per proactive or silent transition.
Purpose:	Proactive and silent events fire when the preconditions hold. These events do not require calls on the component, they are fired automatically when the preconditions hold. - checks if the preconditions hold, - sets correct arguments if the source area is valid, - checks transition condition for the transition, - perform process call if transition condition holds.
OIL elements:	Message, Transition number, Arguments.

Run

Element type:	C++ method
Element name:	`oil_run`
Parameters:	–
Occurrence:	Once.
Purpose:	Run is a loop that fires proactive events if their preconditions hold. The run loops stops if no more proactive events can be fired.
OIL elements:	List of proactive and silent events.

Handle call

Element type:	C++ method
Element name:	`oil_handle_call_`[event]
Parameters:	self, OIL parameters
Occurrence:	Once per reactive event.
Purpose:	Runs the related process call and subsequently activates the Run loop.
OIL elements:	Message, Parameters.

Public method

Element type:	C++ method
Element name:	[method name]
Parameters:	self, OIL parameters
Occurrence:	Once per reactive event.
Purpose:	Public method that runs the related handle call.
OIL elements:	Message, Parameters.

New

Element type:	C++ constructor/static method
Element name:	`New`
Parameters:	–
Occurrence:	Once.
Purpose:	Set of methods that handle the construction of an instance of the component. Important are the creation of the global state and the active object pointer.
OIL elements:	interface name.

A.2 Desugared AST

```
signature
  // Lexicals in UPPERCASE
  // Constructors Capitalized
  constructors
    // OIL module
    DESOILModule : List(Dependency) * List(DESTypeDef) * List(DESVariable)
      * List(DESArea) * List(DESTransition) -> DESOILModule

    // Dependencies
    DESProvides : MODULE * INTERFACE -> Dependency
    DESRequires : MODULE * INTERFACE -> Dependency

    // TypeDefs
    DESStateEnumDef : STATE * List(Exp) -> DESTypeDef

    // Variables
    DESVariable : VARIABLE * Type * Exp -> DESVariable

    // Areas
    DESRegion : REGION * Exp * List(SUPER) * List(DESArea) -> DESArea
    DESState  : STATE * Exp * List(SUPER) * List(DESArea) -> DESArea
    DESScope  : SCOPE * Exp * List(SUPER) * List(DESArea) -> DESArea

    // Transitions
    DESTransition : DESMessage * SOURCE * TARGET * List(CONCERNS) * Option(Exp)
      * Exp * Option(Exp) -> DESTransition

    // Message (or Event)
    DESMessage : Cause * EventTypePrimitive * MODULE * INTERFACE
      * DESMessageMethod -> DESMessage

    // Method
    DESMessageMethod : METHOD * List(DESParameter) * Option(Exp)
      -> DESMessageMethod

    // Paramenters
    DESIn  : PARAMETER * Option(Exp) -> DESParameter
    DESOut : PARAMETER * Option(Exp) -> DESParameter
```

Listing A.1: Signature for Desugared AST

A.3 GPL AST

```
signature
  // Lexicals in UPPERCASE
  // Constructors Capatalized
  constructors
    // OIL specification
    GPLSpec : NAME * MODUL * List(GPLProvides) * List(GPLRequires)
      * List(GPLInterface) * List(GPLEnum) * GPLGlobalState
      * List(GPLAreaCondition) * List(GPLAreaUpdate)
      * List(GPLTransitionCondition) * List(GPLConcernCondition)
      * List(GPLProcess) * List(GPLTryTransition) * GPLOilRun
      * List(GPLHandleCall) * List(GPLMethod) -> GPLSpec

    // Dependencies
    GPLRequires : MODUL * INTERFACE-NAME -> GPLRequires
    GPLProvides : MODUL * INTERFACE-NAME -> GPLProvides

    // Interfaces
    GPLInterface : INTERFACE-NAME * List(GPLInterfaceMethod) -> GPLInterface
    GPLInterfaceMethod : METHOD * List(GPLInterfaceArgument)
      -> GPLInterfaceMethod
    GPLInterfaceArgument : VAR-NAME * Type -> GPLInterfaceArgument

    // Enums
    GPLEnum : VAR-NAME * List(VAR-VALUE) -> GPLEnum

    // Global State
    GPLGlobalState : List(GPLVariable) -> GPLGlobalState
    GPLVariable : VAR-NAME * Type * Exp -> GPLVariable

    // Area conditions
    GPLRegionCondition : AREA-NAME * AREA-PARENT * List(AREA-PARENT)
      -> GPLAreaCondition
    GPLStateCondition : AREA-NAME * AREA-PARENT * VAR-NAME * VAR-VALUE
      * List(AREA-PARENT) -> GPLAreaCondition
    GPLScopeCondition : AREA-NAME * AREA-PARENT * Exp * List(AREA-PARENT)
      -> GPLAreaCondition

    // Area updates
    GPLRegionUpdate : AREA-NAME * AREA-PARENT * List(AREA-PARENT)
      -> GPLAreaUpdate
    GPLStateUpdate : AREA-NAME * AREA-PARENT * VAR-NAME * VAR-VALUE
      * List(AREA-PARENT) -> GPLAreaUpdate
    GPLScopeUpdate : AREA-NAME * AREA-PARENT * Exp * List(AREA-PARENT)
      -> GPLAreaUpdate

    // Transition conditions
    GPLTransitionCondition : GPLTransitionID * GPLMessageInfo
      * List(GPLParameter) * List(Exp) -> GPLTransitionCondition

    // Concerns
    GPLConcernCondition : GPLTransitionID * GPLMessageInfo
      * List(GPLParameter) * List(GPLProcessConcern) -> GPLConcernCondition

    // Handle method call
    GPLProcess : GPLID * GPLMessageInfo * List(GPLParameter)
      * List(GPLProcessPreCondition) * List(GPLProcessConcern)
```

```
      * List(GPLProcessTempsDecl) * List(GPLProcessUpdate)
      * List(GPLProcessPostCondition) -> GPLProcess

    GPLParameterIn : VAR-NAME * Type * Option(Exp) -> GPLParameter
    GPLProcessPreCondition : GPLTransitionID * List(GPLParameter)
      * DEVTransitionRef * DEVConcernRef -> GPLProcessPreCondition
    GPLProcessTransition : GPLTransitionID -> GPLProcessConcern
    GPLProcessConcern : CONCERN * List(GPLTransitionID) -> GPLProcessConcern
    GPLProcessTempsDecl : VAR-NAME * Type -> GPLProcessTempsDecl
    GPLProcessUpdate : GPLTransitionID * TARGET * Option(Exp)
      * List(CONCERN) -> GPLProcessUpdate
    GPLProcessPostCondition : GPLTransitionID * TARGET * Option(Exp)
      * List(CONCERN) -> GPLProcessPostCondition

    // Try proactive transitions
    GPLTryTransition : GPLTransitionID * GPLMessageInfo * List(GPLParameter)
      * Option(Exp) * Exp * METHOD -> GPLTryTransition

    // Run-to-completion loop
    GPLOilRun : List(GPLOilRunTransition) -> GPLOilRun
    GPLOilRunTransition : GPLTransitionID * GPLMessageInfo
      -> GPLOilRunTransition

    // Handle call
    GPLHandleCall : METHOD * GPLMessageInfo * List(GPLParameter)
      -> GPLHandleCall

    // Public method
    GPLMethod : METHOD * GPLMessageInfo * List(GPLParameter) -> GPLMethod

    // Logging
    GPLPrint : TEXT -> GPLPrint

    // Event id
    GPLMessageInfo : GPLID * GPLID * GPLID -> GPLMessageInfo

    // Transition id
    GPLTransitionID : METHOD * GPLINT -> GPLTransitionID

    // References
    GPLAreaRef : REFERENCE -> Exp
    GPLConcernRef : GPLTransitionID * GPLMessageInfo * List(GPLParameter)
      -> Exp
    GPLTransitionRef : GPLTransitionID * GPLMessageInfo * List(GPLParameter)
      -> Exp
```

Listing A.2: Signature for GPL AST

A.4 CPP AST

```
signature
  // Lexicals in UPPERCASE
  // Constructors Capitalized
  constructors
    CPPSpec           : List(CPPStat) -> CPPSpec
    CPPDeclClass      : ID * List(CPPExtends) * List(CPPStat) -> CPPStat
    CPPDeclMethod     : CPPAccess * ID * List(CPPArg) * CPPType * List(CPPStat)
                        -> CPPStat
    CPPDeclENUM       : ID * List(CPPID) -> CPPStat
    CPPDeclObject     : ID * List(CPPArg) * List(CPPStat) -> CPPStat
    CPPDeclVar        : ID * CPPType -> CPPStat
    CPPDeclArgument   : ID * CPPType -> CPPArg
    CPPExtendClass    : CPPAccess * CPPType -> CPPExtends
    CPPAssignVar      : CPPExp * CPPExp -> CPPStat
    CPPReturn         : CPPExp -> CPPStat
    CPPStatement      : CPPExp -> CPPStat
    CPPIf             : CPPExp * List(CPPStat) -> CPPStat
    CPPIfElse         : CPPExp * List(CPPStat) * List(CPPStat) -> CPPStat
    CPPWhile          : CPPExp * List(CPPStat) -> CPPStat
    CPPPrint          : STRING -> CPPStat
    CPPPrintVar       : STRING * List(CPPExp) -> CPPStat
    CPPAssert         : List(CPPExp) * STRING -> CPPStat
    CPPOilThread      : List(CPPArg) * CPPStat -> CPPStat
    CPPAccessPublic   : CPPAccess
    CPPAccessPrivate  : CPPAccess
```

Listing A.3: Signature for CPP AST