Optimizing the code generator for OIL

Canon Production Printing B.V.

Tom Buskens | 1378120 | t.h.buskens@student.tue.nl

September 14, 2021
V1.7

Supervisors: Olav Bunte (TU/e), Tim A. C. Willemse (TU/e) and
Louis van Gool (Canon Production Printing B.V.)

Abstract

OIL, short for Open Interaction Language, is a domain-specific language developed by Canon Production Printing B.V. It is a language that can be used for specifying, analyzing, and implementing models of system behavior. The tooling created for OIL can generate C++ code from OIL specifications. Part of this generated code is a scheduler that schedules so-called proactive events. The focus of this project is to improve this scheduler; the aim is to reduce the number of computations needed to do the scheduling. We investigate basic scheduling improvement strategies that do not need the collection of additional information. We also investigate scheduling strategies for which causal relations have to be gathered from the OIL specifications. These scheduling strategies could be used to skip the scheduling of events that are not needed. Another strategy that we investigate alters the OIL specifications themselves to make them easier to schedule. In this report, these strategies for improving the scheduler are discussed, verified, and achieved results are listed.

The writer was enabled by Canon Production Printing B.V. to perform research that partly forms the basis for this report. Canon Production Printing B.V. does not accept responsibility for the accuracy of the data, opinions, and conclusions mentioned in this report, which are entirely for the account of the writer.

Contents

1 Introduction **6**

2 Context **7**
2.1 What is OIL . . . 7
2.2 Formal definitions . . . 12
2.3 Code generation . . . 15

3 Problem description **19**

4 Basic scheduling improvement strategies **21**

5 Causal relations **24**
5.1 Scheduler framework . . . 24
5.2 Approximated causal relations . . . 27
5.3 Syntax analysis on implied transitions . . . 28
5.4 SMT solving on implied transitions . . . 32
5.5 Multiple concerns . . . 38
5.6 Unstable areas . . . 41
5.7 Combining gathering approaches . . . 43

6 Scheduling improvement strategies with gathered causal relations **44**
6.1 Improved scheduling complementing naive OIL run loop scheduler . . . 44
6.2 Removing events OIL run loop scheduler . . . 47
6.3 Recursive OIL scheduler . . . 49
6.4 Prioritizing events . . . 51

7 Altering OIL specifications **54**
7.1 Mealy machine . . . 54
7.2 LTS . . . 55

8 Verification **58**
8.1 JTorX . . . 58

9 Results **60**
9.1 Selected test traces . . . 60
9.2 Different gathering approaches and scheduling strategies . . . 61
9.3 Different prioritization . . . 66
9.4 Different compiler optimizations . . . 67
9.5 Overall remarks . . . 67

10 Conclusions and future work **68**
10.1 Conclusions . . . 68
10.2 Future work . . . 69

A Appendix **72**
A.1 Test case results . . . 72
A.2 Charts comparing execution time and amount tries . . . 78
A.3 Proofs . . . 79
A.4 Test specification visualizations . . . 90

List of Figures

1 OIL example on/off . . . 7
2 OIL example simple printer . . . 8
3 OIL examples simple printer separated concerns . . . 9
4 OIL scheduler overview . . . 10
5 OIL example printed sheet notifier . . . 11
6 OIL pipeline . . . 15
7 OIL example code generator . . . 16
8 OIL example naive scheduling . . . 19
9 OIL example basic scheduling improvement strategies . . . 21
10 OIL example simple . . . 24
11 OIL example multiple transitions labeled with the same event . . . 25
12 OIL example syntax analysis . . . 28
13 OIL example SMT analysis . . . 33
14 OIL example concern analysis . . . 38
15 OIL example unstable area analysis . . . 41
16 OIL example improved scheduling complementing current OIL run loop scheduler . 44
17 OIL example reduced scheduled OIL run loop scheduler events . . . 47
18 OIL example recursive scheduling . . . 49
19 OIL example recursive scheduling . . . 51
20 OIL example mealy machine . . . 54
21 OIL example hard to schedule . . . 55
22 OIL example created from LTS . . . 56
23 OIL example unstable area SMT analysis . . . 69
24 Chart results from Coffee . . . 78
25 Chart results from Game . . . 78
26 Chart results from EPC . . . 78
27 OIL example Coffee . . . 90
28 OIL example Game . . . 91

List of Algorithms

1 Pseudocode for a reactive event . . . 16
2 Pseudocode for r1 . . . 17
3 Pseudocode for the OIL run loop scheduler . . . 17
4 Pseudocode for a proactive event . . . 18
5 Pseudocode for p1 . . . 18
6 Pseudocode for the naive OIL run loop scheduler . . . 19
7 Pseudocode for the naive OIL run loop scheduler . . . 21
8 Pseudocode for the OIL run loop scheduler with removed entries . . . 22
9 Pseudocode for the OIL run loop scheduler with removed entries . . . 23
10 Pseudocode for the alternate OIL run loop scheduler . . . 23
11 Pseudocode for r1 . . . 45
12 Pseudocode for p2 . . . 46
13 Pseudocode for r1 with removed OIL run call . . . 47
14 Pseudocode for naive the OIL run loop scheduler . . . 48
15 Pseudocode for the OIL run loop scheduler with removed entries . . . 48
16 Pseudocode for a reactive event with recursive scheduling . . . 51

List of Tables

1 Calculated scores for prioritization . . . 53
2 OIL specifications information . . . 56
3 LTSs information . . . 57
4 Coverage information . . . 58
5 Amount events in selected traces . . . 60
6 EPC result overview OIL run loop scheduler . . . 61
7 EPC result overview recursive OIL scheduler . . . 61
8 EPC result overview OIL run loop scheduler . . . 62
9 EPC result overview recursive OIL scheduler . . . 62
10 EPC result overview OIL run loop scheduler . . . 63
11 EPC result overview recursive OIL scheduler . . . 63
12 EPC result overview OIL run loop scheduler . . . 64
13 EPC result overview recursive OIL scheduler . . . 64
14 EPC result overview OIL run loop scheduler . . . 65
15 EPC result overview recursive OIL scheduler . . . 65
16 Coffee result overview prioritizing events . . . 66
17 Game result overview compiler optimizations . . . 67
18 Coffee result overview OIL run loop scheduler . . . 72
19 Game result overview OIL run loop scheduler . . . 72
20 EPC result overview OIL run loop scheduler . . . 73
21 Coffee result overview recursive OIL scheduler . . . 74
22 Game result overview recursive OIL scheduler . . . 74
23 EPC result overview recursive OIL scheduler . . . 75
24 Coffee result overview prioritizing events . . . 76
25 Game result overview prioritizing events . . . 76
26 EPC result overview prioritizing events . . . 76
27 Coffee result overview compiler optimizations . . . 77
28 Game result overview compiler optimizations . . . 77
29 EPC result overview compiler optimizations . . . 77

1 Introduction

Canon Production Printing B.V. is a company that builds industrial printers. A lot of code is written to make these printers work. The complexity of this code is continuously increasing which makes the code harder to read and maintain. Therefore it is necessary to look for ways to reduce this complexity for software engineers.

One approach for reducing the complexity for software engineers is by making models of the systems you want to create. This can be accomplished by so-called domain-specific languages (DSLs). These DSLs are created to accomplish specific tasks and are valuable because, if well-designed, they can be much easier to use than general-purpose languages. A DSL improves the productivity of the programmer and is a way to improve communication. This is because it is not always necessary to be a programmer to understand a well-designed DSL.

Canon Production Printing B.V. developed such a DSL to try to improve the process of implementing software components for their industrial printers. This language is called OIL, short for Open Interaction Language. This is a textual language in which system behavior can be specified. While printing is the primary business domain of Canon Production Printing B.V., OIL contains no language constructs or logic specifically tailored to the domain of printing.

In OIL, system behavior is described using state machines that respond to incoming events. These state machines can keep track of the state of the system and can send out events themselves. OIL utilizes a concept of separation of concerns which helps engineers to cope with complex behavior. These concerns enable the modeling of specific aspects of the system separately in a concise way. OIL has also been designed to have a readable and unambiguous visual representation. These visualizations can be used during discussions among engineers.

Currently, two tools are specifically created to have OIL specifications as input; a version written in Python and one made using Spoofax[1]. Spoofax is a platform specifically designed for creating textual domain-specific programming languages. Both tools can transform OIL specifications to C++ code and mCRL2[2] specifications. However, there are a few differences; the Python tool was created first and code generated by it is already being used in production. The Python tool can simulate OIL specifications which is not possible in the tool created with Spoofax. The tool that uses Spoofax supports the initial DSL based on XML files but also supports a second DSL that does not depend on XML. XML was initially being used because parsers for XML were already available. The Spoofax-based tool is newer and aims to be more maintainable[3]. It is also the version this project focuses on.

This project takes a closer look at the C++ code generator. In the generated code, there is a part that schedules the events that the OIL specification sends out; proactive events. Currently, this scheduler is naive; all scheduling is done during runtime and uses little information from the original OIL specification. This project aims to make this scheduler more efficient so it has to do fewer computations during runtime. This would ideally result in generated C++ code needing fewer resources.

This report starts with some background information in Chapter 2. Next, the problem description containing the research questions can be found in Chapter 3. This chapter is followed by Chapter 4 which discusses basic improvement strategies for the naive scheduler. Afterward, there are two chapters describing a strategy for improving the naive scheduler by gathering causal relations between transitions. This includes an overview of the approaches for the gathering of causal relations in Chapter 5, followed by a chapter describing the strategies how these causal relations can be used; Chapter 6. In Chapter 7 a supporting strategy is discussed in which the original OIL specifications are altered to make it easier to apply the previously mentioned scheduling strategies. Afterward, Chapter 8 is dedicated to verifying the scheduling strategies to gain trust that the generated code is correct. Next, Chapter 9 lists the results that are acquired using the different scheduling strategies. The report finishes with Chapter 10 for the conclusions and future work.

2 Context

This chapter supplies more information on concepts that are used throughout this report.

2.1 What is OIL

OIL, short for Open Interaction Language is a textual language in which system behavior can be specified. An OIL specification can be used to generate C++ code that implements these OIL specifications[6]. Additionally, mCRL2 specifications can be generated to analyze the behavior of the created OIL specifications. The semantics of this language are defined in [4]. This chapter provides an informal description of the OIL language.

Basics Two kinds of OIL specifications exist; component and protocol specifications. This report only focuses on component specifications. Each OIL (component) specification consists of three parts: the declaration of *instance variables*, the definition of *areas*, and the specification of *transitions*. The instance variables are used to keep track of the state of the component. An instance state of the state machine gives a value to each of the instance variables. How the instance state changes, depends on the areas and transitions. An area can be one of three things: a *state*, a *scope*, or a *region*. These areas are organized in a tree structure and each area can have a parent area and ancestor areas.

A state represents a value for a specific instance variable. The closest ancestor region also referred to as the corresponding region dictates which instance variable this is. A scope can be used to restrict behavior using an invariant that always has to hold. An area can be active, this depends on the type of area. A scope is active when all its ancestor areas are active and the invariant of this scope holds in the instance state. A region is active when all its ancestor areas are active. Next, a state is active when all its ancestor areas are active and the instance variable indicated by the corresponding region of the state has the same value as the state.

Lastly, an OIL specification contains transitions. Each transition has a source and target area and is labeled with an event. If the source area is active the transition is enabled. Handling the event of an enabled transition results in the transition being fired. In this case, the values from the instance state are changed so the values of the target state and its ancestor states can be found on the instance variables of their corresponding regions. Note that this target area does not have to become active as scopes can still have an invariant that is not true. Some more advanced concepts can be used with transitions that can restrict their behavior or introduce additional updates to instance variables, these are discussed later in this report.

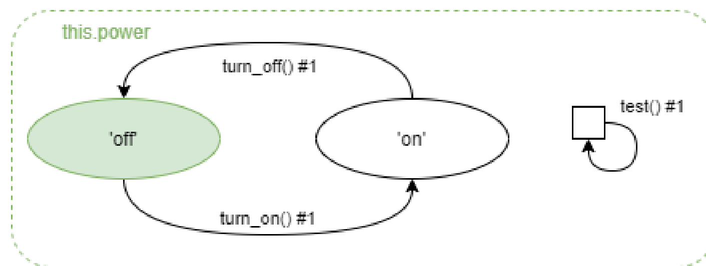

Figure 1: OIL example on/off

Example 2.1 *In Figure 1 a visualization of an OIL specification can be seen. It models a very basic component that can be turned on and off. The component can additionally always handle a test event. This example has two states 'off' and 'on' indicated with the ovals containing the values 'off' and 'on'. It has one region referencing the instance variable this.power shown using the dotted rounded rectangle. Next, there are three transitions indicated with the arrows labeled with the events turn_on, turn_off and test. Note that the labels are followed by an "#" followed by a number. These numbers are used to identify a specific transition if multiple transitions are labeled with the same event. On the right side, a scope can be spotted indicated with the square. This scope does not have an invariant and is therefore always active as all its ancestor areas are also always active. Note that this scope has only the region as its ancestor which is always active as it has no ancestor areas and is a region. The state 'off' is colored indicating that this state is active in the current instance state. The explanation uses this instance state as the initial instance state of this OIL specification.*

The instance variable *this.power* can have two values: *'on'* or *'off'*. In the initial instance state the value for *this.power* is *'off'* and the transition labeled with the *turn_on* event is enabled. When this event is handled, the instance variable *this.power* changes its value to *'on'*. In this instance state, the state *'on'* is active. Here the transition labeled with the *turn_off* event is enabled. Next, the transition labeled with the event *test* is always enabled. Note that this transition does not alter any instance variable.

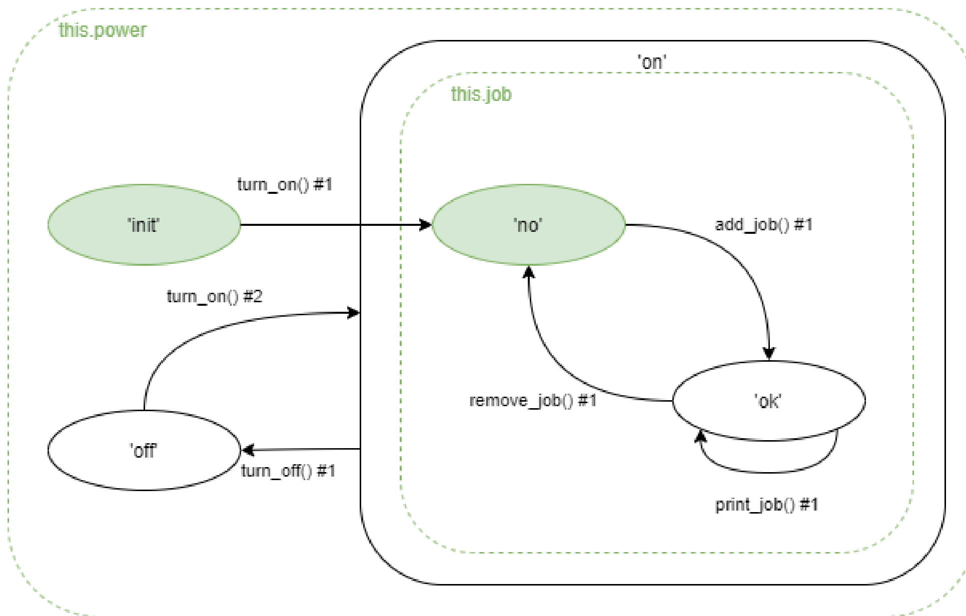

Figure 2: OIL example simple printer

Example 2.2 *In the OIL specification in Figure 2 a more complex component is modeled that represents a simple printer. This printer can be turned on and off. While this printer is on it can accept and remove a print job and when the print job is added the component can print the job. This example has the following states: 'init', 'no', 'ok', 'off' and 'on'. Next, it has regions referencing to the instance variables this.power and this.job. Additionally, transitions are visible with the following events on the labels; turn_on, turn_off, add_job, remove_job and print_job. For this example, we again assume that the initial instance state is depicted by the colored states.*

The instance variable $this.power$ can have the value $'init'$, $'on'$ or $'off'$ and the instance variable $this.job$ can have the value $'no'$ or $'ok'$. In the initial instance state, the instance variable $this.power$ has the value $'init'$ and the instance variable $this.job$ has the value $'no'$. From this initial instance state, there is only one enabled transition; the transition labeled with the event $turn_on$. When this event is handled, the instance variable $this.power$ is set to $'on'$ and the instance variable for $this.job$ is set to $'no'$. The instance variable $this.power$ is altered because the target area of the transition is nested in the state $'on'$. In this new instance state the transitions labeled with the $turn_off$ and add_job events are enabled. If the event add_job is handled the instance variable $this.job$ changes its value to $'ok'$ after which the transitions labeled with the $print_job$, $remove_job$ and $turn_off$ event are enabled. If the $turn_off$ event is handled from this new instance state the instance variable $this.power$ becomes $'off'$. In this instance state, only the transition labeled with the $turn_on$ event is enabled. There is a small difference between this enabled transition and the transition that was enabled from the initial instance state. The second transition labeled with the $turn_on$ event only connects to the state $'on'$ and it does not go into the region job connecting to the state $'no'$. Firing this second transition labeled with the $turn_on$ event therefore only alters the value for the instance variable $this.power$ and the value for $this.job$ is not altered. This approach ensures the component can be turned off and on without losing the added print jobs.

Separation of concerns OIL has a concept called separation of concerns. The idea is that different aspects of the component are modeled separately which helps to deal with complexity. Each transition can be labeled with a concern that focuses on a specific part of the behavior. An event is only enabled if for all concerns there is at least one enabled transition labeled with that event and concern. Concerns that do not have a transition labeled with the specific event are ignored in this synchronization.

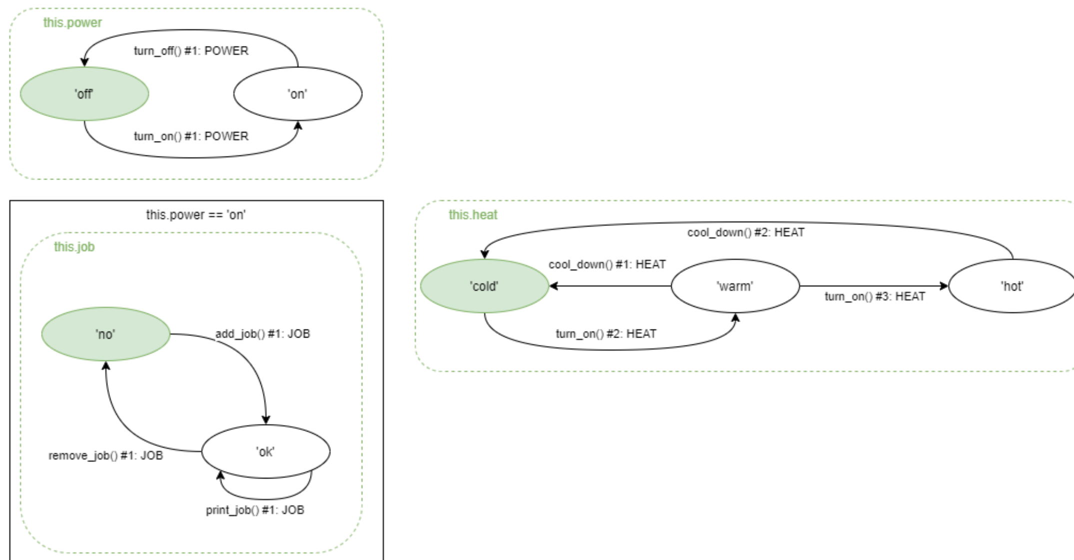

Figure 3: OIL examples simple printer separated concerns

Example 2.3 *The left part of the visualized OIL specification in Figure 3 has the same behavior as Example 2.2. Only now the OIL specification is split into two concerns. Some behavior is added with the right part; the component can cool down after it has turned on and even has to cool down after it has been turned on twice before it can be turned on for the third time. The model is now split up into three concerns; POWER, JOB, and HEAT. The concern of a transition is shown in the label after the ":" character. Additionally, a scope can be spotted indicated with the rectangle.*

The scope is used to ensure that the original behavior of the printer of Example 2.2 is kept. It does this by enforcing that the transitions labeled with the events $remove_job$, add_job and $print_job$ are only enabled whenever the instance variable $this.power$ has the value $'on'$. In this example the event $turn_on$ is restricted with the help of concerns; it can only be handled if both the concerns for $POWER$ and $HEAT$ can handle the $turn_on$ event. So if the instance variable $this.heat$ has the value $'hot'$ a $cool_down$ event has to be handled before a $turn_on$ event can be handled by the component. The concern $HEAT$, therefore, ensures that the component can not handle the $turn_on$ event three times in a row. The $cool_down$ event can still be used however to make additional $turn_on$ events possible. Note that the concern JOB has been ignored in this synchronization because it does not have any transition labeled with the event $turn_on$ and is therefore not taken into account.

Additional transition concepts Until now all examples only used basic transitions. However these transitions can be used with some more advanced concepts; it is possible to add a guard, assignments, and an assert to transitions. Guards can be added to restrict behavior while assignments are used to alter instance variables. Asserts are there to ensure certain properties hold after the transition itself has fired. Furthermore, events containing parameters can be handled. A transition can give expressions to determine the value for these parameters called arguments, these arguments are however optional. When a transition does not supply arguments it has to fire at the same time as another transition with defined arguments to be enabled; the values for the parameters can then be determined by the arguments of this additional transition. These concepts are discussed in more detail later in this chapter using Figure 5.

Next to this, a transition can be labeled with one of two kinds of events; reactive and proactive events. A reactive event is an event that comes from the environment of the component, while a proactive event is an event that comes from the component itself. A proactive event can be specified to send back information to the environment of the component, but it is also possible that it is specified to keep the event internal. A proactive event that is kept internal is referred to as a silent proactive event. Proactive events are produced instead of handled like reactive events. Producing is always done with priority over handling to acquire the run-to-completion semantics of OIL; a reactive event can only be handled when no proactive event can be produced anymore. This behavior is accomplished with a scheduler that schedules all relevant proactive events.

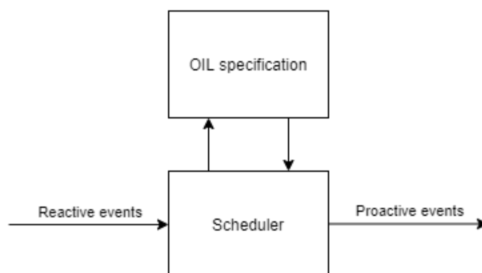

Figure 4: OIL scheduler overview

In the case multiple proactive events are enabled, an arbitrary one can be chosen by the scheduler. OIL specifications are only valid when it does not matter which proactive event is chosen. This is accomplished by requiring that the same events are always produced no matter the choices made by the scheduler. This property is called confluent proactivity. There are a few additional properties that have to hold for a valid OIL specification which are not used in this report but can be found in [4]. This report only considers valid OIL specifications.

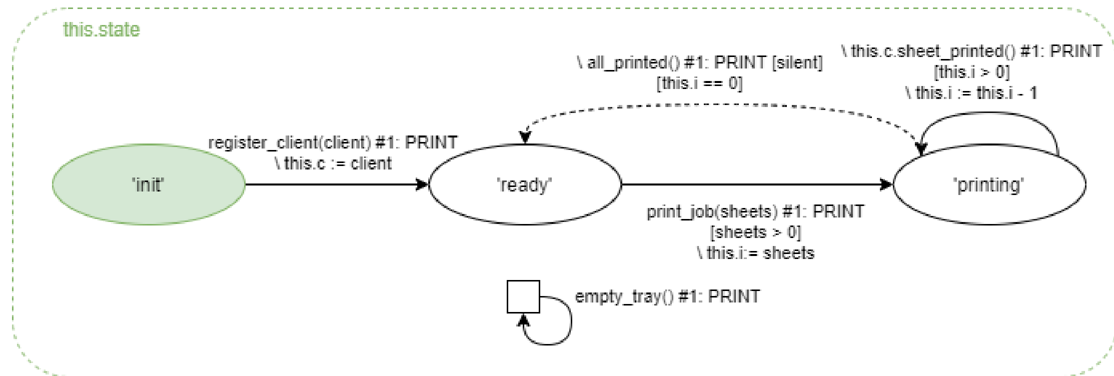

Figure 5: OIL example printed sheet notifier

Example 2.4 *In Figure 5, a visualization of an OIL specification can be seen with some more advanced transitions. It models a component in which a client can be registered. After this registration, the model accepts print jobs and notifies the registered client of every page the printer prints. When all sheets are printed the printer can accept a new job. Before and after each job it is possible to empty the tray in which the printed jobs are placed.*

A few expressions are visible located between square brackets; these are guards. There are also assignments that are indicated using ":=" after a "\" character. Two kinds of events are used on the labels of the transitions. There are two transitions labeled with proactive events which are indicated by the "\" character before the event label. Three transitions are also labeled with reactive events which are indicated by no "\" character before the event label. One of these proactive events is also silent indicated with a dashed arrow and the word silent between square brackets.

Take a look at the transition from the state *'init'* to *'ready'* in Example 2.4. This transition is labeled with a reactive event including a parameter that is assigned to an instance variable c. This instance variable is used to keep a reference to a client instance. Looking at the transition from *'ready'* to *'printing'* a guard can be spotted that ensures the transition is only enabled if there are more than zero sheets in the print job. Next, there are a few transitions that leave the state *'printing'*. One of these goes from *'printing'* to *'printing'* labeled with the event *sheet_printed*. This is a proactive event as it has a "\" before the event label. As long as there are sheets left to print the method *sheet_printed* is called on the client instance c sending information back to the environment. Next there is a transition from *'printing'* back to *'ready'*. This transition is also labeled with a proactive event, but this event is internally produced because it is silent. While there are two transitions labeled with a proactive event leaving the state *'printing'* the guards ensure only one of them can be fired, so no arbitrary choice has to be made here. Finally, there is a transition labeled with the event *empty_tray* which could be used to empty the tray of printed jobs. This transition is placed on an empty scope so it is always enabled as all the ancestor areas of this scope are always active. Note that this scope has only the region as its ancestor which is always active as it has no ancestor areas and is a region. However, the labeled event on this transition is reactive and has a lower priority than the two proactive events. Therefore, the *empty_tray* event can never be handled when the state *'printing'* is active because either the guard $i > 0$ or $i == 0$ holds. So this OIL specification does not allow the tray to be emptied while the printer is printing.

Illegal events In the examples we have seen up to this point only events were considered which were enabled. An event is only enabled if for all concerns with a transition labeled with that event there is at least one of these transitions enabled. Note that this means that events are always enabled if no concerns are used. It however could mean that the instance state is not altered if an event is handled as it is possible that no transitions are fired.

It is possible that an event occurs that is not enabled or that an assertion does not hold after the firing of a transition labeled with this event. In this case, the OIL specification received an illegal event. It is not possible to handle any new event after an illegal event occurred. Behavior after receiving an illegal event is not considered in this report as the OIL specification has crashed.

In Example 2.1 the transitions are not labeled with concerns or have asserts. Therefore this OIL specification has no illegal events as for each concern there is at least one enabled transition. So if the area $'off'$ is active the event *turn_off* is enabled and can be handled. Note, that the handling of this event will not alter the instance state of this OIL specification as no transition labeled with the event *turn_on* is enabled.

Example 2.3 does have transitions labeled with concerns. The *POWER* concern shows very similar behavior as we have seen in Example 2.1. However, the concerns on the labels of the transitions alter the behavior slightly; whenever the area $'off'$ is active the event *turn_off* is not enabled and can not be handled. The concern *POWER* does not have an enabled transition labeled with the event *turn_off*. So in this instance state the *turn_off* event is an illegal event. If for some reason the OIL specification receives the *turn_off* event it is not possible to handle any additional events and the OIL specification has crashed.

2.2 Formal definitions

This report supplies formal definitions for some used concepts. OIL is formally defined[4], this chapter supplies a small recap.

Variables Let X be the set of all variables and $\mathbb{V}^X$ be the set of all valuations over X. Next, let EXP_X be the set of all expressions over the variables X. Whenever events occur new values may be assigned to variables. Expressions are extended with 'old' variables. In such expressions, variables x refer to the state after the last event occurred, and variables x^{old} refer to the state before the last event occurred. The notation $[\![exp]\!]v$ where $exp \in EXP_X$ can be used to evaluate an expression with the evaluation found in $v \in \mathbb{V}^X$.

IOLTS An IOLTS is defined as a tuple $(S, s_0, I, O, H, \rightarrow)$. An IOLTS is a model with states and transitions that models the behavior of a component. The set S denotes the set of states and the set $\rightarrow$ denotes the transitions between them. These transitions are labeled with actions which can be sent to the component or sent from the component itself. There are three sets that denote these actions; incoming actions I, outgoing actions O and internal actions H. We write $s \xrightarrow{act} s' \iff (s, act, s') \in\rightarrow$ for the states $s \in S$, $s' \in S$ and action $act \in (I \cup O \cup H)$ which means that there is a transition between the state s and s' labeled with the action act. Next, we write $s \xrightarrow{act} \iff \exists_{n \in S}(s \xrightarrow{act} n)$ which is used when there exists a transition leaving the state $s \in S$ labeled with the action $act \in (I \cup O \cup H)$. We write $\xrightarrow{act} s \iff \exists_{n \in S}(n \xrightarrow{act} s)$ which is used when there exists a transition entering the state $s \in S$ labeled with the action $act \in (I \cup O \cup H)$.

OIL specification An OIL specifications is defined as a tuple $(\mathbb{X}, \mathbb{A}, \mathbb{T})$. The tuple consists of three parts; $\mathbb{X}$ declares the instance variables, parameters and their initial values, the areas are defined in $\mathbb{A}$ and the transitions are specified in $\mathbb{T}$.

The tuple $\mathbb{X} = (X, \mathcal{I})$ is the variable specification.

- The set X represents all variables which is partitioned in the set of instance variables X_I and parameters X_P.
- The set $\mathcal{I} \in \mathbb{V}^{X_I}$ represents the initial values of the instance variables.

The tuple $\mathbb{A} = (A, \sqsubset, \mathcal{RE}, \mathcal{EXP})$ is the area specification.

- The set A represents all areas of the OIL specification. The set A is partitioned in three sets A_{Re}, A_{St} and A_{Sc}. The set A_{Re} represents all regions. The set A_{St} represents all states and the set A_{Sc} represents all scopes.
- The $\sqsubset$ relation is used to define which areas are ancestor areas of each other; $a \sqsubset a'$ indicates a' is a ancestor area of a. The $\sqsubseteq^*$ relation is the reflexive transitive closure of the $\sqsubset$ relation.
- The function $\mathcal{RE}$ associates each state with its corresponding region.
- The function $\mathcal{EXP}$ associates each area with an expression. For regions, this expression is always a variable in X_I. For states, this expression always is a constant value in EXP. For scopes, this expression always is a boolean expression in EXP_{X_I}

The tuple $\mathbb{T} = (E, \mathcal{PAR}, T, CO, \mathcal{CO})$ is the transition specification.

- The set E represents the set of all events and this set is partitioned in the set for proactive events E_P and the set for reactive events E_R.
- The function $\mathcal{PAR}$ associates each event with its set of parameters.
- The set T represents all the transitions in the OIL specification.
- The set CO represents all concerns in the OIL specification.
- The function $\mathcal{CO}$ associates each transition with the concerns it is labeled with. It also associates a set of transitions to the union of all of the concerns that each transition is labeled with.

Transitions The tuple $(so, gu, e, \mathcal{ARG}, AG, ta, ar) \in T$ is a transition.

- The area $so \in A$ represents the source area.
- The boolean expression $gu \in EXP_X$ represents the guard.
- The event $e \in E$ represents the event on the label.
- The function $\mathcal{ARG}$ associates each parameter from $\mathcal{PAR}(e)$ with its argument expression.
- The set AG represents the assignments.
- The area $ta \in A$ represents the target area.
- The boolean expression $ar \in EXP_X$ represents the assert.

The set of transitions that are labeled with the event $e \in E$ are denoted with T_e. Next, $T_{e,c}$ denotes the set of transitions that are labeled with the event $e \in E$ and are labeled with the concern $c \in CO$. The set T_e^v denotes the enabled transitions for the event $e \in E$ and valuation $v \in \mathbb{V}^X$

Other The function $\mathcal{AC}$ associates each area with its area condition. The area condition is a boolean expression that represents if the area is active. As stated earlier it depends on the type of area when an area is active. A scope is active when all its ancestor areas are active and the invariant of this scope holds in the instance state. A region is active when all its ancestor areas are active. Next, a state is active when all its ancestor areas are active and the instance variable indicated by the corresponding region of the state has the same value as the state. This definition looks as follows for $a \in A$.

$$\mathcal{AC}(a) = \bigwedge\{\mathcal{EXP}(\mathcal{RE}(a')) = \mathcal{EXP}(a')|a' \in A_{St} \wedge a \sqsubseteq^* a'\} \wedge \bigwedge\{\mathcal{EXP}(a')|a' \in A_{Sc} \wedge a \sqsubseteq^* a'\}$$

Next, the function $\mathcal{PRC}$ associates each transition with a boolean expression checking if the transition is enabled. This boolean expression makes sure the source area is active and the guard does hold. The $\mathcal{PRC}$ function is defined for each $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar) \in T$

$$\mathcal{PRC}(t) = \mathcal{AC}(so) \wedge gu \wedge \bigwedge\{p = \mathcal{ARG}(p)|p \in dom(\mathcal{ARG})\}$$

The function $\mathcal{POC}$ associates each transition with a boolean expression that represents if the transition correctly fired. It makes sure the target area is active and the assert does hold. The $\mathcal{POC}$ function is defined for each $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar) \in T$

$$\mathcal{POC}(t) = \mathcal{AC}(ta) \wedge ar$$

The $\mathcal{POC}$ function also associates a set of transitions with a boolean expression. In this case, the definition looks as follows for each $T' \subseteq T$.

$$\mathcal{POC}(T') = \bigwedge_{t \in T'} \mathcal{POC}(t)$$

The function CC associates each event with its concern condition. This concern condition is a boolean expression that represents if the event is enabled. As stated earlier an event is only enabled if for each concern with a transition labeled with the event there is a transition enabled.

$$\mathcal{CC}(e) = \bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t \in T_{e,c}} \mathcal{PRC}(t)$$

The $\mathcal{AU}$ function associates an area with its area update. The area update is a set of assignments needed to make the area active. The function does not consider assignments needed to make invariants from scopes hold.

$$\mathcal{AU}(a) = \{\mathcal{EXP}(\mathcal{RE}(a')) := \mathcal{EXP}(a')|a' \in A_{St} \wedge a \sqsubseteq^* a'\}$$

Next, $\mathcal{U}$ associates a transition with all the updates that take place on the instance variables when the transition is fired. These updates can be the result of the target area becoming active or assignments in the AG set.

$$\mathcal{U}(t) = \mathcal{AU}(ta) \cup AG$$

The $\mathcal{U}$ function also associates a set of transitions with all the updates that take place on the instance variables when these transitions are fired. In this case, the definition looks as follows for each $T' \subseteq T$.

$$\mathcal{U}(T') = \bigcup_{t \in T'} \mathcal{U}(t)$$

Next, an OIL specification can be defined as an IOLTS with the so-called acceptor semantics. In these acceptor semantics the transitions for the IOLTS can be defined as follows. To model if the OIL specification crashed after receiving an illegal event the $failure$ state is used. The $failure$ state is denoted as $\textcircled{F}$. There are transitions defined that leave the $failure$ state that are always labeled with the $fail$ event. During the analysis of OIL specifications using mCRL2 these $fail$ events are used to detect if the OIL specification has postconditions that do not hold and the received event was an illegal event.

Where $e \in E$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $S_{\textcircled{F}} = \mathbb{V}^{X_I} \cup \textcircled{F}$, $s \in S$, $s' \in S$ and $v = s \cup p$.

$$
\begin{aligned}
s \xrightarrow{e(p)} s' &\iff [\![\mathcal{CC}(e)]\!]v \wedge s' = v[\mathcal{U}(T_e^v)] \wedge [\![\mathcal{POC}(T_e^v)]\!]^v_{\mathcal{U}(T_e^v)} \\
s \xrightarrow{e(p)} \textcircled{F} &\iff [\![\mathcal{CC}(e)]\!]v \wedge \neg[\![\mathcal{POC}(T_e^v)]\!]^v_{\mathcal{U}(T_v^e)} \\
\textcircled{F} \xrightarrow{fail} \textcircled{F} &
\end{aligned}
$$

2.3 Code generation

Currently, there is a tool specifically created to have OIL specifications as input made using Spoofax. Spoofax is a platform designed for creating textual domain-specific languages. The Spoofax platform can generate parsers and type checkers which can be used to acquire abstract syntax trees of source files. Next, the Spoofax platform has a language called Stratego[5] which can be used to define transformations on these abstract syntax trees. These transformations are used to generate C++ code and mCRL2 specifications from OIL specifications[6].

Overview code generation The code generation for OIL specifications in the Spoofax tooling is done with a few transformations to different abstract syntax trees. An overview is given in Figure 6.

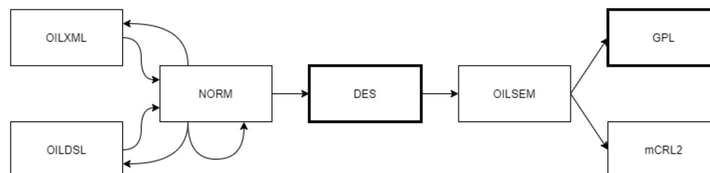

Figure 6: OIL pipeline

The OIL tooling made with Spoofax can generate code from two kinds of OIL specifications. The first one is based on XML (OILXML) while the second is using a custom DSL (OILDSL) created for OIL. Both abstract syntax trees of these OIL specifications are transformed to a normalized abstract syntax tree of OIL (NORM), this transformation does not alter used concepts in the OIL specification in any way but is used as common ground to continue. It is also possible to go back to both the DSL or XML-based OIL specifications. This makes it possible to transform an XML-based OIL specification into a DSL-based OIL specification and vice versa. A few desugaring and explication transformations can be used on this normalized form to refine different aspects of an OIL specification. For example, the initial states can be defined implicitly and an auto transformation is used to generate these initial states explicitly.

Next, the normalized abstract syntax tree of OIL is transformed to a desugared abstract syntax tree (DES). In this transformation, a lot of information is standardized. For example, empty guards are set to true. Next, a transformation to the semantic abstract syntax tree (OILSEM) is done. This abstract syntax tree is close to the formally defined semantics of OIL[4]. This includes things such as the grouping of all transitions per event. At last, a transformation to the GPL abstract syntax tree can be done. This representation only uses concepts from an object-oriented general-purpose language. This representation maps easily to for example C++.

Overview generated code The Spoofax implementation of OIL can create C++ code from OIL specifications[6]. It generates a class that has a few members corresponding to the instance variables of the OIL specification. It also generates a method for each reactive and proactive event in the OIL specification. These methods are used by the naive OIL run loop scheduler to achieve the run-to-completion semantics of proactive events.

To aid the explanation the following OIL specification is used.

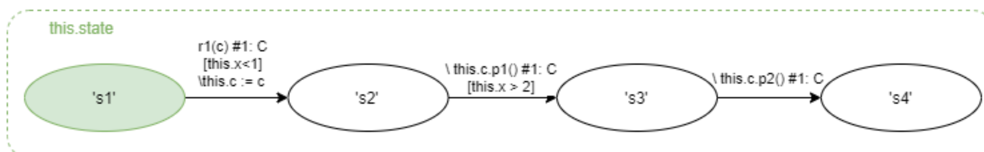

Figure 7: OIL example code generator

Example 2.5 *Three transitions can be spotted. The transition r1#1 is labeled with a reactive event, has a guard and has an assignment. Both the transitions p1#1 and p2#1 are labeled with proactive events. Note that p1#1 also has a guard.*

Algorithm 1: Pseudocode for a reactive event

```
1  Function reactiveExample:
2      /* Checking precondition reactiveExample                          */
3      if not (AREA_CONDITION_source_area and guard) then
4          /* Can not handle reactive event reactiveExample               */
5
6      /* Update instance variables reactiveExample                       */
7      AREA_UPDATE_reactive_example()
8
9      /* Checking postcondition reactiveExample                          */
10     if not (AREA_CONDITION_target_area and assert) then
11         /* The postcondition of transition reactiveExample #1 failed   */
12
13     /* Produced event reactiveExample                                  */
14     _OIL_RUN()
```

In Algorithm 1 an example can be seen of a method generated for a reactive event. It is split up into four parts. First, the precondition is checked. Such a precondition does check if there is a transition enabled for each concern labeled with the event. A transition is enabled if the source area is active which is checked by calling the *AREA_CONDITION* method generated for each area. Next, the guard has to hold in the current instance state.

After the precondition is checked, the *AREA_UPDATE* method of the transitions is executed; these are generated methods that are used to update the instance variables so they match the values of target states and their ancestor states. An event with multiple transitions can have multiple of these area updates.

Next, the postcondition is checked. It checks that all enabled transitions did fire correctly by making sure their target area is active with an *AREA_CONDITION* method. The postcondition also checks if the assert of the enabled transitions holds in the new instance state.

At last, the *_OIL_RUN* method is called which is the method that contains the scheduler that ensures the run-to-completion semantics of OIL. If either the pre or postconditions do not hold an error is returned as the component has received an illegal event.

Taking this into account, the generated code for the *r1* event looks as follows.

Algorithm 2: Pseudocode for r1

```
1  Function r1(c):
2      /* Checking precondition r1                                        */
3      if not (this.state = 1 and this.x < 1) then
4          /* Can not handle reactive event r1                            */
5
6      /* Update instance variables r1                                    */
7      this.c := c
8      this.state := 2
9
10     /* Checking postcondition r1                                       */
11     if not (this.state = 2) then
12         /* The postcondition of transition r1 #1 failed                */
13
14     /* Handled event r1                                                */
15     _OIL_RUN()
```

Try methods are generated for each proactive event and return the boolean value *true* if they produce their event. The naive OIL run loop scheduler ensures the run-to-completion semantics by calling all these try methods. If one of the proactive events got actually produced the scheduler calls all try methods again. This pattern is repeated until no try method returns *true* and no proactive event can be produced anymore. In this way, the run-to-completion semantics is guaranteed. The naive OIL run loop scheduler is generated in the following way for the example.

Algorithm 3: Pseudocode for the OIL run loop scheduler

```
1  Function _OIL_RUN:
2      _scheduler_busy := true
3
4      while _scheduler_busy do
5          _scheduler_busy := false
6          _scheduler_busy := (TRY_EVENT_p1() or _scheduler_busy)
7          _scheduler_busy := (TRY_EVENT_p2() or _scheduler_busy)
8
```

In the *_OIL_RUN* method, a while loop can be found that only stops looping if no proactive try method succeeded to produce their proactive event. If one can be produced however a boolean variable called *_scheduler_busy* is set to true and all proactive events are tried again. The short circuit evaluation of a boolean expression is used to accomplish that all try event method calls in the naive OIL run loop scheduler are always called in each iteration. The *_OIL_RUN* method only returns when no proactive events can be produced anymore. Note that this approach is naive and results in the scheduling of proactive events all being done during run time.

Algorithm 4: Pseudocode for a proactive event

```
1  Function TRY_EVENT_proactiveExample:
2      _oil_e_enabled := AREA_CONDITION_source_area()
3
4      /* Checking precondition proactiveExample                                  */
5      if _oil_e_enabled then
6          /* Update instance variables proactiveExample                          */
7          AREA_UPDATE_reactive_example()
8
9          /* Checking postcondition proactiveExample                             */
10         if not AREA_CONDITION_target_area then
11             /* The postcondition of transition proactiveExample #1 failed     */
12
13         proactiveExample()
14         /* Produced event proactiveExample                                      */
15
16     return _oil_e_enabled
```

In Algorithm 4 a try method generated for a proactive event is visible. The main difference with a method generated for a reactive event is that this method does not return an error if the precondition does not hold. In the case the precondition does not hold, the method just returns *false* and does not alter anything. This method also does not include a call to the *_OIL_RUN* method. The scheduler is only called after a reactive event is handled.

The try method looks as follows for the event *p1* from the example.

Algorithm 5: Pseudocode for p1

```
1  Function TRY_EVENT_p1:
2      _oil_e_enabled := this.state = 2 and this.x < 2
3
4      /* Checking precondition p1                                                */
5      if _oil_e_enabled then
6          /* Update instance variables p1                                        */
7          this.state := 3
8
9          /* Checking postcondition p1                                           */
10         if not this.state = 3 then
11             /* The postcondition of transition p1 #1 failed                   */
12
13         this.c.p1()
14         /* Produced event p1                                                    */
15
16     return _oil_e_enabled
```

3 Problem description

The naive scheduler makes all decisions at runtime and uses little information from the original OIL specification. This results in the scheduling of proactive events taking a significant time during the execution of the generated C++ code. A small OIL specification is used to illustrate this.

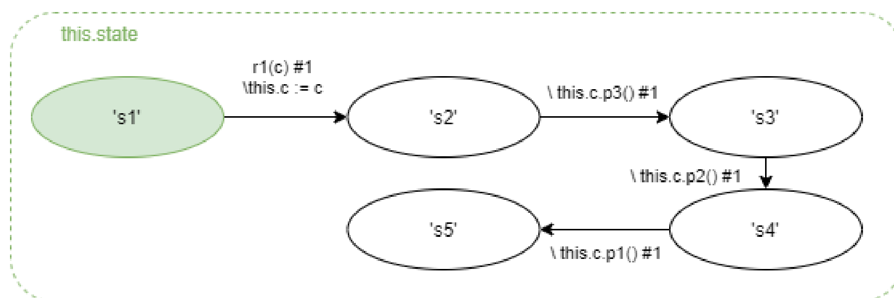

Figure 8: OIL example naive scheduling

Example 3.1 *The OIL specification in Figure 8 has one reactive event r1 and three proactive events p1, p2, and p3. This results in the following OIL run loop being generated.*

Algorithm 6: Pseudocode for the naive OIL run loop scheduler

```
1 Function _OIL_RUN:
2     _scheduler_busy := true
3
4     while _scheduler_busy do
5         _scheduler_busy := false
6         _scheduler_busy := (TRY_EVENT_p1() or _scheduler_busy)
7         _scheduler_busy := (TRY_EVENT_p2() or _scheduler_busy)
8         _scheduler_busy := (TRY_EVENT_p3() or _scheduler_busy)
9
```

For scheduling this OIL specification the following order of events is tried; *p1*, *p2*, *p3*, *p1*, *p2*, *p3*, *p1*, *p2*, *p3*. It is however sufficient to try the events in the following order; *p3*, *p2*, *p1*.

Improving the naive scheduler for proactive events should ideally result in fewer try method calls being computed during the execution of the generated C++ code. Try method calls that do not result in the production of an event could be skipped if the scheduler could guarantee the run-to-completion semantics in some other way. An improved scheduler should result in more efficient C++ code being generated from OIL specifications. Generating more efficient C++ code is useful especially when it is used in production as fewer resources are required to run the generated code.

The goal of this project is to investigate how the scheduling of proactive events can be improved and what the effect of these improvements is on the performance of the generated C++ code. To be able to improve the scheduling it is investigated which information can be used to remove redundant try method calls. Next, this project looks for ways to make sure that the scheduling is always correct and the run-to-completion semantics always holds even after the scheduler for proactive events is altered.

This problem is translated to the following research question:

RQ1 How can the naive scheduler for proactive events be improved, so fewer try methods have to be called?

- **RQ1.1** What information from OIL specifications can be used for improving the naive scheduler for proactive events?
- **RQ1.2** How to make sure that proposed schedulers for proactive events are always generated correctly and guarantee the run-to-completion semantics of OIL?

RQ2 What is the effect of improvements to the scheduler of proactive events on the performance of the generated C++ code?

4 Basic scheduling improvement strategies

During the project, a few scheduler improvement strategies were found which can be used without using any complex information gathering. These strategies are discussed in this chapter.

The following OIL specification is used to aid the explanation.

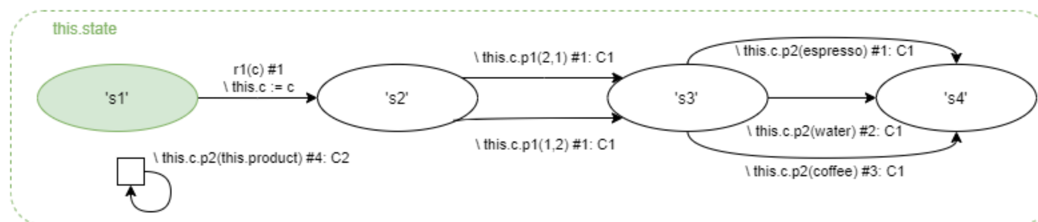

Figure 9: OIL example basic scheduling improvement strategies

Example 4.1 *This OIL specification has one reactive event r1 and two proactive events p1 and p2. Note that the proactive events have multiple argument combinations.*

The naive OIL run loop scheduler for this OIL specification looks as follows. Note that each possible argument combination is tried for each proactive event.

Algorithm 7: Pseudocode for the naive OIL run loop scheduler

```
1  Function _OIL_RUN:
2      _scheduler_busy := true
3
4      while _scheduler_busy do
5          _scheduler_busy := false
6          _scheduler_busy := (TRY_EVENT_p1(1, 1) or _scheduler_busy)
7          _scheduler_busy := (TRY_EVENT_p1(1, 2) or _scheduler_busy)
8          _scheduler_busy := (TRY_EVENT_p1(2, 1) or _scheduler_busy)
9          _scheduler_busy := (TRY_EVENT_p1(2, 2) or _scheduler_busy)
10         _scheduler_busy := (TRY_EVENT_p2(coffee) or _scheduler_busy)
11         _scheduler_busy := (TRY_EVENT_p2(water) or _scheduler_busy)
12         _scheduler_busy := (TRY_EVENT_p2(espresso) or _scheduler_busy)
13         _scheduler_busy := (TRY_EVENT_p2(this.product) or _scheduler_busy)
14
```

Trying existing argument expression combinations In the generation of the naive OIL run loop scheduler the cartesian product was used to obtain all possible combinations of arguments for scheduling an event. This is visible for the *p1* event. There are two transitions with the arguments $\{1, 2\}$ and $\{2, 1\}$. These defined arguments result in the following four combinations of arguments that are tried; $\{1, 1\}$, $\{1, 2\}$, $\{2, 1\}$ and $\{2, 2\}$. However, the production of a proactive event always corresponds to the firing of at least one transition. So therefore it is also sufficient to only try the combinations of arguments that are used on transitions. This reduces the number of combinations that have to be tried to schedule an event. There is the possibility that a transition has not all arguments defined, this is however not a problem as these transitions are fired together with at least another transition for which the arguments are known.

This scheduling strategy does not evaluate expressions so it is still possible that multiple arguments evaluate to the same argument value. This is visible for the argument *this.product* as it can only have the value *coffee*, *water* or *espresso* which are all in the OIL run loop scheduler already.

Trying all combinations of existing arguments looks as follows for the OIL run loop scheduler. Note that this approach reduces the number of events tried for the event *p1*

Algorithm 8: Pseudocode for the OIL run loop scheduler with removed entries

```
1  Function _OIL_RUN:
2      _scheduler_busy := true
3
4      while _scheduler_busy do
5          _scheduler_busy := false
6          _scheduler_busy := (TRY_EVENT_p1(1, 2) or _scheduler_busy)
7          _scheduler_busy := (TRY_EVENT_p1(2, 1) or _scheduler_busy)
8          _scheduler_busy := (TRY_EVENT_p2(coffee) or _scheduler_busy)
9          _scheduler_busy := (TRY_EVENT_p2(water) or _scheduler_busy)
10         _scheduler_busy := (TRY_EVENT_p2(espresso) or _scheduler_busy)
11         _scheduler_busy := (TRY_EVENT_p2(this.product) or _scheduler_busy)
12
```

Trying only arguments expression combinations from concern that has the least of them There may exist an event with parameters that is located in multiple concerns because concerns always have to be synchronized, it is allowed to only try the argument combinations of a single concern. By taking the concern that has the least of these combinations, the number of tried combinations can be reduced. This scheduler improvement strategy is only allowed when each transition in the concern with the least argument combinations has arguments defined for each of its parameters.

In the example, there are multiple transitions labeled with the event *p2* which are also labeled with different concerns; *C1*, *C2*. This information can be used to only try *p2*(*this.product*) as it is the only occurrence of the event *p2* in the concern *C2*. Note that each transition labeled with the *p2* event and *C2* concern has arguments defined for its parameter. This looks as follows in the OIL run loop scheduler.

Algorithm 9: Pseudocode for the OIL run loop scheduler with removed entries

```
1  Function _OIL_RUN:
2      _scheduler_busy := true
3
4      while _scheduler_busy do
5          _scheduler_busy := false
6          _scheduler_busy := (TRY_EVENT_p1(1, 2) or _scheduler_busy)
7          _scheduler_busy := (TRY_EVENT_p1(2, 1) or _scheduler_busy)
8          _scheduler_busy := (TRY_EVENT_p2(this.product) or _scheduler_busy)
9
```

Track last produced event The naive OIL run loop scheduler always tries all proactive events and only after they are all tried a check is conducted to see if any of them was successfully produced. If this is the case all events are tried again. This behavior is accomplished using a while loop. It is certainly possible that not all of the events have to be tried again to guarantee the run-to-completion semantics. The previous while loop iteration of tried events could already have had a few events that could not be produced. If no event can be found that can be produced there is no need for trying these events again. By keeping track of the last produced event the scheduler could decide to try a smaller subset of proactive events.

The generated code is altered for this scheduler. In this code structure, a counter is used that counts the events that are not produced since the last produced event. In the case an event is produced this counter is reset to zero. When the counter exceeds the total number of events it is known that all events have been tried and the remaining tries can safely be skipped. Below this scheduler can be seen that uses the same events as in Algorithm 9. The short circuit evaluation of a boolean expression is used to skip try method calls when the counter exceeds the total number of events.

Algorithm 10: Pseudocode for the alternate OIL run loop scheduler

```
1  Function _OIL_RUN:
2      _total_events := 3
3      _event_counter := 0
4
5      while _event_counter < _total_events do
6          _event_counter = ((_event_counter < _total_events) and
7                              TRY_EVENT_p1(1, 2))?0 : _event_counter + 1
8          _event_counter = ((_event_counter < _total_events) and
9                              TRY_EVENT_p1(2, 1))?0 : _event_counter + 1
10         _event_counter = ((_event_counter < _total_events) and
11                             TRY_EVENT_p2(this.product))?0 : _event_counter + 1
12
```

Iterating self-loops There are transitions labeled with a proactive event that have the same source and target area while the production of their event is limited by a guard. Possibly it is more effective to produce the events on the labels of these transitions separately by generating a loop that only exits when the guard does not hold. This can prevent that other events are tried when they are not needed to. This scheduling strategy is currently not implemented but it could still improve the scheduling.

5 Causal relations

In the previous chapter, a few scheduling strategies were discussed to improve the scheduling without using complex information gathering. However, information can be extracted from OIL specifications that can be used to enable the use of additional scheduling strategies. This chapter specifies which information is found to be useful and how it can be extracted from the OIL specifications.

5.1 Scheduler framework

The following OIL specification is used in the first part of this chapter. This example is used to motivate which information should be gathered. A formal definition of the gathered information is supplied later.

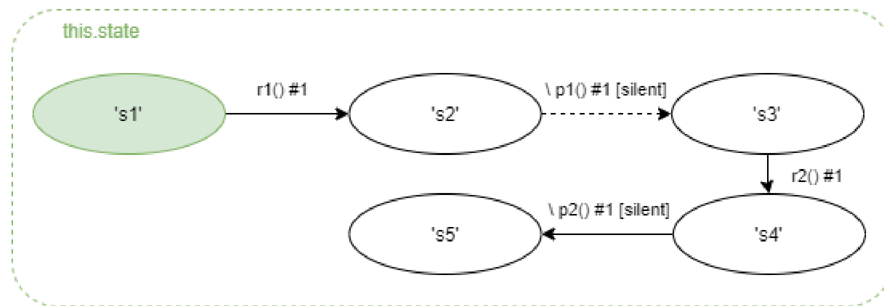

Figure 10: OIL example simple

Example 5.1 *This OIL specification has four transitions each labeled with its own event. Two of these events are reactive; r1 and r2. The other two events are proactive and need to be scheduled; p1 and p2.*

The naive OIL run loop scheduler just schedules all proactive events until none of them can be produced. In Example 5.1 it is clearly visible the scheduler should focus on producing the *p1* event after the *r1* event is handled; the *p1* event can always be produced after the *r1* event is handled. Additionally, there is no need for trying the *p2* event after the *r1* event is handled; the *p2* event can only be produced after the *r2* event. The naive OIL run loop scheduler however takes time trying to produce the *p2* event after the *r1* event.

So it is helpful to collect relations between events that tell which events can be produced after each other. These causal relations can then be used to try proactive events that can always be produced directly. There are however situations for which there are no events that are always enabled after a specific event. To still be able to improve the scheduling it is helpful to collect events that are never able to be produced after this event. With these causal relations, it can be determined that these events do not have to be tried as they can never be produced.

For Example 5.1 we focused on causal relations between events. There are however a few advantages to gathering these causal relations for transitions instead. These advantages are discussed using Example 5.2.

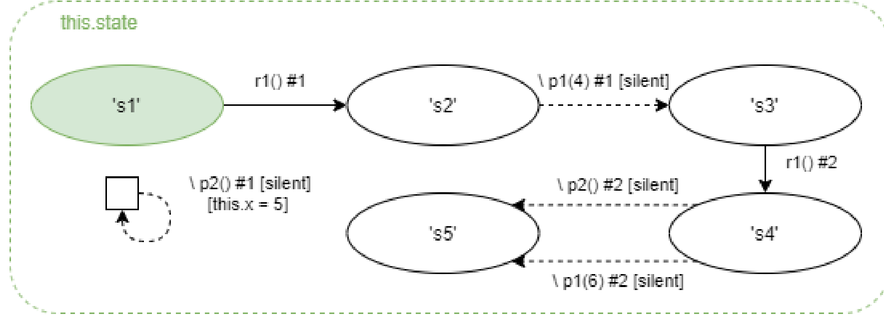

Figure 11: OIL example multiple transitions labeled with the same event

Example 5.2 *This example has six transitions. Two of them are labeled with a reactive event; r1#1 and r1#2. There are also four transitions labeled with a proactive event; p1#1, p1#2, p2#1 and p2#2.*

In Example 5.2 it is visible that multiple transitions can be labeled with the same event. If it is known that an event has been handled it may not be clear which of its transitions fired. This lost information reduces the optimization opportunities. In Example 5.2, it is not possible to produce the event *p2* after the transition *r1*#1 fired. However, there is another transition labeled with the same reactive event *r1*#2 after which *p2* always can be produced. Both transitions are labeled with the same reactive event *r1*. It can not be stated that the event *p2* always or never can be produced after the event r1 is handled. This is not a problem if the causal relations are gathered for transitions. We can now state that *p2#1* is always enabled and *p2#2* is never enabled after *r1#1* fired.

For this part, we introduce the concept of proactive transitions. These proactive transitions are transitions that are labeled with a proactive event. If the causal relations indicate that a specific proactive transition can always fire the correct arguments for this specific proactive transition can be generated by the scheduler. This is possible because each transition has at most one argument defined for each parameter. An event can have multiple argument expressions defined for each parameter from all the transitions labeled with this event. If only the event is known it is not possible to reconstruct the specific arguments expressions. This would mean that all argument expressions should be tried. In the example the event *p1* has two arguments defined; $\{4\}$ and $\{6\}$. If the causal relations are gathered for the events both argument expressions have to be tried when the event *p1* is scheduled. This is not the case when the causal relations are gathered for transitions; if for example, the event needs to be tried for the transition *p1*#1 the arguments $\{4\}$ can be used. This property however requires that each transition that is gathered for causal relations has to have arguments defined for each of its parameters. So for transitions that have not all arguments defined it is still necessary to try all argument expressions that are defined for transitions labeled with the same event.

Gathering the transitions instead of the events makes it possible to alter the scheduler in such a way that a part of the precondition does not have to be checked. Computations for checking which transitions are enabled or not could be skipped because the scheduler already has this information when causal relations are found regarding these transitions.

With this in mind, we can associate each transition with two sets of proactive transitions to indicate causal relations; which proactive transitions are always enabled and which are never enabled. The specific transition for which the sets are gathered is referred to as the preceding transition.

- Successor transition function (T_S): Transitions returned from T_S are proactive transitions that are always enabled after the preceding transition.
- No successor transition function (T_{NS}): Transitions returned from this function are proactive transitions that are never enabled after the preceding transition.

Formal definition For the formal definition of the causal relations, the formal OIL semantics[4] are used.

Let $(\mathbb{X}, \mathbb{A}, \mathbb{T})$ be an OIL specification and let $(S, s_0, I, O, H, \rightarrow)$ be an IOLTS that has the same behavior as the OIL specification $(\mathbb{X}, \mathbb{A}, \mathbb{T})$. The transition specification was defined earlier in Chapter 2.2 as a tuple $\mathbb{T} = (E, \mathcal{PAR}, T, CO, \mathcal{CO})$.

With these, the set of all proactive transitions can be defined. These are transitions that are labeled with a proactive event.

$$T_P = \{(so, gu, e, \mathcal{ARG}, AG, ta, ar) \in T | e \in E_P\}$$

Now the functions for the causal relations can formally be defined. First, T_S is formally defined.

Definition 1 $T_S : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of successor transitions. For each successor transition, the function is defined as follows. Let* $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ *and* $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_S(t) = \{t' \in T_P | \forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}} ((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{\textcircled{F}}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}} (s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))))\}$$

So a transition is associated with the T_S function if the event on the label of this transition always has the possibility to be produced after the event that is on the label of the preceding transition. Next, both the associated transition and the preceding transition are always enabled before their corresponding event is being handled. Note that S does not include the $failure$ state. So only preceding transitions are considered for which the postconditions do hold.

Now T_{NS} is defined.

Definition 2 $T_{NS} : T \rightarrow \mathbb{P}(T_P)$: *associates each transition with its no successor transitions. For each no successor transition the function is defined as follows. Let* $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ *and* $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_{NS}(t) = \{t' \in T_P | \forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}} ((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies \neg(\exists_{s'' \in S_{\textcircled{F}}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}} (s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))))\}$$

A transition is associated with the T_{NS} function if the transition is never enabled or the event on its label never has the possibility to be produced after the event that is on the label of the preceding transition.

5.2 Approximated causal relations

Determining all causal relations exactly between all the transitions is expensive or even impossible. The entire IOLTS has to be generated which can become very large in size. Therefore other approaches are used to collect the causal relations. These are however not perfect and only approximate all the causal relations. So not all causal relations are gathered.

If we want to refer to an approximation for which it does not matter which exact gathering approach is used the T^*_S notation is used. These approximations are made statically on the OIL specifications so therefore the approximations are not able to use information from the exact state of the component.

Each transition is associated with four disjoint sets of transitions. The specific transition for which the sets of transitions are associated is referred to as the preceding transition. The associated sets of transitions are mutually exclusive and together contain all proactive transitions of the OIL specification.

- Successor transition function (T^*_S): Transitions returned from T^*_S are proactive transitions for which enough information was found to see that they are always enabled after the preceding transition.
- No successor transition function (T^*_{NS}): Transitions returned from this function are proactive transitions for which enough information was found to see that they are never enabled after the preceding transition.
- Potential successor transition function (T^*_{PS}): Transitions returned from this function are proactive transitions that could be enabled after the preceding transition. Information was found by the gathering approaches that show that there is a good chance that they are enabled. However, no guarantees are given that the transitions are always enabled. The returned transitions are only used to give priority to events labeled with these transitions.
- Unknown transition function (T^*_U): Transitions returned from this function are proactive transitions for which not enough information could be found to say something useful.

The following sets of transitions are gathered for Example 5.2 if we take *r1#1* as the preceding transition. Note that the instance state after the preceding transition has fired has the value *'s2'* for the instance variable *this.state*.

- T^*_S: *p1#1*. This transition is always enabled if the instance variable *this.state* has the value *'s2'*.
- T^*_{NS}: *p1#2*. This transition is only enabled when the instance variable *this.state* has the value *'s4'*. This is never the case directly after the transition *r1*#1 is fired.
- T^*_{PS}/ T^*_U: *p2#1*. Looking at the OIL specification it is not clear if this transition can always fire or not. This completely depends on the value of the instance variable *this.x*. The approximations are done statically so the instance variable *this.x* can not be evaluated. This transition is therefore either associated with T^*_{PS} or T^*_U. This depends on how much priority it should be given.

All proactive transitions that are not associated with the T_S^*, T_{NS}^* or T_{PS}^* functions are associated with the T_U^* function. It is however important that each of the four sets that are associated using the four functions (T_S^*, T_{NS}^*, T_{PS}^*, and T_U^*) have no overlap with each other. This is because different scheduling decisions are made dependent on each set of associated transitions. Overlapping sets could result in events being tried in multiple places while trying them at one place is sufficient. Note that the definitions for T_S and T_{NS} do not overlap, so approximations of these sets do also not overlap.

In the formal definitions of T_S and T_{NS}, all states and transitions were considered from the IOLTS. It could also have been possible to only consider reachable states and transitions. Currently, it is not trivial to take these reachable states and transitions into account while approximating the sets. This is because the exact behavior of an OIL specification has to be computed. Therefore the choice is made to not take into account the reachability of states and transitions. However, it is definitely possible that unreachable transitions are associated using the T_U^* function for some transitions. So limiting the considered transitions to the reachable transitions may definitely still hold some improvements.

The causal relations between the transitions are gathered from the desugared abstract syntax tree, see Chapter 2.3. This tree is generated as part of the code generation process. In this tree, all relevant information is stored in a standardized way. Specific things like guards and checks that compute if an area is active are merged to single expressions when the desugared abstract syntax tree is transformed in the next step to the semantic abstract syntax tree. It is possible to use these expressions to collect the causal relations, however, it adds unnecessary complexity as for example the guards have to be extracted again from these expressions. In the next chapters, multiple approaches for gathering causal relations are described that use the desugared abstract syntax tree.

5.3 Syntax analysis on implied transitions

The first approach for gathering the causal relations uses syntax analysis on the desugared abstract syntax tree. The syntax analysis uses only information found in the two transitions that are compared and does not evaluate any expression. The T_S^{syntax} and T_{NS}^{syntax} function will be used to refer to the function returning successor and no successor transitions found by the gathering approximation using syntax analysis on implied transitions. Next, the T_{PS}^{syntax} function will be used to refer to the function returning potential successor transitions found by the gathering approximation using syntax analysis.

To explain this gathering approach the following OIL specification is used.

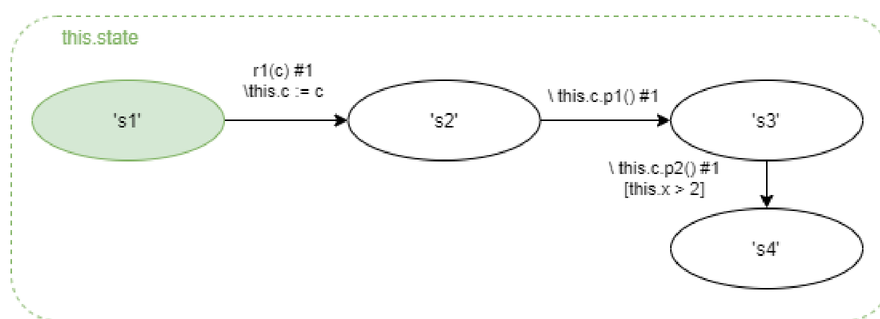

Figure 12: OIL example syntax analysis

Example 5.3 *This OIL specification has three transitions; r1#1, p1#1 and p2#1. Note that the transition p2#1 also has a guard (this.x > 2).*

Successor transitions The causal relations tell something about which transitions are enabled or not after the preceding transition is fired. Information can be used from this preceding transition to detect these causal relations. After the preceding transition is fired we know that the instance state of the OIL specification has specific properties; for example, the target area of the preceding transition is active in this instance state. For a target area to be active its ancestor areas also have to be active. So we can determine a set of areas that are all active after the preceding transition is fired. The acquired active area set can even be expanded. This can be done by looking for scopes that have no invariant or looking for regions. If their parent area is already in the obtained active set they can be added.

There may be transitions for which the source area is active after the preceding transition is fired. If these transitions have no guard and are labeled with an event synchronized with at most one concern they can be associated with the T_S^{syntax} function. This is allowed because in this case the transition is always enabled when the source area is active. By focusing on transitions located in at most one concern it is known that other concerns can not restrict the firing of this transition.

In the desugared abstract syntax tree invariants on scopes and the guard on transitions are always filled with a boolean expression. When no invariant or guard is supplied in the original OIL specification this expression has the value *true*. So we can check for this value to see that a transition has no guard or that we have a scope without invariant. It is certainly possible that a specification had a guard or invariant with the expression *true*. The behavior is exactly the same so this guard or invariant can be considered as if they were not defined. This syntax analysis gathering approach is purely analyzing the syntax of the OIL specification and does not evaluate any expression. An invariant or guard with the expression $\neg false$ is therefore not considered.

In Example 5.3, we take a closer look at the transition *r1#1*. Its target area is the state *'s2'*. So the syntax analysis on implied transitions starts looking for transitions that have this area as its source area. This is the case for the transition *p1#1*. This transition does also not have a guard and is not labeled with a concern; it is therefore identified as a transition associated with the T_S^{syntax} function of the transition *r1#1*.

No successor transitions The syntax analysis on implied transitions detecting successor transitions used the target area of the preceding transition. This target area is active after the preceding transition has fired. For this area to be active there are combinations of regions and states that set instance variables to specific values. These can be extracted to find a list of instance variables and the values that they have in the active target area. This list of instance variables and values can be used to compare to the list of instance variables and their values extracted from other areas. If there is an instance variable that has to have different values to make one of the two areas active we know that these areas can never be active at the same time. When such a mismatch is found for the source area of a transition we know that this transition is never enabled after the preceding transition and it can be associated with the T_{NS}^{syntax} function.

If the preceding transition is fired with another transition labeled with the same event it is possible that this additional transition updates additional instance variables. Note however that the postconditions of the preceding transition have to hold after its firing. So the instance variables that are used to find transitions associated with T_{NS}^{syntax} are never altered to different values.

We again take a closer look at the transition *r1#1* in Example 5.3. The target area of this transition is the state *'s2'*. After the transition *r1#1* is fired this area is active and the instance variable *this.state* has the value *'s2'*. If we take a closer look at the transition *p2#1*, we see that its source area is only active when the instance variable *this.state* has the value *'s3'*. The syntax analysis on implied transitions detects this mismatch in values for the instance variable *this.state* and the transition is identified as a transition associated with the T_{NS}^{syntax} function.

Potential successor transitions The syntax analysis on implied transitions may find transitions for which the source area is active after the preceding transition is fired. However, the found transitions may have a guard or have an event that is located in multiple concerns. These transitions are associated with the T_{PS}^{syntax} function. The syntax analysis on implied transitions is not capable of evaluating the guards or give any guarantees when the transition has to be synchronized over multiple concerns. However, the syntax analysis found the source area to be active so this transition is more likely to be fired than any other transition that does not have this property.

In Example 5.3, we take the transition *p1#1*. Its target area is the state *'s3'*. The syntax analysis on implied transitions looks for transitions that have this area as its source area and finds the transition *p2#1*. However, this transition has a guard which the syntax analysis can not evaluate. So the *p2#1* transition is identified as a transition associated with the T_{PS}^{syntax} function of the transition *p1#1*.

Formal definition In this chapter, the syntax analysis on implied transitions is formally defined. These definitions use the semantics of OIL[4]. Let $(\mathbb{X}, \mathbb{A}, \mathbb{T})$ be an OIL specification. The transition specification and area specification are defined in Chapter 2.2. The transition specification is a tuple $\mathbb{T} = (E, \mathcal{PAR}, T, CO, \mathcal{CO})$. The area specification is also a tuple $\mathbb{A} = (A, \sqsubset, \mathcal{RE}, \mathcal{EXP})$.

Now T_S^{syntax} can be defined.

Definition 3 $T_S^{syntax} : T \to \mathbb{P}(T_P)$ *associates each transition with its set of successor transitions found by the syntax analysis of implied transitions. For each of these transitions, the function is defined as follows.*

$$T_S^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \\ ta \sqsubseteq^* so' \land gu' = true \land \#(\mathcal{CO}(T_{e'})) \leq 1\}$$

So the transitions associated with the T_S^{syntax} function are only located in at most one concern and their guard is *true*. The source area also has to be active after the preceding transition and this is ensured by $ta \sqsubseteq^* so'$. This relation only holds when the so' area is an ancestor area of the ta area.

For defining T_{NS}^{syntax} a helper function is defined; $AS : A \to \mathbb{P}(A_{St})$. The function collects a set of of all ancestor states of an area. These states are found with the $\sqsubseteq^*$ relation.

$$AS(area) = \{state \in A_{St} | area \sqsubseteq^* state\}$$

Now T_{NS}^{syntax} can be defined.

Definition 4 $T_{NS}^{syntax} : T \to \mathbb{P}(T_P)$ *associates each transition with its set of no successor transitions found by the syntax analysis of implied transitions. For each of these transitions, the function is defined as follows.*

$$T_{NS}^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \\ \exists_{state \in AS(ta)} (\exists_{state' \in AS(so')} (\\ \mathcal{RE}(state) = \mathcal{RE}(state') \land \neg(\mathcal{EXP}(state) = \mathcal{EXP}(state'))))\}$$

This definition uses mismatches in values assigned to the same instance variables to find transitions associated with the T_{NS}^{syntax} function. Note that the $\mathcal{EXP}$ function always returns a constant value for a state. A mismatch means that the source area is never active after the preceding transition has fired.

Next, T_{PS}^{syntax} is defined.

Definition 5 $T_{PS}^{syntax} : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of potential successor transitions found by the syntax analysis of implied transitions. For each of these transitions, the function is defined as follows.*

$$T_{PS}^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \\ ta \sqsubseteq^* so' \wedge \neg(gu' = true \wedge \#(\mathcal{CO}(T_{e'})) \leq 1)\}$$

The definition collects transitions in which the source area is implied by the target area of the preceding transition. However, a guard or concern may limit its firing. Therefore it is associated with the T_{PS}^{syntax} function.

The following lemmas show that the gathered transitions are always correct; the associated transitions are correct approximations and do not overlap. The proofs for these lemmas can be found in Appendix A.3.1.

The lemmas for T_S^{syntax} and T_{NS}^{syntax} look as follows.

Lemma 1 *All gathered transitions for T_S^{syntax} by the syntax analysis are located in T_S.* $\forall_{t \in T} T_S(t) \supseteq T_S^{syntax}(t)$.

Lemma 2 *All gathered transitions for T_{NS}^{syntax} by the syntax analysis are located in T_{NS}.* $\forall_{t \in T} T_{NS}(t) \supseteq T_{NS}^{syntax}(t)$

Next, lemmas are supplied that show that there is no overlap with the associated transition of T_{PS}^{syntax}. Earlier it was already stated that there is no overlap between transitions associated with T_S and T_{NS}. So Lemma 1 and Lemma 2 are sufficient to show that there is no overlap between transitions associated with T_S^{syntax} and T_{NS}^{syntax}.

Lemma 3 *The gathered transitions for T_S^{syntax} and T_{PS}^{syntax} do not overlap for the syntax analysis.* $\forall_{t \in T} T_S^{syntax}(t) \cap T_{PS}^{syntax}(t) = \emptyset$

Lemma 4 *The gathered transitions for T_{NS}^{syntax} and T_{PS}^{syntax} do not overlap for the syntax analysis.* $\forall_{t \in T} T_{NS}^{syntax}(t) \cap T_{PS}^{syntax}(t) = \emptyset$

5.4 SMT solving on implied transitions

The next approach for gathering the causal relations is with the help of an SMT solver. T_S^{smt} function will be used to refer to the function returning successor transitions found by the gathering approximation using SMT analysis on implied transitions. Likewise, the T_{NS}^{smt} function will return the no successor transitions found by the gathering approximation using SMT analysis on implied transitions.

The previous causal relation gathering approach which used syntax analysis to find implied transitions could not evaluate expressions. So the existence of these resulted in the transition not being considered for a transition associated with the T_S^{syntax} function. For example, a transition with the guard $\neg$*false* is associated with the T_{PS}^{syntax} function while it does not limit the firing of the transition in any way. The SMT analysis gathering approach aims to fix this by making it possible to evaluate expressions. Evaluating expressions also makes it possible to associate transitions with T_{NS}^{smt}; a transition with the guard *false* can never fire and can safely be associated with T_{NS}^{smt}.

An SMT solver[8][9] is a tool for deciding the satisfiability of certain problems. In such a problem expressions can be defined using boolean types or even types like integers, real numbers, and enums. The SMT solver only states that an SMT problem is satisfiable when there exists a value for each variable that guarantees the expression always holds. An SMT problem is not satisfiable if these values do not exist. So the problem $x > 5 \land true$ where x is an integer variable is satisfiable; an x with the value 10 makes sure this expression always holds. However, an expression like $x > 5 \land false$ where x is an integer variable is not satisfiable. No integer value for x could result in this expression being true.

If the SMT problems are formatted in a certain way the SMT solver can be used to see if assumptions imply that assertions hold. This looks as follows; $Assumptions \land \neg(Assertions)$. When such an SMT problem is unsatisfiable it means that the assumptions imply the assertions. So a problem with the assumption $x = 10$ and assertion $x > 5$ will results in the following SMT problem $x = 10 \land \neg(x > 5)$. This SMT problem is unsatisfiable as either $x = 10$ or $\neg(x > 5)$ is true, they are never simultaneously true. So we can conclude that $x = 10$ implies $x > 5$.

The preceding transition and the transition that is considered for a causal relation can be used to generate satisfiability problems. Conclusions can be drawn when such a problem is found satisfiable or not by the SMT solver. The big advantage of using the SMT solver is that expressions can now be taken into account. Ideally, this would result in more transitions being gathered for T_S^{smt} and T_{NS}^{smt}. The choice is made to initially limit the causal relation gathering approach of T_S^{smt} to transitions that are labeled with events that are in at most one concern. This restriction is introduced to simplify the SMT analysis and is revisited in Chapter 5.5.

If the preceding transition is fired with another transition labeled with the same event it is possible that this additional transition updates additional instance variables. Note however that the postconditions of the preceding transition have to hold after its firing. So the instance variables that are used to find transitions associated with T_{NS}^{smt} are never altered to different values.

In the implementation, Z3[8] is used to solve the satisfiability problems. The SMTlib[10] standard is used for all these problems. This is a standard used by all big SMT solver implementations that state how SMT problems can be formatted. So adding support for other solvers should be relatively easy.

To explain the generation of the satisfiability problems the following OIL specification is used.

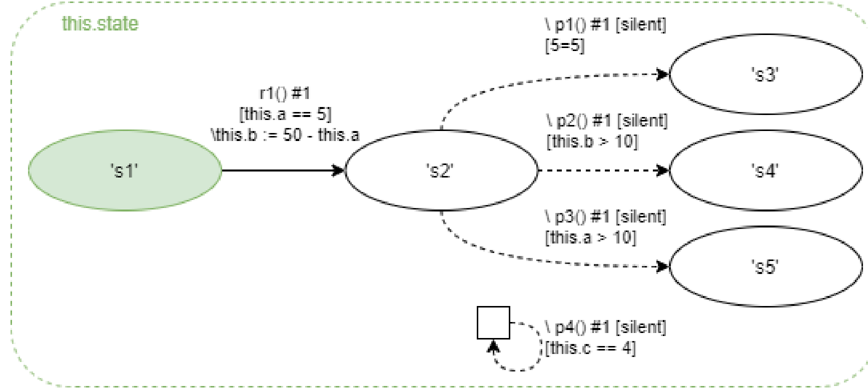

Figure 13: OIL example SMT analysis

Example 5.4 *This OIL specification has a transition with a reactive event; r1#1. This transition has a guard ($this.a == 5$) and an assignment ($this.b := 50 - this.a$). Next, a few transitions with silent proactive events are visible. Note, that all these transitions have a guard.*

Active target area and assertion After the preceding transition is fired we know that the target area of this transition is active and that its assertion holds. These properties can be used to show that a considered transition can be associated with T_S^{smt} or T_{NS}^{smt}. To see if a transition is associated with T_S^{smt} the solver needs to show that the source area is always active and the guard does always hold in the instance state after the preceding transition is fired. A similar check can be used to see if a transition is associated with T_{NS}^{smt}. In this case, the source area has to be never active or the guard has to never hold in the instance state after the preceding transition is fired.

The active target area and assertion of the preceding transition can directly be used to see if the source area of the evaluated transition is active and if its guard holds. All this information is from the same instance state. The '.' in the instance variable names is replaced with a '_' character to be compatible with the SMTlib standard.

Looking at the example the following assumptions and assertions are generated from the transitions *r1#1* and *p1#1*. These are used to check if *p1* is associated with the T_S^{smt} function of *r1*.

- Assumptions: $this_state = 2$
- Assertions: $this_state = 2 \wedge 5 = 5$

Looking at these assumptions and assertions it is clear that the assertions are implied by the assumptions. So the transition labeled with *p1* is associated with the T_S^{smt} function of the transition labeled with *r1*.

This approach can be used as-is for checking transitions associated with T_S^{smt}. To check if the evaluated transition is associated with T_{NS}^{smt} the assertions are negated. This gives a satisfiability problem that checks if the assertions never hold.

Using the SMT solver in this way is close to the syntax analysis finding implied transitions. It mainly focuses on showing that the source area is active and the guard holds of the evaluated transition. It however should find more causal relations as it can interpret the expressions found in the guard and scopes.

Assignments, active source area, and guard Next, the instance state of the OIL specification before the preceding transition was fired is also used, this instance state is referred to as the old instance state. The instance state after the preceding transition is referred to as the present instance state. The two instance states are connected via the assignments that the preceding transition can contain.

To be able to use information from the old instance state, additional variables are used. Each instance variable is now used twice; one referring to the old instance state and one referring to the present instance state. The variables that are defined for the old instance state are prefixed with *old_*.

Assignments give values to variables from the present instance state but may use values from the old instance state. This behavior is visible in the assignment of the transition *r1#1*; $this.b := 50 - this.a$. This assignment can be used to add the following assumption to the SMT problem $this_b = 50 - old_this_a$. The added variables for the old instance state can also be used to add information from the guard and the knowledge that the source area was active before the preceding transition was fired.

The generated assumptions and assertions can be seen below that check if the transition *p2#1* is associated with T_S^{smt} or T_{NS}^{smt} of the transition *r1#1*.

- Assumptions: $this_state = 2 \wedge old_this_state = 1 \wedge this_b = 50 - old_this_a \wedge old_this_a = 5$
- Assertions: $this_state = 2 \wedge this_b > 10$

So now the solver can use this information to find that old_this_a has to be 5 as a result of the guard of the transition *r1#1*. This information can further show that $this_b$ has to be 45 which is enough information to see that the guard of the transition *p2#1* always holds and that the transition *p2#1* is associated with T_S^{smt} of the transition *r1#1*.

Reactive transitions There is an additional source of information that could be used to expand the number of assumptions. Earlier it was stated that OIL specifications have a so-called run-to-completion semantics; reactive events are only handled when no proactive event can be produced. So, if a reactive transition is being fired no proactive transition could be fired in that instance state. Now it is known that either the source area of the evaluated transition was not active in this instance state or the guard did not hold in it.

The satisfiability problem generated for reactive transitions can have additional assumptions; the negation of both the guard and the source area being active of the evaluated transition on the old variables. Adding these assumptions means that the proactive transition could not fire in the old instance state. These assumptions could be added for all proactive transitions in the OIL specification however this will greatly increase the size of the satisfiability problems while the information from the proactive transition that is evaluated has most likely the most relevant information. Therefore the assumptions are only introduced for the evaluated transition.

So in Example 5.4, the assumptions and assertions generated to see if the transition *p4#1* is associated with either T_S^{smt} or T_{NS}^{smt} of the transition *r1#1* looks as follows.

- Assumptions: $this_state = 2 \wedge old_this_state = 1 \wedge this_b = 50 - old_this_a \wedge old_this_a = 5 \wedge \neg(old_this_c = 4)$
- Assertions: $this_c = 4$

The $\neg(old_this_c = 4)$ assumption is added to the generated satisfiability problem. However, this additional assumption does not result in enough information to show any additional causal relation.

More updates Currently, there are no restrictions on how many transitions can fire at the same time as the preceding transition. However, this next approach is only allowed when the preceding transition only fires on its own.

It is possible to detect instance variables that are not altered by the firing of a transition. This information can be used to add assumptions that set the value of these present instance variables to the same value as the old instance variables.

These added assumptions do complement the added assumptions from the reactive transitions. For example, there can be reactive transitions that do not assign any new values to instance variables. So, no proactive transition can be fired in the instance state before this transition as it was labeled with a reactive event and because it does not alter any instance variables the solver can find that there are also no proactive transitions that can be fired after it.

So looking at Example 5.4, the assumptions and assertions generated to see if the transition *p4#1* is associated with T_S^{smt} or T_{NS}^{smt} of the transition *r1#1* now look as follows.

- Assumptions: $this_state = 2 \wedge this_a = old_this_a \wedge this_b = 50 - old_this_a \wedge this_c = old_this_c \wedge old_this_a = 5 \wedge \neg(old_this_c = 4)$
- Assertions: $this_c = 4$

Because the instance variable c is never changed by the transition *r1#1* the solver now finds that the transition *p4#1* is associated with T_{NS}^{smt} for the *r1#1* transition.

Unfortunately, the benefits of these added assumptions are limited. This is the result of the restriction that the preceding transition can only fire on its own that has to be introduced.

It is however also possible to take into account all transitions that the preceding transition can fire with at the same time. The detection that finds which instance variables are not altered becomes a bit trickier as other transitions can also update instance variables. Taking all these transitions into account increases the generated satisfiability problem significantly while only finding new causal relations in very specific situations, see Chapter 9 for the exact results. A problem with all assumptions takes around 1.6 times longer to solve than a problem without the assumption that specific instance variables are not altered. The reduced effectiveness is mostly the result of that multiple combinations of transitions can fire together with the preceding transition. All possible combinations of enabled transitions have to be considered to show that a specific instance variable is not assigned a new value.

During testing, no example is found for which valuable information survived these additional assumptions or the restrictions that had to be introduced otherwise. Therefore the assumptions for not altered instance variables are not added in the current implementation. This results in smaller problems that are solved faster.

Formal definition The SMT analysis on implied transitions is formally defined.

Let $(\mathbb{X}, \mathbb{A}, \mathbb{T})$ be an OIL specification. The transition specification and the $\mathcal{AC}$ function were defined earlier in Chapter 2.2. The transition specification is a tuple $\mathbb{T} = (E, \mathcal{PAR}, T, CO, \mathcal{CO})$. and the $\mathcal{AC}$ function associates each area with a boolean expression which can be used to evaluate if that area is active in an instance state. Note that X is the set of all variables, and EXP_X is the set of all expressions using the variables from X.

To be able to define the SMT analysis the $\sigma_{old} : EXP_X \rightarrow EXP_X$ function is defined first. This function is used to substitute the variables an expression contains to alternate variables so that they can be used to refer to the instance state before the preceding transition. The definition looks as follows

$$\sigma_{old}(c) = c$$
$$\sigma_{old}(x) = x^{old}$$
$$\sigma_{old}(op(exp_1, ..., exp_n)) = op(\sigma_{old}(exp_1), ..., \sigma_{old}(exp_n))$$

Where c is a constant, $c \in \mathbb{V}$, $x \in X$ is a variable and op is an n-ary operator where $n > 0$ and $op : \mathbb{V}^n \rightarrow \mathbb{V}$.

Next, $Assign : Assignments \rightarrow EXP_X$ converts a set of assignments in the form $x := f$ where $x \in X$ and $f \in EXP_X$ to an expression in the following way.

$$Assign(assignments) = \bigwedge_{(x:=f) \in assignments} (x = \sigma_{old}(f))$$

Note that the assignments use variables from the old instance state. Next, two functions are supplied that represent the SMT problems for finding the causal relations. The SMT problem looks as follows for T_S^{smt} indicated with $\varphi_S : T \times T_P \rightarrow EXP_X$.

$$\varphi_S((so, gu, e, \mathcal{ARG}, AG, ta, ar), (so', gu', e', \mathcal{ARG}', AG', ta', ar')) =$$
$$\mathcal{AC}(ta) \wedge ar \wedge \sigma_{old}(\mathcal{AC}(so) \wedge gu) \wedge Assign(AG) \wedge \neg(\mathcal{AC}(so') \wedge gu')$$

A similar function is defined for T_{NS}^{smt} indicated with $\varphi_{NS} : T \times T_P \rightarrow EXP_X$.

$$\varphi_{NS}((so, gu, e, \mathcal{ARG}, AG, ta, ar), (so', gu', e', \mathcal{ARG}', AG', ta', ar')) =$$
$$\mathcal{AC}(ta) \wedge ar \wedge \sigma_{old}(\mathcal{AC}(so) \wedge gu) \wedge Assign(AG) \wedge (\mathcal{AC}(so') \wedge gu')$$

The only difference is the removed negation around $(\mathcal{AC}(so') \wedge gu')$ to find transitions that are never implied.

Now, these functions can be used to give the definitions of the different analyses. First, T_S^{smt} is defined.

Definition 6 $T_S^{smt} : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of successor transitions found by the SMT analysis of implied transitions. For each of these transitions, the function is defined as follows. Let* $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ *and* $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_S^{smt}(t) = \{t' \in T_P | \varphi_S(t, t') \textit{is unsatisfiable} \wedge \#(\mathcal{CO}(T_{e'})) \leq 1\}$$

The satisfiability problem is placed in conjunction with a check that makes sure the transitions associated by T_S^{smt} are labeled with at most one concern.

Next, T_{NS}^{smt} is defined.

Definition 7 $T_{NS}^{smt} : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of no successor transitions found by the SMT analysis of implied transitions. For each of these transitions, the function is defined as follows. Let* $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ *and* $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_{NS}^{smt}(t) = \{t' \in T_P | \varphi_{NS}(t, t') \textit{is unsatisfiable}\}$$

In this definition the satisfiability problem φ_{NS} can be found.

The following lemmas are created that are proven to show that the gathered transitions for the SMT analysis are always correct; are the associated transitions always approximations. The proofs for these lemmas can be found in Appendix A.3.2.

Lemma 5 *All gathered transitions for* T_S^{smt} *by the SMT analysis are located in* T_S.
$\forall_{t \in T} T_S(t) \supseteq T_S^{smt}(t)$

Lemma 6 *All gathered transitions for* T_{NS}^{smt} *by the SMT analysis are located in* T_{NS}.
$\forall_{t \in T} T_{NS}(t) \supseteq T_{NS}^{smt}(t)$

Earlier it was already stated that there is no overlap between transitions associated with T_S and T_{NS}. So Lemma 5 and Lemma 6 are sufficient to show that there is no overlap between transitions associated with T_S^{smt} and T_{NS}^{smt}.

5.5 Multiple concerns

Until now we only considered transitions labeled with events that are located in at most one concern for transitions associated with T^*_S. However, this greatly reduces the number of causal relations that can be found as concerns are often used to model different aspects of an OIL specification separately and concisely. So a gathering approach that does consider transitions labeled with events that are located in multiple concerns is certainly beneficial. The T^{co}_S function will be used to refer to the function returning successor transitions found by the gathering approximation using concern analysis. Likewise, the T^{co}_{NS} function will return the no successor transitions found by the gathering approximation using concern analysis.

To accomplish this, we drop the restriction that the events of transitions have to be located in at most one concern for both the syntax and SMT analysis on implied transitions. So now these gathering approaches can collect transitions that have events located in multiple concerns. Synchronization between transitions over multiple concerns is not considered so it is now possible for the syntax and SMT analysis on implied transitions to find transitions that do not have the properties of a transition associated with T_S. The function $T^{*'}_S$ refers to either the already gathered T^{syntax}_S or T^{smt}_S without the concern restriction.

A post-analysis is introduced that makes sure that after it completes only transitions are gathered that have the properties of T_S. It can also be used to associate transitions with T_{NS}. This post-analysis can be introduced directly after both the syntax and SMT analysis on implied transitions. It can however also be used once both have been completed as neither the syntax nor SMT analysis on implied transitions depend on already gathered transitions associated with T^*_S or T^*_{NS}. This post-analysis consists of two parts; one part looks at transitions associated with T^{co}_S and another looks at transitions associated with T^{co}_{NS}. Both mimic the behavior that can be found in the concern condition discussed in Chapter 2.2.

$$\mathcal{CC}(e) = \bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t \in T_{e,c}} \mathcal{PRC}(t)$$

To explain the concern analysis in more detail the following OIL specification is used.

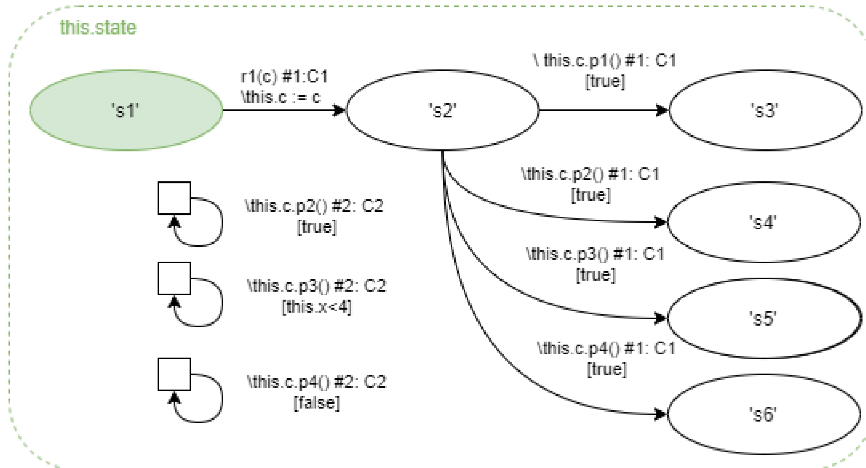

Figure 14: OIL example concern analysis

Example 5.5 *The OIL specification in Figure 14 has one transition labeled with a reactive event and seven transitions labeled with proactive events. Note that the events p2, p3 and p4 are defined in multiple concerns. For this example, it is assumed that all transitions which have the guard true have been identified as a transition associated with* $T_S^{*'}$ *of r1#1. The transition that is guarded with the guard false is identified as a transition associated with the* T_{NS}^* *function of r1#1. There is one transition p3#2 with a different guard and this transition is associated with* $T_{PS}^{*'}$ *of r1#1.*

Successor transitions The concern analysis starts by taking a preceding transition and selecting a transition associated with it using $T_S^{*'}$. The concern analysis for T_S^{co} collects all concerns that have a transition labeled with the same event as the selected transition from $T_S^{*'}$. Next, it collects all transitions in each concern that are labeled with the same event which could be synchronized with the selected transition from $T_S^{*'}$. Now, the concern analysis checks if for each concern there is at least one transition that is associated with $T_S^{*'}$ of the preceding transition. If this is the case, the transition can be associated with T_S^{co} of the preceding transition. However, if this is not the case the transition is associated with T_{PS}^{co} of the preceding transition. There is a concern for which the concern analysis can not guarantee that there is an enabled transition labeled with the same event as the selected transition from $T_S^{*'}$. This in turn means the concern analysis can not guarantee that the transition is always enabled.

In Example 5.5, the transition *p1#1* has been identified as a transition associated with the $T_S^{*'}$ function of *r1#1*. Notice that the event *p1* only occurs once and is only located in one concern. So the concern analysis concludes that each concern has at least one transition associated with the $T_S^{*'}$ function. So the *p1#1* transition is identified as a transition associated with the T_S^{co} function for the *r1#1* transition. Note that this is exactly the same behavior as we would have had if we kept the restriction on the syntax and SMT analysis to only gather transitions labeled with events located in at most one concern.

Next, a closer look is taken at *p2#1*. This is a transition associated with the $T_S^{*'}$ function for the preceding transition *r1#1* but notice that the event *p2* also occurs in another concern. This is however no problem because both concerns have a transition that is associated with the $T_S^{*'}$ function of *r1#1*. So the *p2#1* transition is associated with T_S^{co} for the *r1#1* transition.

The *p3#1* transition was found to be a transition associated with the $T_S^{*'}$ for the preceding transition *r1#1* by the syntax analysis on implied transitions. Note that the event *p3* occurs in another concern. However, for this *C2* concern, no transition is associated with $T_S^{*'}$. So this concern limits the production of the event *p3*. This means that the transition *p3#1* is associated with the T_{PS}^{co} function for the *r1#1* transition.

No successor transitions Next, the dual of the concern analysis for transitions associated with T_S^{co} will be used to gather transitions associated with T_{NS}^{co}. The concern analysis takes a preceding transition and selects a transition associated with it using T_{NS}^*. The concern analysis for T_{NS}^{co} collects all concerns that have a transition labeled with the same event as the selected transition from T_{NS}^*. Next, it collects all transitions in each concern that are labeled with the same event which could be synchronized with the selected transition from T_{NS}^*.

Now the concern analysis checks if there is a concern for which all transitions are located in T_{NS}^* of the preceding transition. If this is the case we may associate the selected transition to T_{NS}^{co} of the preceding transition. There is a concern that can never fire a transition that is labeled with the same event as the selected transition. So this selected transition can therefore also never fire.

We take a closer look at the transition *p4#1*. The event *p4* is found in the concern *C1* and *C2*. The concern analysis detects that all transitions labeled with this event in the $C2$ concern are associated with the T_{NS}^* function of the *r1#1* transition. Meaning that the *p4* event can never be produced after the transition *r1#1*. So now *p4#1* is associated with T_{NS}^{co} of the *r1#1* transition.

Formal definition The concern analysis is formally defined. Let $(\mathbb{X}, \mathbb{A}, \mathbb{T})$ be an OIL specification. The transition specification was defined earlier in Chapter 2.2 as a tuple that looks as follows $\mathbb{T} = (E, \mathcal{PAR}, T, CO, \mathcal{CO})$.

The formal definition of this concern analysis consists of two parts. First, the formal definitions of both the syntax analysis and SMT analysis are used but now with the removal of the $\#(\mathcal{CO}(T_{e'})) \leq 1$ restriction. These look as follows. Let $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ and $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_S^{syntax'}(t) = \{t' \in T_P | ta \sqsubseteq^* so' \wedge gu' = true\}$$

$$T_S^{smt'}(t) = \{t' \in T_P | (\varphi_S(t, t') is\ unsatisfiable)\}$$

These sets are then used in the following definitions.

Definition 8 $T_S^{co} : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of successor transitions found by the concern analysis. For each of these transitions, the function is defined as follows.*

$$T_S^{co}(t) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \forall_{c \in \mathcal{CO}(T_{e'})}(\exists_{t' \in T_{e',c}}(t' \in T_{NS}^{*'}(t)))\}$$

Definition 9 $T_{NS}^{co} : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of no successor transitions found by the concern analysis. For each of these transitions, the function is defined as follows.*

$$T_{NS}^{co}(t) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \exists_{c \in \mathcal{CO}(T_{e'})}(\forall_{t' \in T_{e',c}}(t' \in T_{NS}^{*}(t)))\}$$

Definition 10 $T_{PS}^{co} : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of potential successor transitions found by the concern analysis. For each of these transitions, the function is defined as follows.*

$$T_{PS}^{co}(t) = T_S^*(t) \setminus (T_S^{co}(t) \cup T_{NS}^{co}(t))$$

These definitions are used in the following lemmas that are proven to make sure the gathered transitions are correct; are the associated transitions always approximations. The shown lemmas are proven in Appendix A.3.3.

The lemmas look as follows for the transitions associated with T_S^{co} and T_{NS}^{co}.

Lemma 7 *All gathered transitions for T_S^{co} by the concern analysis are located in T_S.*
$\forall_{t \in T} T_S(t) \supseteq T_S^{co}(t)$

Lemma 8 *All gathered transitions for T_{NS}^{co} by the concern analysis are located in T_{NS}.*
$\forall_{t \in T} T_{NS}(t) \supseteq T_{NS}^{co}(t)$

Next, two lemmas make sure there is no overlap with transitions associated with T_{PS}^{co}. Earlier it was already stated that there is no overlap between transitions associated with T_S and T_{NS}. So Lemma 7 and Lemma 8 are sufficient to show that there is no overlap between transitions associated with T_S^{co} and T_{NS}^{co}.

Lemma 9 *The gathered transitions for T_S^{co} and T_{PS}^{co} do not overlap for the concern analysis.*
$\forall_{t \in T} T_S^{co}(t) \cap T_{NS}^{co}(t) = \emptyset$

Lemma 10 *The gathered transitions for T_S^{co} and T_{PS}^{co} do not overlap for the concern analysis.*
$\forall_{t \in T} T_{NS}^{co}(t) \cap T_{PS}^{co}(t) = \emptyset$

5.6 Unstable areas

The previous causal relation gathering approaches struggle when the transitions are located in separate parts of OIL specifications; for example when an OIL specification has multiple regions that can be active at the same time. In this chapter, a causal relation gathering approach is discussed which uses the already obtained causal relations together with the OIL specification to try to see if additional causal relations can be found. This gathering approach makes it possible to collect causal relations for transitions that are located in separate parts of OIL specifications. The $T_{NS}^{unstable}$ function will be used to refer to the function returning no successor transitions found by the gathering approximation using unstable area analysis

To aid this explanation the following OIL specification is used.

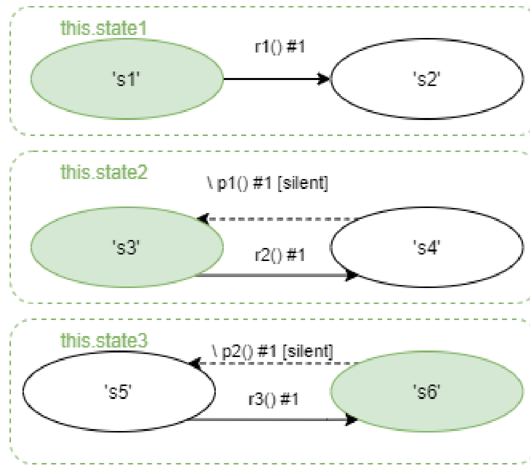

Figure 15: OIL example unstable area analysis

Example 5.6 *The OIL specification in Figure 15 has three regions each with an active state; 's1', 's3', and 's6'. In this example these states are also the initial states of this OIL specification. Next, there are three transitions labeled with a reactive event and two transitions labeled with a proactive event. The proactive transition p1#1 is identified as a transition associated with* T_S^* *of r2#1 and the transition p2#1 is identified as a transition associated with* T_S^* *of the transition r3#1. Besides these two causal relations no other causal relations are found. So both the transitions p1#1 and p2#1 are associated with* T_U^* *of r1#1.*

Unstable areas The unstable area analysis looks for unstable areas. These are areas for which there are always transitions enabled that leave this area. If these unstable areas are detected properly they can be used to acquire additional no successor transitions for $T_{NS}^{unstable}$.

An unstable area has the following properties. The area has at least one incoming and one outgoing transition. The outgoing transition has to be labeled with a proactive event. At least one of the outgoing transitions has to be associated with T_S^* for all the incoming transitions. So if this area is entered via one of the incoming transitions it is known from an earlier analysis that this area can always be left immediately with one of the outgoing transitions. However, there are a few nuances we have to cover. The unstable area has to be a state that is not an initial state. Initial states are unwanted because we can not depend on the incoming transitions to detect if an outgoing transition is enabled. Next, an unstable area can not be the parent area of any other area. This property is currently enforced to be certain that the unstable areas have the correct behavior as edge cases like an outgoing transition connecting to a nested area of the unstable area are removed. It may be correct to drop the requirement that an unstable area can not be a parent area but this is currently not known.

If there are transitions that leave an unstable area we know that the transitions in T_S^* capture the behavior of these transitions. This is the case even if the transition is never fired. In this case, it is never needed to fire this transition as other transitions are always enabled and are fired with priority. When all transitions of an event over multiple concerns all leave such an unstable area it is allowed to say that T_S^* captures the behavior of the complete event. So if transitions in this event are still associated with T_U^* they can be associated with $T_{NS}^{unstable}$.

In Example 5.6, the syntax and SMT analysis on implied transition would not have found any causal relation between the transitions *r1#1* and *p1#1*. However, the state *'s4'* is an unstable area. Whenever *r2#1* is fired the area is always left immediately by the firing of *p1#1* because there is only one transition labeled with the event *p1* it means the event *p1* is completely accounted for by the transitions associated with T_S^*. So all transitions labeled with the event *p1* can be associated with $T_{NS}^{unstable}$ of both the transition *r1#1* and *r3#1* as these had this transition associated with T_U^*. State *'s6'* is an initial state so not an unstable area. Therefore the unstable area analysis can not be used to draw conclusions for causal relations between the transitions *r1#1* and *p2#1*.

Formal definition The unstable area analysis is formally defined. However, no lemmas are formulated using this definition in this report.

Let $(\mathbb{X}, \mathbb{A}, \mathbb{T})$ be an OIL specification. The transition specification, area specification and variable specification are defined in Chapter 2.2. The transition specification is a tuple $\mathbb{T} = (E, \mathcal{PAR}, T, CO, \mathcal{CO})$. The area specification is a tuple $\mathbb{A} = (A, \sqsubset, \mathcal{RE}, \mathcal{EXP})$ and the variable specification is also a tuple $\mathbb{X} = (X, \mathcal{I})$.

The following two functions are used to return a set of transitions that enter or leave a specific area. First, $In : A \rightarrow \mathbb{P}(T)$ associates an area with all the transitions that have this area as the target area.

$$In(area) = \{(so, gu, e, \mathcal{ARG}, AG, ta, ar) \in T | ta = area\}$$

Next, $Out : A \rightarrow \mathbb{P}(T)$ associates an area with all the transitions that have this area as source area.

$$Out(area) = \{(so, gu, e, \mathcal{ARG}, AG, ta, ar) \in T | so = area\}$$

The Out function also associates a set of areas with the set of transitions that have a source area in this area set; $Out : \mathbb{P}(A) \rightarrow \mathbb{P}(T)$.

$$Out(areas) = \bigcup_{area \in areas} Out(area)$$

The set of unstable areas is defined as follows.

$$\begin{aligned} A_U = \{area \in A_{St} | \neg \exists_{a \in A}(a \sqsubset area) \\ \wedge In(area) \neq \emptyset \wedge \forall_{t \in In(area)}((T_S^*(t) \cap Out(area)) \neq \emptyset) \wedge \neg(\mathcal{EXP}(area) \in \mathcal{I})\} \end{aligned}$$

So an unstable area is not a parent area of any area. It also has at least one incoming transition. All incoming transitions do have at least one transition associated with the T_S^* function that is an outgoing transition of that same area. So when an incoming transition enters this area there is an outgoing transition that leaves the same area that can always be fired. The last thing that is checked is that the area can not be an initial state. The $\mathcal{EXP}$ function returns a constant value when a state is inputted. When this constant value is in the set of initial values we know that this state is an initial state.

Using these unstable areas a definition can be supplied for $T_{NS}^{unstable}$.

Definition 11 $T_{NS}^{unstable} : T \rightarrow \mathbb{P}(T_P)$ *associates each transition with its set of no successor transitions found by the unstable area analysis. For each of these transitions, the function is defined as follows. Let* $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ *and* $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_{NS}^{unstable}(t) = \{t' \in T_P | \forall_{c' \in (\mathcal{CO}(T_{e'}) \setminus \mathcal{CO}(t'))}(Out(A_U) \supseteq T_{e',c'}) \land t' \in Out(A_U) \land \neg(t' \in T_S^*(t)\}$$

A transition can be associated with $T_{NS}^{unstable}$ if that transition is not associated with T_S^* already and it leaves an unstable area. It is also important that all transitions from other concerns that can be synchronized with the same event do leave such an unstable area.

5.7 Combining gathering approaches

The gathering approaches using syntax and SMT analysis for gathering implied transitions can be used on their own. However, the approach using the SMT solver is eight times slower while it finds more causal relations in the end, see Chapter 9 for exact numbers. Therefore the choice is made to combine the two gathering approaches. First, the syntax analysis finds some causal relations, and afterward, the SMT solver is used to try to find even more causal relations. The SMT analysis on implied transitions only generates problems for transitions that are not associated with T_S^* or T_{NS}^*. In this way, the SMT solver is only used when the syntax analysis can not say something useful. For example, this is the case when a transition has a guard.

Both syntax and SMT analysis on implied transitions ignored the restriction for transitions having at most one concern. So after these two analyses, the concern analysis takes place to collect the transitions that do conform to multiple concerns. This concern analysis can be used after either the syntax or SMT analysis on implied transitions has finished however it is also fine to only use it once after both have finished. This is allowed because neither analysis does depend on already gathered transitions associated with T_S^* or T_{NS}^*.

At last, the unstable area analysis takes place. This gathering approach needs transitions associated with T_S^* and is put last to ensure the concern analysis is finished.

6 Scheduling improvement strategies with gathered causal relations

In the previous chapter, approaches were discussed for gathering causal relations that can be used to make more informed scheduling decisions. In this chapter, the scheduling strategies are listed that were found that use the gathered causal relations to make more informed decisions. The scheduling strategies always produce a valid schedule even in the case no causal relations are gathered. The scheduling strategies also do not alter the OIL specifications in any way, they only try to improve the scheduling itself.

6.1 Improved scheduling complementing naive OIL run loop scheduler

The naive OIL run loop scheduler is used to ensure the run-to-completion semantics, see Chapter 2.3. However, causal relations are now gathered which can be used to improve the scheduling. In this chapter, a few scheduling strategies are listed that can be used to complement the naive OIL run loop scheduler. The naive OIL run loop scheduler is still used to fall back to after these scheduling strategies are completed.

To aid this explanation a small OIL specification is used.

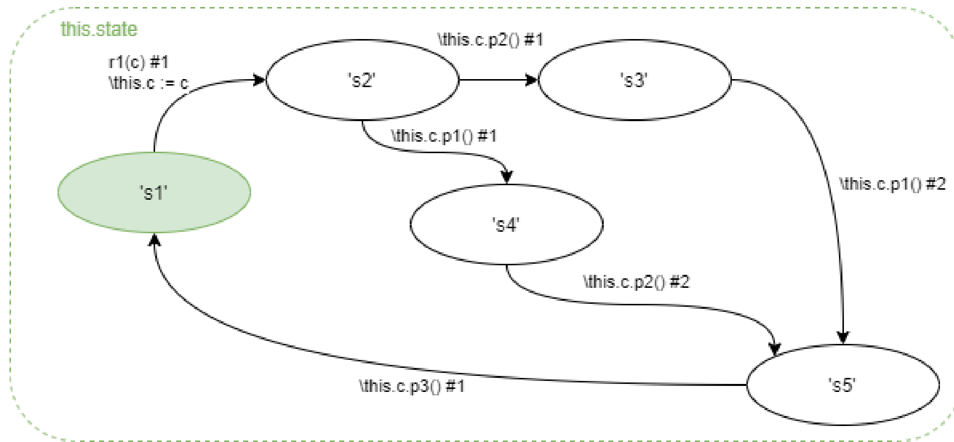

Figure 16: OIL example improved scheduling complementing current OIL run loop scheduler

Example 6.1 *The OIL specification in Figure 16 has one reactive event; r1. Next it also has three proactive events p1, p2 and p3. For this example, all causal relations are correctly identified and all proactive transitions are either associated with T^*_S or T^*_{NS}. So p2#1 and p1#1 are associated with T^*_S of r1#1. Next, p1#2 is associated with T^*_S of p2#1 and p2#2 is associated with T^*_S of p1#1. The transition p3#1 is associated with T^*_S of both p1#2 and p2#2.*

Scheduling successor events When an event is handled the generated code determines which of the transitions of this event can fire and checks if each concern has an enabled transition. If this is the case, it executes the updates of each enabled transition, and afterward the naive OIL run loop scheduler is used for the run-to-completion semantics, see Chapter 2.3.

The enabled transitions from the handled event can be used to see if one of the transitions has any transitions associated with T^*_S. The events from these associated transitions can always be produced so one of them can be chosen arbitrarily and tried. This arbitrary choice is allowed because of the confluent proactivity property of valid OIL specifications, see Chapter 2.1. It is again known which transitions are enabled during the production of the tried event so T^*_S can be used again to continue chaining proactive events. At some point, T^*_S will not associate any

transitions for any of the enabled transitions in an event. Now the naive OIL run loop scheduler can be used to still ensure the run-to-completion semantics.

Scheduling these successor events is accomplished by adding the try method call of the event at the end of the method that handles the event of the preceding transition. These added try method calls are surrounded by if statements ensuring that they are only called when the appropriate preceding transition was enabled. The insertion of the try method call according to transitions associated with T_S^* can be done for each generated method; both reactive and proactive events. When this is done, it also indirectly means that events are chained.

In Example 6.1 it is visible that after the handling of the *r1* event it is possible to produce the following events; *p2*, *p1* followed by *p3*. The code generated for the event *r1* and *p2* can be found in Algorithm 11 and 12. The generated code for event *r1* has a $TRY_EVENT_p2()$ method call. So first this event is tried before the scheduler falls back to the naive OIL run loop. This event comes from a transition associated with T_S^* so it is known that this event can always be produced. Now we take a closer look at the if statements surrounding the try method calls, the event *p2* has two transitions; one from the state *4* to *5* and one from the state *2* to *3*. So here two try event method calls are added as each transition has its own transitions associated with T_S^*. One try method call is called when the transition leaving the state *4* is enabled calling $TRY_EVENT_p3()$. The other try method call is called when the transition leaving the state *2* is enabled calling $TRY_EVENT_p1()$. Only after the two added try event method calls have been completed the naive OIL run loop scheduler is called.

Algorithm 11: Pseudocode for r1

```
1  Function r1:
2      _t_enabled1 := state = 1
3
4      /* Checking precondition r1                                    */
5      ...
6      /* Update instance variables r1                                */
7      ...
8      /* Checking postcondition r1                                   */
9      ...
10     /* Handled event r1                                            */
11
12     if _t_enabled1 then
13         TRY_EVENT_P2()// Added successor event
14
15     _OIL_RUN()
```

Algorithm 12: Pseudocode for p2

```
1  Function TRY_EVENT_P2:
2      _t_enabled1 := state = 2
3      _t_enabled2 := state = 4
4
5      _oil_e_enabled := _t_enabled1 or _t_enabled2
6
7      /* Checking precondition p2                                    */
8      if _oil_e_enabled then
9          /* Update instance variables p2                             */
10         ...
11         /* Checking postcondition p2                                */
12         ...
13         /* Produced event p2                                        */
14
15         if _t_enabled1 then
16             TRY_EVENT_P1() // Added successor event
17
18         if _t_enabled2 then
19             TRY_EVENT_P3() // Added successor event
20
21
22     return _oil_e_enabled
```

Remove OIL run loop scheduler call It is known which transition is fired last before the naive OIL run loop scheduler is used. This could be the result of the original reactive event that the component received or the last scheduled successor event, see the last section. When all proactive transitions are associated with T^*_{NS} for this last fired transition it is known that no proactive transition is enabled after this specific transition. This means that also no proactive event can be produced. So, there is no need to fall back on the naive OIL run loop scheduler if only this transition fires.

The *_OIL_RUN* method call is currently added at the end of the handling of a reactive event. An if statement is used to surround the *_OIL_RUN* method call to ensure that it is only called for the transitions that do not have all proactive transitions associated with T^*_{NS}. If no transitions labeled with the same event need the *_OIL_RUN* method call it is not added. It is also possible that all transitions need the naive OIL run loop scheduler which results in the *_OIL_RUN* method call just being added similar to the original implementation.

This technique can be used as-is, but it could also be enhanced further. This is done by combining this technique with the scheduling strategy that schedules successor events, see the last section. It is possible to follow the chain of scheduled transitions associated with T^*_S and determine the last transition that results in the scheduling of an event. This transition can then be used to determine if the naive OIL run loop scheduler is needed by checking if all proactive transitions are associated with T^*_{NS}. Note that it is possible that multiple transitions fire at the same time and enable different behavior therefore the detection of these chains is currently limited to transitions that fire on their own.

So in the example, the events *p2*, *p1*, and *p3* are chained by the scheduled successor events. So the last produced event for the event *r1* is *p3*. The last fired transition labeled with this event *p3#1* has no proactive transitions that are enabled after it; all proactive transitions are associated with T^*_{NS} for *p3#1*. Meaning that the *_OIL_RUN* method call can safely be removed.

The generated method for the *r1* event looks as follows.

Algorithm 13: Pseudocode for r1 with removed OIL run call

```
1  Function r1:
2      _t_enabled1 := state = 1
3
4      /* Checking precondition r1                                    */
5      ...
6      /* Update instance variables r1                                */
7      ...
8      /* Checking postcondition r1                                   */
9      ...
10     /* Handled event r1                                            */
11
12     if _t_enabled1 then
13         TRY_EVENT_P2()// Added successor event
14
15     // Removed _OIL_RUN method call
```

6.2 Removing events OIL run loop scheduler

After the proposed scheduling strategies of the last chapter, there are still situations in which the scheduler has to fall back to the naive OIL run loop scheduler. However, there is still room to improve the naive OIL run loop scheduler itself with the found causal relations.

For this part, the following OIL specification is used to aid the explanation.

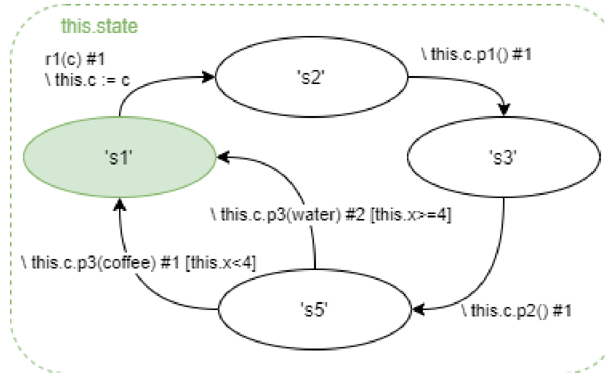

Figure 17: OIL example reduced scheduled OIL run loop scheduler events

Example 6.2 *In Figure 17 an OIL specification can be seen with one reactive event r1 and three proactive events p1, p2 and p3. The gathering approaches detect the that p1#1 is associated with* T_S^* *of r1#1 and p2#1 is associated with* T_S^* *of p1#1. Next the p1#1 transition is associated with* T_{NS}^* *of p2#1 and both transitions labeled with the p3 event. The p2#1 transition is associated with* T_{NS}^* *of r1#1 and both transitions labeled with the p3 event. The gathering approaches are not able to detect transitions associated with* T_S^* *for the p2#1 transition. The gathering approaches associated both transitions labeled with the p3 event with* T_{PS}^* *of p2#1.*

Removing events OIL run loop scheduler The naive OIL run loop scheduler tries to schedule all proactive events to ensure the run-to-completion semantics. This behavior of the scheduler could be improved by only scheduling events that are not completely accounted for.

In one of the previous scheduling strategies, successor events were scheduled, see the first section of this chapter. Also, there is knowledge from transitions associated with T^*_{NS}. If all transitions from a specific event are always associated with either T^*_S or T^*_{NS} it is not needed to depend on the OIL run loop scheduler to schedule this specific event. The specific event is either always scheduled already or there is no need for scheduling it at all. Note that this check does not depend on the last handled event. Each unique argument combination for an event is analyzed separately as each unique argument combination is also tried separately by the OIL run loop scheduler. This makes it possible to remove single argument combinations from the OIL run loop scheduler.

In Example 6.2, the *p1* and *p2* events are not scheduled from the OIL run loop scheduler as can be seen by the removed try method call for both events. This is because for these events all transitions (*p1#1* and *p2#1*) are associated with either T^*_S or T^*_{NS} of all transitions in the OIL specification. During the code generation, the events on the labels of the transitions *p3#1* and *p3#2* are analyzed separately as these transitions have unique argument combinations.

The original OIL run method looks as follows for the OIL specification of Example 6.2.

Algorithm 14: Pseudocode for naive the OIL run loop scheduler

```
1  Function _OIL_RUN:
2      _scheduler_busy := true
3
4      while _scheduler_busy do
5          _scheduler_busy := false
6          _scheduler_busy := (TRY_EVENT_p1() or _scheduler_busy)
7          _scheduler_busy := (TRY_EVENT_p2() or _scheduler_busy)
8          _scheduler_busy := (TRY_EVENT_p3(water) or _scheduler_busy)
9          _scheduler_busy := (TRY_EVENT_p3(coffee) or _scheduler_busy)
10
```

The OIL run method looks as follows when the scheduling strategy is used that removes events from the naive OIL run loop scheduler. The try method calls for the event *p1* and *p2* are removed.

Algorithm 15: Pseudocode for the OIL run loop scheduler with removed entries

```
1  Function _OIL_RUN:
2      _scheduler_busy := true
3
4      while _scheduler_busy do
5          _scheduler_busy := false
6          _scheduler_busy := (TRY_EVENT_p3(water) or _scheduler_busy)
7          _scheduler_busy := (TRY_EVENT_p3(coffee) or _scheduler_busy)
8
```

6.3 Recursive OIL scheduler

Another strategy would be to stop using the OIL run loop scheduler for ensuring the run-to-completion semantics and start using a recursive scheduling strategy. The big downside of the OIL run loop scheduler is that it is a single global scheduler that can not be optimized for specific parts of the OIL specification. A recursive OIL scheduler aims to fix this by having a different local scheduler for every event. The basic scheduling improvement strategies can still be used together with this recursive scheduler.

The following OIL specification is used to explain the recursive OIL scheduler.

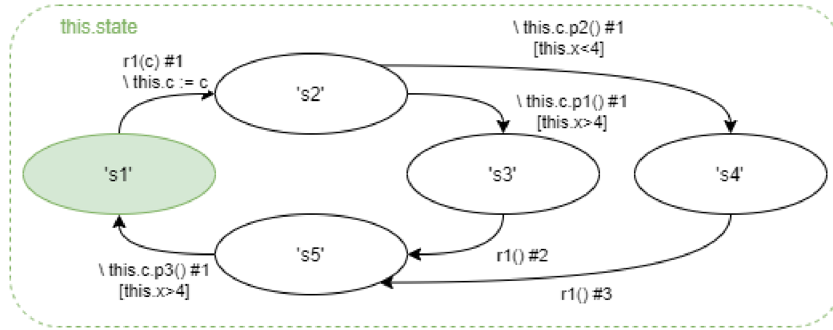

Figure 18: OIL example recursive scheduling

Example 6.3 *The OIL specification in Figure 18 has 3 reactive transitions labeled with the same event. All of them are followed by a transition labeled with a proactive event r1. There are also 3 transitions labeled with different proactive events; notice that guards are placed on these transitions. In this example, the gathering approaches find that p1#1 and p2#1 are associated with T^*_{PS} of the transition r1#1. Next, the p3#1 transition is associated with T^*_{PS} of both r1#2 and r1#3. The gathering approaches also find that both the transition p1#1 and p2#1 are associated with T^*_{NS} of both r1#2 and r1#3. The transition p3#1 is associated with T^*_{NS} of r1#1. Using the OIL run loop scheduler this OIL specification could not be optimized; the OIL run loop scheduler is discussed in Chapter 6.1.*

Recursive scheduling Adding a local scheduler at the end of each handled event creates a situation in which event-specific information can be used by the scheduler. In this explanation we refer to the event for which the local scheduler is created as the preceding event.

First, we need to gather the relevant events that are needed in each local scheduler. Each transition labeled with the preceding event can when fired result in the need for the scheduling of different proactive events to guarantee the run to completion semantics. So we need to find the proactive events that are needed to guarantee the run to completion semantics for each transition labeled with the preceding event. Whenever a transition labeled with the preceding event has a transition associated with T^*_S one of them can be selected arbitrarily and the event on the label of this associated transition can be used in the local scheduler. Only using this event in the local scheduler is sufficient as it can always be produced when the transition fires. Whenever a transition labeled with the preceding event has no transitions associated with T^*_S all the events on the labels of the transitions associated with both T^*_{PS} and T^*_U are used in the local scheduler of the preceding event. In this way transitions associated with T^*_{NS} are used to reduce the number of events in the local scheduler. When no causal relations are gathered all proactive transitions will be used as all proactive transitions are associated with T^*_U.

The events in the local scheduler are used in such a way that only the events are tried that were gathered for a transition that is enabled. When none of the tried events result in the production of an event the run to completion semantics of OIL is guaranteed. This is because all events are tried on the labels of the transitions associated with T^*_{PS} and T^*_U of enabled transitions. In this case an event is not tried if all transitions labeled with this event are associated with T^*_{NS} of all the enabled transitions. These events can safely be skipped because the gathering approaches showed that none of its transitions can fire. However, it could happen that one of the tried events successfully produces. In this case the current local scheduler is abandoned and the local scheduler of the produced event is used to try different events. This is because the abandoned scheduler is only relevant directly after its preceding event has been handled. The local scheduler of the produced event is now responsible for the run to completion semantics. Note that the local scheduler is guaranteed to change when events are scheduled from transitions associated with T^*_S.

This scheduling strategy makes sure that events are never tried when all transitions labeled with these events are associated with T^*_{NS}. It also implements a few optimizations that had to be introduced separately using the OIL run loop scheduling strategy, such as tracking the last produced event, scheduling successor events, and removing events from the OIL run loop scheduler completely, see Chapter 4, 6.1, and 6.2.

In Algorithm 16 the generated code for the *r1* event of Example 6.3 can be seen including its local scheduler. First, note that each try method call in this local scheduler is only listed ones, so a try method is only called at most once for each combination of enabled transitions. This minimizes the number of tried events when multiple transitions are enabled. Next, the *_triggered* variable is used to track if an event has been successfully produced. When this variable becomes *true* all remaining events are not tried anymore as this local scheduler is abandoned. The short circuit evaluation of a boolean expression is used to accomplish this behavior. Whenever an event is successfully produced another local scheduler is used as this produced event has its own local scheduler.

In Example 6.3 there are three transitions labeled with the event *r1*. For the transition *r1#1* the events *p1* and *p2* are used as the transition *r1#1* has no transitions associated with T^*_S and the transitions *p1#1* and *p2#1* are associated with T^*_{PS} of *r1#1*. For both the transitions *r1#2* and *r1#3* the event *p3* is used as both transitions have no transitions associated with T^*_S and associate *p3#1* with T^*_{PS}. In Algorithm 16 try method calls of the three used events are visible. These are only called when the appropriate transition is enabled.

After the transition *r1#2* has fired the state *'s5'* becomes active. The *r1#2* transition has no transitions associated with T^*_S. So if we would have used the OIL run loop scheduler we would have to fall back to it filled with the try methods for the events *p1*, *p2* and *p3*. However, we are using the recursive scheduler so we only need to schedule the events used for the transition *r1#2*; this is only the event *p3*. Now there are two options the *p3* event can be produced or not. When the event can not be produced the run to completion semantics are guaranteed as all transitions on the labels of the remaining proactive events are associated with T^*_{NS} of *r1#1* and can therefore never fire and result in the production of an event. The other option is that the *p3* event can be produced; now the local scheduler of *r1* is abandoned and replaced by the local scheduler of *p3*. This local scheduler has no events as no transitions are associated with either T^*_S, T^*_{PS} or T^*_U of *p3#1*.

Algorithm 16: Pseudocode for a reactive event with recursive scheduling

```
1  Function r1:
2      _t_enabled1 := state = 1
3      _t_enabled2 := state = 3
4      _t_enabled3 := state = 5
5
6      /* Checking precondition r1                                              */
7      ...
8      /* Update instance variables r1                                          */
9      ...
10     /* Checking postcondition r1                                             */
11     ...
12     /* Handled event r1                                                      */
13
14     _triggered := false
15     _triggered := (_triggered or (_t_enabled1 and TRY_EVENT_P1()))
16     _triggered := (_triggered or (_t_enabled1 and TRY_EVENT_P2()))
17     (_triggered or ((_t_enabled2 or _t_enabled3) and TRY_EVENT_P3()))
```

6.4 Prioritizing events

When either the OIL run loop scheduler or recursive OIL scheduler is generated each try method call is placed in a specific scheduling order, in Algorithm 16 the scheduling order *p1*, *p2* followed by *p3* is visible. This scheduling order determines which try method is called first when multiple have to be called to ensure the run-to-completion semantics.

To aid the explanation of how the scheduling order can be generated the following example is used.

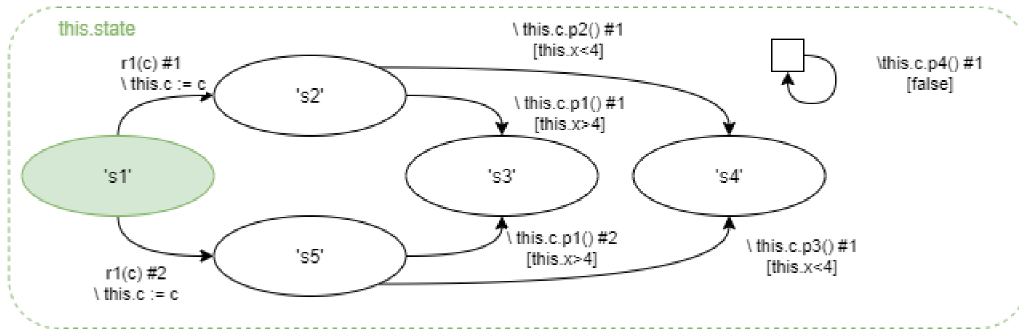

Figure 19: OIL example recursive scheduling

Example 6.4 *For this example we focus on the causal relations from the transitions r1#1 and r1#2. The transitions p1#1 and p2#1 are associated with* T^*_{PS} *of r1#1. Next, the p1#2 and p3#1 transitions are associated with* T^*_{PS} *of r1#2. The transition p4#1 is associated with* T^*_U *of r1#1. In this example the transition are defined in the following order in the OIL specification p4#1, p3#1, p2#1, p1#1, p1#2, r1#1 and r1#2.*

The causal relation gathering approaches gathered transitions associated with T_S^*, T_{PS}^*, and T_U^*. Each of these functions should have its own priority for creating the scheduling order. Transitions associated with T_S^* can always fire after the preceding transition so events on their labels should have a high priority. The gathering approaches could not determine if transitions associated with T_{PS}^* can always be fired after the preceding transition. However, the gathering approaches determined that these transitions are more likely to fire than transitions associated with T_U^*. Events on their labels therefore deserve priority over events on the labels of transitions associated with T_U^*. It is possible to create an event order using this prioritization for each transition.

The event orders for the OIL run loop scheduler are first filled with the transitions associated with T_S^* followed by the transition associated with T_{PS}^*, followed by the transitions associated with T_U^*. The recursive OIL run loop scheduler will also use the same prioritization but note that only one event will be in the event order if the preceding transition has a transition associated with T_S^*, see Chapter 6.3.

These event orders have to be combined for all transitions to one scheduling order for the OIL run loop scheduler. For the recursive OIL scheduler it is only needed to combine the event orders from transitions labeled with the same event. During testing these scheduling orders are only generated for the recursive OIL scheduler as the scheduling orders retain more meaning when they are the result of fewer combined event orders.

As a result of the found causal relations the event order of *r1#1* looks as follows; *p1*, *p2* followed by *p4* as both *p1#1* and *p2#1* are associated with T_{PS}^* of *r1#1* and *p4#1* is associated with T_U^* of *r1#1*. The event order for *r1#2* is *p1* followed by *p3* as both are associated with T_{PS}^* of *r1#2*. These two event orders have to be combined to get one scheduling order that can be used for ordering the events in the recursive OIL scheduler.

Keep order from OIL specification This approach makes users of the OIL language responsible for determining an ideal scheduling order. The original OIL specification had transitions defined in a specific order. The events on the transitions in this specification ordering can be used to order the scheduling order. Events are only added to the scheduling order if they are located in one of the combined event orders.

So the scheduling order looks as follows; *p4*, *p3*, *p2* and *p1* as it was stated that the transitions are defined in the specification order $p4\#1$, $p3\#1$, $p2\#1$, $p1\#1$, $p1\#2$, $r1\#1$ and $r1\#2$.

Round-robin The next prioritization strategy that is considered is the round-robin prioritization strategy. To construct the scheduling order it first takes the first event of the event order of the first transition in the specification order. Afterward, it takes the first event of the event order of the second transition in the specification order. When the first event of each event order is taken the second event of the event orders is taken and so on. Each event is only needed once so whenever an entry already exists in the scheduling order it is not appended.

So the events in the scheduling order of *r1* are acquired as follows. First, the event *p1* is taken as it is the first event in the event order of the first transition in the specification order (*r1#1*). Next, the event *p1* is taken as it is the first event of the event order of the second transition in the specification order (*r1#2*). Note that this event is already taken so it is not appended to the scheduling order. Now the event *p2* is taken as it is the second event in the event order of the first transition in the specification order. Followed by *p3* as it is the second event in the event order of the second transition in the specification order. The scheduling order finishes with the event *p4* as it is the third event in the event order of the first transition in the specification order. So the events in the scheduling order of *r1* are ordered as follows; *p1*, *p2*, *p3* followed by *p4*.

Calculate score Another prioritization strategy is to calculate a score for each event from the event orders. The score should take into account that an event is on the label of transitions associated with different combinations of T^*_S, T^*_{PS} and T^*_U.

This is accomplished with the help of the following formula for determining the score of an event in a single event order. The position of an event in the event order is referred to as the index and starts counting from one for this formula. The scores are added up for all the events in the event orders that are combined. Afterward, the scores are ordered from high to low to acquire a single scheduling order.

$$1/(EventsInEventOrder * IndexInEventOrder)$$

Using this formula priority is given to three aspects of the combined event orders. First, events in short event orders are given priority. These schedules are the result of multiple transitions associated with T^*_{NS} or a transition associated with T^*_S in the case of the recursive scheduler. These causal relations guarantee that specific transitions can always fire or not and should influence the score. Next, the index in the event order influences the priority. The events resulting from transitions associated with T^*_S are placed earlier in the event orders and by using the index in the event order priority is introduced to these events. This index is also used to introduce priority for events on the labels of transitions associated with T^*_{PS} over events on the labels of transitions associated with T^*_U. At last, whenever an event occurs in multiple event orders its priority increases. This is relevant as multiple transitions can be enabled for which causal relations were found. If a transition is located in multiple event orders it is more likely that one of the enabled transitions results in the production of the event so its priority should increase.

The influence of these three aspects can be scaled individually in the calculated score by introducing a factor in front of the variable *EventsInEventOrder* or *IndexInEventOrder* or by introducing a factor after the scores of the event orders are summed up. This last factor should always depend on the number of times an event is located in one of the combined event orders. Some tests were conducted but in the end no factors were introduced as this gave the best results.

So looking at the example each event gets the following score.

Event	Score for *r1#1*	Score for *r1#2*	Total score
p1	1/(3*1)	1/(2*1)	5/6
p2	0	1/(2*2)	1/4
p3	1/(3*2)	0	1/6
p4	1/(3*3)	0	1/9

Table 1: Calculated scores for prioritization

So when these scores are ordered the scheduling order becomes; *p1*, *p2*, *p3* followed by *p4*.

7 Altering OIL specifications

In the previous chapters, a few strategies were discussed to improve the scheduling of OIL specifications. This chapter focuses on altering OIL specifications to make them easier to schedule.

7.1 Mealy machine

A mealy machine can be transformed into an OIL specification. These OIL specifications are set up in such a way that each reactive event is always followed by a proactive event. This set of OIL specifications does also not contain any proactive events that are not preceded by a reactive event. The transitions of proactive events are connected directly to the transitions from the reactive event via a single state. These OIL specifications also only have one region and the events are located in at most one concern. Next, transitions of these events do not have a guard.

An example of such an OIL specification can be found below.

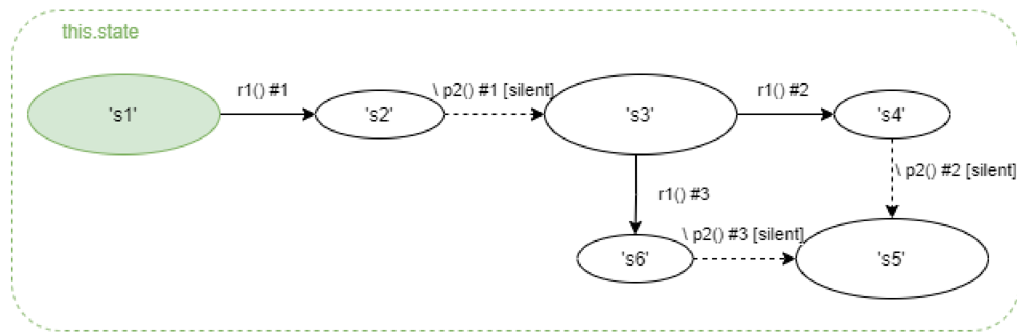

Figure 20: OIL example mealy machine

Note that the states connecting reactive and proactive transition indicated with the smaller ovals are all unstable areas as we have seen in Chapter 5.6. The syntax analysis looking for implied transitions finds all causal relations correctly for the OIL specifications transformed from a mealy machine. This is because there is at most one concern and no guards so there are no transitions associated with T_{PS}^{syntax}. This specification also only has one region resulting in the syntax analysis on implied transition only having to look for the connecting state between transitions. This results in the correct detection of transitions associated with both T_{S}^{syntax} and T_{NS}^{syntax} which results in the scheduling being perfect; the number of tried events is equal to the number of produced events.

Canon Production Printing B.V. already uses tooling that generates these mealy machines. Testing with OIL specifications transformed from these mealy machines resulted in perfect scheduling.

7.2 LTS

OIL specifications can contain guards, assignments, and synchronization between concerns that make analysis hard. It is possible to remove these by generating an LTS from an OIL specification. In this case, every possible LTS state of the OIL specification is defined together with all LTS transitions between them. The OIL tooling is already capable of generating mCRL2[2] specifications from OIL specifications. The mCRL2 tooling can linearise these mCRL2 specifications and generate an LTS from them. So a transformation strategy could be introduced to transform these LTSs back to OIL specifications. This can be done by creating a single region and placing OIL states for each LTS state in this region. Next, all LTS transitions can be used to create OIL transitions connecting the OIL states. These OIL transitions are all labeled with the same concern. This transformation strategy would make the OIL specification simpler to analyze as guards, assignments, and multiple concerns are removed and all arguments are defined.

Below you can see an example of an OIL specification that is hard to schedule as a result of the assignments and guards that are used. This specification is also hard to schedule as there can be multiple regions active at the same time.

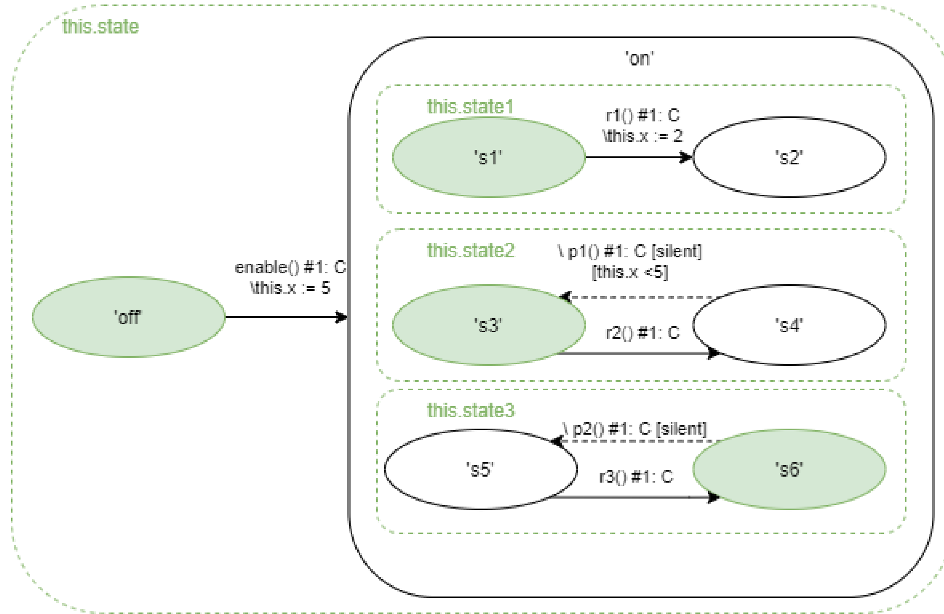

Figure 21: OIL example hard to schedule

The previous OIL specification looks as follows after it has been transformed to an LTS and this LTS is used to generate a new OIL specification.

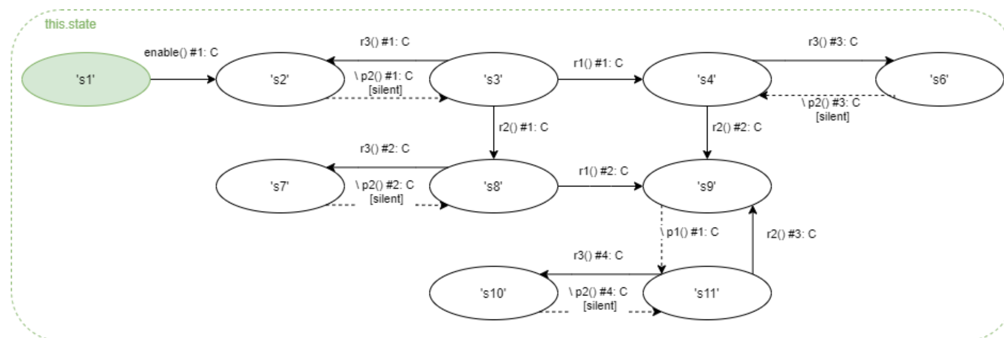

Figure 22: OIL example created from LTS

Note that this generated OIL specification does not have any guards, assignments, has only one region, has at most one concern, and all arguments are defined.

The syntax analysis on implied transitions can do a perfect job on these generated OIL specifications. This is because there is at most one concern and no guards so there are no transitions associated with T_{PS}^{syntax}. The specification also only has one region resulting in the syntax analysis on implied transition only having to look for the connecting state between transitions. This results in the correct detection of transitions associated with both T_{S}^{syntax} and T_{NS}^{syntax} which results in the scheduling always being perfect; the number of tried events is equal to the number of produced event. However, there is a big downside to this transformation strategy; the generated OIL specifications can become very big.

Below you can see a list of the used test OIL specifications with some of their properties. Visualizations of these specifications can be found in Appendix A.4. The EPC case is not visualized.

- Coffee: An OIL specification that uses a few instance variables to keep track of properties of a coffee machine. In this OIL specification, it is possible to make a selection of one of three consumables; water, coffee, and espresso. These consumables each require different amounts of water and beans which can always be refilled. The cost of each consumable is also different and can only be paid using 50 cent or 1 euro coins. The OIL specification will always return the correct amount of change if too much is paid.

- Game: An OIL specification with many different proactive events used over different concerns. In this OIL specification, a game is modeled in which a character can be in a few different states. The character can be laying, standing, crouching, and sitting. Only when the character is standing it can also start walking or running and it is possible to eat but this is only the case when the character is not walking or running.

- EPC: An OIL specification that models a software component used in production.

These test OIL specifications have the following properties.

Specification	#OIL Areas	#OIL Transitions	#OIL instance variables
Coffee	14	22	8
Game	21	24	5
EPC	26	22	8

Table 2: OIL specifications information

In the next table properties of the LTSs generated by mCRL2 can be seen for these test OIL specifications. To acquire the LTS specifications weak trace equivalence was used to reduce the number of states and transitions.

Specification	#LTS States	#LTS Transitions	LTS generation time [s]
Coffee	2564	5729	2.745
Game	10	17	0.005
EPC	1172	3201	2.105

Table 3: LTSs information

The Coffee and EPC specifications have a lot of states and transitions when transformed to an LTS compared to the number of states and transitions that the original OIL specifications had. This results in very big OIL specifications after the LTSs are transformed back to OIL specifications; the coffee example has 8331 lines that take around four seconds to generate. The Spoofax tooling struggles with these big file sizes and it takes around seven minutes to generate code for it.

Whenever an event is handled the generated code starts by determining which transitions are enabled labeled with this event. So, the number of computations does increase when the number of transitions increases labeled with an event. In the coffee example, there is an event that is on the label of 905 transitions which results in 905 checks every time this event is handled.

This transformation strategy has the benefit that the proposed gathering approaches collect enough causal relations to do the scheduling perfectly. During testing, it was however only really useful for specific OIL specifications like Game, which has a minimal number of states and transitions in its LTS. So to generalize this transformation strategy is useful for specifications that have less than 50 transitions in the generated OIL specification. Using this transformation strategy on bigger specifications will quickly result in very ineffective code.

There are however code structures for generating faster code that can be generated from OIL specifications having one region, no guards, have no assignment, and at most one concern. These properties ensure that only one transition can be fired at a time and that if the source area of a transition is active the transition is always enabled. A code structure called the state functions struct pattern could be used in this case to be able to generate effective code even in the case when an OIL specification has a lot of transitions. This code structure and some additional templates which may be helpful for generating code are discussed in [12][13][14]. However, no implementation was created for any of these alternate code structures as big rewrites of the code generation process were needed.

8 Verification

Three verification approaches are used to verify parts of the additions to the code generator. First, the formal definitions are used to prove all lemmas that were stated for the different causal relation gathering approaches, see Appendix A.3. Proving that the causal relation gathering approaches always collect transitions associated with T_S and T_{NS} gives additional confidence in the used causal relation gathering approaches.

For each implemented gathering approach postconditions are always checked to verify part of the implementation. These checks make sure that the gathered transitions associated with T_S^*, T_{NS}^*, T_{PS}^* and T_U^* partition all the proactive transitions correctly.

Next, a tool called JTorX[15] is used to verify the generated code for a selection of OIL specifications.

8.1 JTorX

Mark Frenken[6] was responsible for creating the initial version of the code generator for the OIL tooling[6]. Here a tool was used called JTorX to verify parts of the code generator. This tool uses an LTS of a model and an executable that implements this model to do the verification. By inputting allowed events from the LTS and comparing outgoing events to the LTS JTorX tests that the executable has the same behavior as the LTS. This verification however does not give guarantees, it only inputs a finite number of events and compares the output. It does not prove that the LTS and the executable have exactly the same behavior. This verification approach is reused to see if the altered parts in the code generator still function correctly. The LTS of an OIL specification can be acquired by generating an mCRL2 specification and using the mCRL2 tooling to generate an LTS from it.

The scheduler decides which proactive event is scheduled so not all states of the LTS can always be reached. This can happen when for example a specific proactive event has a higher priority over another when both may be produced. Therefore the coverage detection of JTorX can not always detect if all possible behavior of the executable has been explored. It is however certainly possible to let it run for a while and see how many states it can verify. Three runs are used in the case that the coverage detection of JTorX could not detect that all behavior was covered in 80,000 execution steps.

The code generator is verified for three test OIL specifications; coffee, game, and EPC. A description of each specification can be found in Chapter 7.2. Visualizations of these specifications can be found in Appendix A.4. The EPC case is not visualized. The coverage of JTorX for these test OIL specifications is listed next in Table 4.

Test	#Exec steps	#LTS states	#Cov states	#LTS transitions	#Cov transitions
Coffee	(3x) 80000	2564	1376	5729	2697
Game	32	10	10	17	17
EPC	(3x) 80000	1172	750	3201	2231

Table 4: Coverage information

Only the transitions associated with T_S and T_{NS} do influence which events are tried by the scheduler. There are however only a few ways that these causal relations result in different events being scheduled by the scheduling strategies.

- All transitions from an event are always associated with either T_S or T_{NS}.
- All transitions from an event are not always associated with either T_S or T_{NS}.
- A transition in an event has a transition associated with T_S.
- A transition in an event has no transitions associated with T_S but has a transition associated with T_{NS}.
- A transition in an event has all proactive transitions associated with T_{NS}.
- A transition in an event has no transitions associated with either T_S or T_{NS}.

All these cases are covered by the three test OIL specifications tested with JTorX. The verification is repeated for both the recursive OIL scheduler and OIL run loop scheduler with all gathering approaches enabled. See Chapter 6 for more information on the recursive OIL scheduler and the OIL run loop scheduler.

The JTorX tooling did not find any behavior in the executables that was not allowed by the LTSs. However, this verification approach gives no guarantees. First, it is possible that the model-based testing is terminated too soon; it could be possible that the executables have wrong behavior after 100,000 execution steps. Next, the three selected test cases do not cover all possible behavior that can be specified with OIL. The generated code for the three OIL specifications can be correct but this does not say anything about OIL specifications that use behavior that is not covered by these specifications.

While the test OIL specifications did not cover all possible behavior that can be specified using OIL they did cover all possible scheduling alternations that can be introduced by the new scheduling strategies. So because of these tests and the proves delivered for the causal relations it is currently believed that the altered code generator always produces the correct code.

9 Results

In this chapter, the results of a few test cases with different schedulers are discussed. The number of tried events is compared to the total number of produced events. Improvements would preferably avoid tries that have no possibility of producing. For a perfect scheduler, this means that the number of tried events is equal to the number of produced events.

A few properties of each test are listed in tables throughout this chapter. In the first column, the name of the test is listed indicating how the code was generated. In the following column, the specific trace is listed, this is either a handpicked or randomly generated trace, see Chapter 9.1. Next, the number of tried events is listed, these are the number of times a try method is called. The relative improvement is listed after each number relative to the naive scheduler. The number of the produced events is listed in the next column, these numbers do only change when a different trace or OIL specification is used. As confluent proactivity ensures that for valid OIL specifications each scheduling decision always eventually results in the production of the same proactive events. The last two columns list the time it takes to execute the generated code and the time it takes to generate the code itself. The execution times are acquired by running the test cases 10,000 times and averaging the measured times. Measured execution times smaller than 0.0001 seconds are still not very accurate as other programs running at the same time definitely influenced the results. The relative improvement of the execution times is also listed after each number relative to the execution time from the generated code using the naive scheduler. All tables with results can be found in Appendix A.1. Interesting parts of these tables will be copied throughout this chapter.

9.1 Selected test traces

All the optimizations aim to reduce the number of tries that are needed to produce the proactive events. A few test OIL specifications are used to see the effectiveness of these optimizations. These are the OIL specifications Coffee, Game, and EPC. A description of each specification can be found in Chapter 7.2. Visualizations of these specifications can be found in Appendix A.4. The EPC case is not visualized.

A specific test case has a specific trace of events through these OIL specifications. These can be selected manually but may result in biased results. Therefore the choice is made to have two test traces per tested OIL specification; the first is a manually selected trace that mirrors normal usage and the second is a selection of random traces.

The used random traces are constructed by using 80 different traces that are looped through the OIL specification, each event in these traces is selected randomly from the enabled events. In Table 5, an overview can be seen of the number of events in each test case. With a few small tests it was confirmed that all these traces covered all transitions from the test OIL specifications; each OIL transition is fired at least once during one of the traces.

Test case	#Reactive events	#Proactive events	#Total events
Coffee handpicked	15	9	24
Coffee random	730	240	970
Game handpicked	8	22	30
Game random	638	890	1528
EPC handpicked	14	28	42
EPC random	1669	2423	4092

Table 5: Amount events in selected traces

It is important to understand that these test OIL specifications do not cover all behavior that can be specified using OIL. However, they cover real-life examples and behavior for which the gathering approaches struggle as these specifications use synchronization between concerns, guards, and assignments. So conclusions drawn from these cases are definitely insightful but take into account that these conclusions may not be true for all OIL specifications.

9.2 Different gathering approaches and scheduling strategies

First, different combinations of causal relation gathering approaches are enabled together with a fixed set of scheduling strategies. The used schedulers are the OIL run loop scheduler with all its proposed improvements and the recursive OIL scheduler both with round-robin priority. The name of the test case indicates which gathering approaches are used. No analysis means no gathering approach is used for approximating the causal relations and the code is generated without these relations. Note that in the cases where the concern analysis is disabled the restriction to only analyze transitions with events from at most one concern is used. The basic scheduling improvement strategies are always applied except for the test cases using the original scheduler. Next, no compiler optimization is enabled during the compilation of the generated code using the -O0 flag.

The results of all the test cases can be found in Appendix A.1.1 for the OIL run loop scheduler and in Appendix A.1.2 for the recursive OIL scheduler. In this section, the results of the EPC test specification will be used to discuss the different gathering approaches.

No analysis First, no causal relation gathering approaches are used during the generation of the code.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
No analysis	Handpicked	315 (-24%)	28	< 0.0001	0.766
Naive	Random	42125	2423	0.0064	0.695
No analysis	Random	39449 (-6%)	2423	0.0055 (-14%)	0.766

Table 6: EPC result overview OIL run loop scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
No analysis	Handpicked	413 (-0%)	28	< 0.0001	0.760
Naive	Random	42125	2423	0.0064	0.695
No analysis	Random	42125 (-0%)	2423	0.0064 (-0%)	0.760

Table 7: EPC result overview recursive OIL scheduler

The basic scheduling improvement strategies described in Chapter 4 result in an improvement that is visible during the test cases using the OIL run loop scheduler for the EPC case. The OIL run loop scheduler is still always used, but fewer entries are added to this scheduler.

Note that the OIL run loop scheduler without analysis performs better for all cases in comparison to the recursive OIL scheduler without analysis. This is mainly the result of another tactic that is used to iterate through the scheduling orders. The OIL run loop scheduler goes through each event once before starting at the start again of its scheduling order. The recursive OIL scheduler generated without any causal relations resets to the start of the scheduling order any time an event is produced. Both are correct but for the test specifications, it is more beneficial to go through all events before starting over.

Good The basic scheduling improvement strategies reduce the number of tried events needed for scheduling using the OIL run loop scheduler; -24%. It does this without increasing the time it takes for generating the code.

Bad N/a

Syntax analysis on implied transitions Next, syntax analysis on implied transitions is used to generate code for the test cases, see Chapter 5.3 for more information on the syntax analysis on implied transitions.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
Syntax	Handpicked	148 (-64%)	28	< 0.0001	0.792
Naive	Random	42125	2423	0.0064	0.695
Syntax	Random	14505 (-65%)	2423	0.0024 (-62%)	0.792

Table 8: EPC result overview OIL run loop scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
Syntax	Handpicked	122 (-70%)	28	< 0.0001	0.748
Naive	Random	42125	2423	0.0064	0.695
Syntax	Random	11454 (-73%)	2423	0.0021 (-67%)	0.748

Table 9: EPC result overview recursive OIL scheduler

The OIL specification EPC still has all proactive events in the OIL run loop scheduler while not all of them are necessary, the syntax analysis on implied transitions unfortunately can not detect sufficient causal relations to remove any of them. In a few places, the syntax analysis on implied transitions found enough causal relations to be sure that successor events could be scheduled directly. In some other places the OIL run loop scheduler was not needed anymore and the call to the OIL run loop scheduler was removed.

Using only syntax analysis on implied transitions is also the moment the recursive OIL scheduler starts to shine. It can already make bigger improvements with the same number of causal relations available.

Good The number of tried events is reduced further in comparison to using no analysis to -70% and -73% for the recursive OIL scheduler. The syntax analysis on implied transitions does this without noticeably slowing down the code generation; the results show a very similar generation time of around 0.7 seconds.

Bad N/a

SMT analysis on implied transitions Next, the SMT analysis on implied transitions is taken a closer look at, see Chapter 5.4 for more information on the SMT analysis on implied transitions.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
SMT	Handpicked	148 (-64%)	28	< 0.0001	4.533
Syntax, SMT, Concern	Handpicked	148 (-64%)	28	< 0.0001	2.446
Naive	Random	42125	2423	0.0064	0.695
SMT	Random	14505 (-65%)	2423	0.0024 (-62%)	4.533
Syntax, SMT, Concern	Random	14505 (-65%)	2423	0.0024 (-62%)	2.446

Table 10: EPC result overview OIL run loop scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
SMT	Handpicked	122 (-70%)	28	< 0.0001	4.611
Syntax, SMT, Concern	Handpicked	122 (-70%)	28	< 0.0001	2.601
Naive	Random	42125	2423	0.0064	0.695
SMT	Random	11454 (-73%)	2423	0.0021 (-67%)	4.611
Syntax, SMT, Concern	Random	11454 (-73%)	2423	0.0021 (-67%)	2.601

Table 11: EPC result overview recursive OIL scheduler

While the SMT analysis on implied transitions does find more causal relations, it does not result in changes in the results. The SMT analysis on implied transitions is mostly better when OIL specifications use data, the tested OIL specifications however to do not really use a lot of data. So in the end similar results to the syntax analysis on implied transitions are achieved.

When only the SMT analysis on implied transitions is enabled the code generation process takes significantly longer. Instead of a generation time of around 0.7 seconds, a single generation time could now take up to 4.6 seconds. This should not be a problem but is certainly something to take into account when selecting the gathering approaches during the generation of code. The combination of the syntax and SMT analysis on implied transitions reduces the time needed to analyze an OIL specification while still finding all the same causal relations. This is because the SMT analysis on implied transitions is only used on transitions for which the syntax analysis on implied transitions could not find any causal relations.

Good The SMT analysis on implied transitions can interpret more complex transitions as expressions can be taken into account. However, this is not really visible in the results as the SMT analysis on implied transitions is only able to match the results of the syntax analysis on implied transitions.

Bad The SMT analysis on implied transitions takes significantly longer. The results show that the generation time becomes as long as 4.6 seconds when only the SMT analysis on implied transitions is used.

Concern analysis The concern analysis is discussed next, more information for the concern analysis can be found in Chapter 5.5.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Syntax	Handpicked	148 (-64%)	28	< 0.0001	0.792
SMT	Handpicked	148 (-64%)	28	< 0.0001	4.533
Syntax, Concern	Handpicked	148 (-64%)	28	< 0.0001	0.744
Syntax, SMT, Concern	Handpicked	148 (-64%)	28	< 0.0001	2.446
Syntax	Random	14505 (-65%)	2423	0.0024 (-62%)	0.792
SMT	Random	14505 (-65%)	2423	0.0024 (-62%)	4.533
Syntax, Concern	Random	14505 (-65%)	2423	0.0024 (-62%)	0.744
Syntax, SMT, Concern	Random	14505 (-65%)	2423	0.0024 (-62%)	2.446

Table 12: EPC result overview OIL run loop scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Syntax	Handpicked	122 (-70%)	28	< 0.0001	0.748
SMT	Handpicked	122 (-70%)	28	< 0.0001	4.611
Syntax, Concern	Handpicked	122 (-70%)	28	< 0.0001	0.712
Syntax, SMT, Concern	Handpicked	122 (-70%)	28	< 0.0001	2.601
Syntax	Random	11454 (-73%)	2423	0.0021 (-67%)	0.748
SMT	Random	11454 (-73%)	2423	0.0021 (-67%)	0.611
Syntax, Concern	Random	11454 (-73%)	2423	0.0021 (-67%)	0.712
Syntax, SMT, Concern	Random	11454 (-73%)	2423	0.0021 (-67%)	2.601

Table 13: EPC result overview recursive OIL scheduler

The concern analysis does not increase the code generation time in any significant way. It however does find some additional causal relations in for example the EPC specification. Unfortunately, this does not result in any improvements in the achieved results. The additional causal relations do not enable any additional scheduling strategies.

Good The concern analysis is fast and does find additional causal relations.

Bad In the tested specifications no improvements were acquired when the concern analysis was enabled.

Unstable area analysis The following table shows the results for the unstable area analysis, more information from the unstable analysis can be found in5.6.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Syntax	Handpicked	148 (-64%)	28	< 0.0001	0.792
Syntax, SMT, Concern	Handpicked	148 (-64%)	28	< 0.0001	2.446
Syntax, Unstable	Handpicked	107 (-74%)	28	< 0.0001	0.610
All	Handpicked	107 (-74%)	28	< 0.0001	2.328
Syntax	Random	14505 (-65%)	2423	0.0024 (-62%)	0.792
Syntax, SMT, Concern	Random	14505 (-65%)	2423	0.0024 (-62%)	2.446
Syntax, Unstable	Random	10564 (-74%)	2423	0.0020 (-69%)	0.610
All	Random	10564 (-74%)	2423	0.0020 (-69%)	2.328

Table 14: EPC result overview OIL run loop scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Syntax	Handpicked	122 (-70%)	28	< 0.0001	0.748
Syntax, SMT, Concern	Handpicked	122 (-70%)	28	< 0.0001	2.601
Syntax, Unstable	Handpicked	96 (-76%)	28	< 0.0001	0.791
All	Handpicked	96 (-76%)	28	< 0.0001	2.567
Syntax	Random	11454 (-73%)	2423	0.0021 (-67%)	0.748
Syntax, SMT, Concern	Random	11454 (-73%)	2423	0.0021 (-67%)	2.601
Syntax, Unstable	Random	8937 (-79%)	2423	0.0019 (-70%)	0.791
All	Random	8937 (-79%)	2423	0.0019 (-70%)	2.567

Table 15: EPC result overview recursive OIL scheduler

The unstable area analysis does not increase the code generation time in any meaningful way. It however does increase the number of found causal relations which also reduces the number of tries. The recursive OIL scheduler still keeps its edge when these additional causal relations are supplied.

Good The unstable area analysis is fast and does find additional causal relations. Looking at the results the number of tried events for the EPC specification can be reduced with -76 and -79% for the recursive scheduler.

Bad N/a

OIL run loop scheduler The OIL run loop scheduler seems to be the better choice in comparison to the recursive OIL scheduler when no causal relations are found. As stated earlier this is mainly the result of a different tactic that is used to iterate through the scheduling orders.

Good Achieves better results when no causal relations are gathered.

Bad N/a

Recursive OIL scheduler The recursive OIL scheduler is the more effective choice in comparison to the OIL run loop scheduler when causal relations are gathered. In every tested case where causal relations are gathered less try event method calls are needed to do the scheduling whenever the recursive OIL scheduler is used.

Good Achieves better results when causal relations are gathered.

Bad N/a

9.3 Different prioritization

In this chapter, the different prioritization strategies discussed in Chapter 6.4 are tested. These test cases are all conducted on the recursive OIL scheduler with all causal relation gathering approaches enabled. Next, no compiler optimization is enabled during the compilation of the generated code using the -O0 flag. The name of the test case indicates exactly which priority strategy is used. The results for the Coffee specification look as follows. See Appendix A.1.3 for all the tables with results.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	127	9	$<$ 0.0001	0.710
Keep order	Handpicked	40 (-69%)	9	$<$ 0.0001	2.746
Round-robin	Handpicked	58 (-54%	9	$<$ 0.0001	2.615
Calculate score	Handpicked	40 (-69%)	9	$<$ 0.0001	2.404
Naive	Random	5726	240	0.0034	0.710
Keep order	Random	2082 (-64%)	240	0.0009 (-74%)	2.746
Round-robin	Random	2892 (-49%)	240	0.0013 (-62%)	2.615
Calculate score	Random	2082 (-64%)	240	0.0009 (-74%)	2.404

Table 16: Coffee result overview prioritizing events

Keep order from OIL specification The keep order prioritization strategy can achieve the best results together with the prioritization strategy that calculated a score. However, this result is a bit artificial as the users of the OIL language are responsible for determining an ideal scheduling order. The results can easily be ruined by altering the order of transitions in the original OIL specification.

Good The prioritization can give good results. It achieves a reduction of tried events of -69% and -64% with the gathered causal relations.

Bad The users of the OIL language are responsible for determining an ideal scheduling order. The results can easily be ruined by changing the order of transitions in the original OIL specification.

Round-robin Round-robin seems like the worst prioritization strategy. In most cases, it seems that the scheduling order is not really important for the number of tries. This is visible in the table for both the Game and EPC case as nearly everywhere the same results are acquired with all the three prioritization strategies, see Appendix A.1.3. However, there are differences visible in the Coffee OIL specification; in this specification the round-robin prioritization strategy consistently gives the worst results. This is because the round-robin prioritization strategy results in transitions having a higher priority than they do deserve. Events that are listed in the first event order that is combined have a higher priority using this prioritization strategy. The order of combined event orders has no meaning, it should therefore not have any influence.

Good N/a

Bad Gives priority to events that do not deserve it. This results in only a reduction of -54% and -49% of tried events; the lowest reduction in tried events in these test cases.

Calculate score This is the best prioritization strategy that is tested to reduce the number of tries. It also has the benefit that it does not depend on the order of transitions in the original OIL specification. There do however exist edge cases where multiple transitions are enabled in which an event on the label of a transition associated with T^*_{PS} is tried before the event on the label of a transition associated with T_S.

Good Gives good consistent results. It achieves a reduction of tried events of -69% and -64% with the gathered causal relations.

Bad There do exist edge cases where multiple transitions fire at once in which a non-ideal scheduling order is used.

9.4 Different compiler optimizations

In this chapter, the results of a few test cases are listed that focus on the impact of the compiler optimizations. It could be possible that optimizations made to the generated code could already be introduced by the compiler. Therefore the generated code is also compiled using the O2 flag. The O2 flag is chosen because its the highest optimization level of the compiler that does not disregard the size of the generated code like the O3 flag, which might result in increased code size. By comparing the results from the O0 and O2 flags the impact of the compiler can be seen. The test cases for the OIL run loop scheduler and the recursive OIL scheduler are both used with all gathering approaches enabled. The name of the test indicates which scheduler is used together with the O level that is used for compiling the code. The tables with all results for these test cases can be found in Appendix A.1.4. The table with the results from the Game specification can be found below.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive O0	Random	8518	890	0.0026	0.634
Naive O2	Random	8518	890	0.0007 (-73%)	0.634
Improved run loop O0	Random	2166 (-75%)	890	0.0008 (-69%)	3.436
Improved run loop O2	Random	2166 (-75%)	890	0.0002 (-92%)	3.436
Recursive O0	Random	2022 (-76%)	890	0.0008 (-69%)	2.844
Recursive O2	Random	2022 (-76%)	890	0.0002 (-92%)	2.844

Table 17: Game result overview compiler optimizations

First, note that the number of tries is not altered when different optimization levels are chosen. It is also visible that in all test cases a lot of performance is gained when the O2 optimization level is used; for example -73% in execution time for the original generated code. Additionally, this performance gain seems to be independent of the performance gains that were accomplished with the scheduling strategies. It may be possible that different optimizations are used but the O2 level consistently improves the execution time with a factor of around 4.

9.5 Overall remarks

In the end, all the gathering approaches can find a lot of causal relations in the OIL specifications, it seems that the recursive OIL scheduler is most effective in translating these causal relations into generated code compared to the OIL run loop scheduler. It is however important that prioritization strategies are used during scheduling without using the order of transitions in the original OIL specification as these have no meaning. The further usage of the round-robin and keep order prioritization strategy is discouraged as calculating a score for prioritization gives the best results.

Comparing the recursive OIL scheduler with all gathering approaches enabled to the naive scheduler gives an idea of the biggest improvements that could be achieved. The number of tried events is reduced between 64% and 81%, both numbers can be found in Appendix A.1.3. This reduction in the number of tried events also significantly influences the execution time; improvements in execution speed varied between a factor of three and four. In the Appendix A.2 a few charts can be found that plot the execution time as the number of tries changes for each tested OIL specification. These graphs show that the execution times definitely get faster when fewer tries have to be computed. As stated earlier the measurements for the execution times are however not really accurate. In any way, the proposed scheduling strategies improved the execution time for all the test cases.

Further testing also showed that the performance gain acquired by the compiler optimizations was independent of the performance gain of the proposed scheduling strategies. Similar performance gains could be achieved by the compiler on code generated with the naive scheduler and the improved schedulers.

10 Conclusions and future work

In this chapter, the conclusions and future work are discussed.

10.1 Conclusions

We have shown how the scheduling can be improved for code generated from OIL specifications. These improvements result in fewer computations being done during runtime for the scheduling which improves the overall performance of the generated code. For answering Research question RQ1 three main strategies were used.

First, a few basic scheduling improvement strategies can be used without gathering any causal relations from the OIL specification; for example, tracking the last produced event.

However, more improvements can be made to the scheduler if causal relations are gathered. Four functions are used to associate transitions with specific properties to all transitions of an OIL specification (RQ1.1). The T_S^* function is used to associate transitions that are always enabled after a preceding transition and the T_{NS}^* function is used to associate transitions that are never enabled after a preceding transition. The transitions associated with these functions are needed to be sure that alternations in the scheduler are allowed while still guaranteeing the run-to-completion semantics. Additionally, transitions associated with the T_{PS}^* and T_U^* functions are used to introduce priority between events in the scheduler.

The causal relations can be gathered with the use of syntax analysis on the desugared abstract syntax tree but also approaches that use an SMT solver are useful for finding the causal relations. While the SMT solver can evaluate expressions it comes at the cost of being significantly slower. Additionally, two gathering approaches are used to acquire even more causal relations. One focuses on concerns to be able to not be limited by the number of concerns a specific event is located in. Another gathering approach focuses on unstable areas. These are areas in an OIL specification for which the already collected causal relations are enough to do the scheduling when these areas become active.

Using these causal relations it was possible to improve the OIL run loop scheduler and introduce a new scheduler. Improvements can be introduced that complement the OIL run loop scheduler and entries can be removed from the OIL run loop scheduler when specific constraints are met. Additionally, a recursive scheduler can be introduced that is able to act as a local scheduler for each handled event. All these scheduling strategies result in perfect scheduling for simple OIL specifications and improve the scheduling for more complex OIL specifications.

At last, another strategy was looked into to improve the scheduling that altered the OIL specifications. Transforming an OIL specification to its LTS counterpart made scheduling easier but came with the cost of an excessive amount of transitions in some cases. This could however certainly be a good solution for OIL specifications with an LTS containing few transitions. These generated OIL specifications also have specific properties that can certainly still be exploited when code is generated for them.

To answer Research question RQ1.2 and to gain trust that the code generator produces correct code lemmas for the causal relation gathering approaches were proven, checks were introduced in the implementation of the code generator and the generated code of a few test OIL specifications was verified using JTorX.

For answering Research question RQ2 some tests were conducted. The results show a reduction of tries needed to do the scheduling of around 64% up to 81%. This makes the code approximately three to four times faster. The selected test cases were selected to cover real-life examples and behavior that is hard to analyze by to proposed causal relation gathering approaches. It is therefore expected that most OIL specifications have similar or even better results. However, this selection does not cover all OIL specifications, so these results may not be true for all OIL specifications.

In the end, significant improvements can be introduced to the scheduler in the generated code of OIL specifications. However, there are certainly still ways to improve the scheduling even further.

10.2 Future work

Multiple approaches can still be used to improve the scheduling further. These improvements can be used to find even more causal relations between transitions. There is also still room to improve the generated code itself.

There could be transitions in OIL specifications that are labeled with a proactive event that loop to the area they left from and have a guard. Their events can be scheduled differently by trying them until the guard of the transition does not hold anymore, see Chapter 4.

The definitions for T_S and T_{NS} could be restricted to only allow reachable transitions to be used. Currently, it is not clear how reachability can be detected by the gathering approaches. There is no benefit in scheduling events from unreachable transitions so if they can be detected properly they should not be taken into account, see Chapter 5.

The gathering approaches are able to find causal relations. The generated code always computes which transitions are enabled when an event is handled. The causal relations can be used to skip computations needed to check if a transition is enabled or not.

An OIL specification generated from an LTS is easy to analyze by the causal relation gathering approaches. However, the transformation strategy transforming an OIL specification to an LTS and back to an OIL specification is limited as code generated from these OIL specifications is not very effective when a generated OIL specification has a lot of transitions. So it could be tried to create a code generator for OIL specifications that are generated from LTSs. These OIL specifications have specific properties that can certainly be helpful, see Chapter 7.2.

During the verification, JTorX was used to verify the generated code. The generated code had to be altered a bit to interface with JTorx. Ideally, this process should be an option during code generation. This would make it possible to verify generated code more easily.

The recursive OIL scheduler uses recursion which can lead to stack overflow errors when a lot of proactive events are produced after each other. The scheduling is done at the end of each generated method so it should be possible to alter the generated code slightly to introduce tail recursion. A prototype has been implemented that accomplishes this.

Currently, unstable areas are restricted to not be parent areas of any other area. It is not entirely clear if this restriction is necessary. This restriction can be revisited to possibly loosen the restrictions for unstable areas, see Chapter 5.6.

For the detection of unstable areas, only an approach is implemented that uses syntax analysis on the desugared abstract syntax tree. An SMT solver can be used to increase the number of found unstable areas. Conceptually this approach is very similar but the use of the SMT solver should increase the number of causal relations that can be found. In the figure below an unstable area *'s2'* can be spotted. Either the transition labeled with the *p1* or *p2* event is always enabled. Possibly these kinds of areas can be detected using an SMT solver. Note that this construction can be found in the Example 2.4 and the Coffee OIL specification that was used for testing.

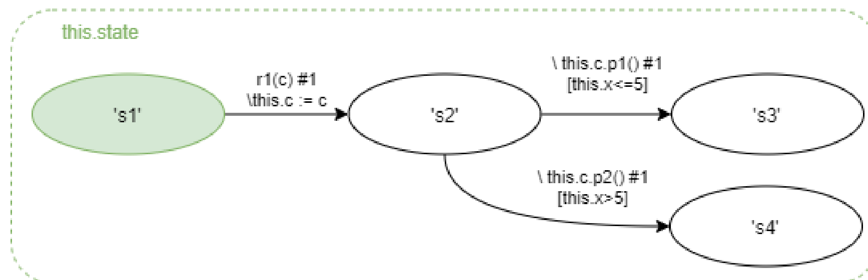

Figure 23: OIL example unstable area SMT analysis

The dual of an unstable area is a stable area. This is an area that if entered is never left; the area stays always active. One of the conditions of a stable area is that its parent area is also a stable area. If such a stable area is a region it may be possible to schedule proactive events from transitions that leave the initial state of this region directly after the event from the transition that entered the region. There is the suspicion that this is possible as the transition entering the region always has the same behavior after it has fired; the behavior after the initial state. Always scheduling the events from these transitions could also relax the restriction on unstable areas not being initial states. This restriction was introduced to make sure events from transitions leaving the initial state were always scheduled correctly. The scheduling of these events can now also be done using these stable areas.

Unstable areas have a requirement that they can not be initial states. This limits the number of unstable areas that can be found. However, it may be possible to alter OIL specifications to have some kind of lasso behavior. So first some initial behavior followed by a loop. By separating the initial behavior with the loop it may be possible to point out more unstable areas in this loop.

References

[1] Lennart C. L. Kats and Eelco Visser, The Spoofax language workbench, in OOPSLA (2010)

[2] Olav Bunte, Jan Friso Groote, Jeroen J.A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, Tim A.C. Willemse, The mCRL2 Toolset for Analysing Concurrent Systems: Improvements in Expressivity and Usability, in TACAS (2019)

[3] Jasper Denkers, Louis C.M. van Gool and Eelco Visser, Migrating custom DSL implementations to a language workbench (tool demo), in SLE (2018)

[4] Olav Bunte, Louis C.M. van Gool, and Tim A. C. Willemse, Formal Verification of OIL Component Specifications using mCRL2, submitted to STTT (March 2021)

[5] Martin Bravenboer, Karl T. Kalleberg, Rob Vermaas and Eelco Visser, Stratego/xt 0.17. A language and toolset for program transformation, in Science of Computer Programming (2008)

[6] Mark Frenken, Code generation and model-based testing in context of OIL, (November 2019)

[7] Michal Bliznak, Tomas Dulik, Roman Jasek and Pavel Varacha, Optimized Source Code Generation from State Charts, in CSCC (July 2012)

[8] Leonardo de Moura and Nikolaj Bjørner, Z3: An Efficient SMT Solver, in TACAS (April 2008)

[9] Nada Amin and Tiark Rompf, Computing with an SMT Solver, in TAP (May 2014)

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli, The SMT-LIB Standard Version 2.6, (July 2017)

[11] Vaclav Pech, Alex Shatalin and Markus Völter, JetBrains MPS as a tool for extending Java, in PPPJ (2013)

[12] Paul Adamczyk, Selected Patterns for Implementing Finite State Machines, in PLoP (November 2004)

[13] Eladio Domínguez, Beatriz Pérez , Ángel L. Rubio and Marla A. Zapata, A systematic review of code generation proposals from state machine specifications, in Inf Softw Technol (May 2012)

[14] Maamar Hamri, RRabah Messouci and Claudia Frydman, Discrete Event Design Patterns, SIGSIM PADS (May 2013)

[15] Axel Belinfante, JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution, TACAS (Oktober 2010)

[16] Axel Belinfante, JTorX: exploring model-based testing, (February 2014)

A Appendix

A.1 Test case results

In this chapter, all results are listed from all the test cases.

A.1.1 Results OIL run loop scheduler

In the table below an overview can be found of the results from the different combinations of causal relation gathering approaches used together with the OIL run loop scheduler.

Take into account that the execution times are not accurate. They can be used to get an idea of the impact of causal relation gathering approaches but differences smaller than 0.0001 seconds should be considered irrelevant.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	127	9	< 0.0001	0.710
No analysis	Handpicked	72 (-43%)	9	< 0.0001	0.732
Syntax	Handpicked	72 (-43%)	9	< 0.0001	0.693
SMT	Handpicked	72 (-43%)	9	< 0.0001	5.509
Syntax, Concern	Handpicked	70 (-45%)	9	< 0.0001	0.654
Syntax, SMT, Concern	Handpicked	70 (-45%)	9	< 0.0001	2.947
Syntax, Unstable	Handpicked	70 (-45%)	9	< 0.0001	0.694
All	Handpicked	70 (-45%)	9	< 0.0001	2.464
Naive	Random	5726	240	0.0034	0.710
No analysis	Random	3240 (-43%)	240	0.0015 (-56%)	0.732
Syntax	Random	3240 (-43%)	240	0.0015 (-56%)	0.693
SMT	Random	3240 (-43%)	240	0.0015 (-56%)	5.509
Syntax, Concern	Random	3216 (-44%)	240	0.0015 (-56%)	0.654
Syntax, SMT, Concern	Random	3216 (-44%)	240	0.0015 (-56%)	2.947
Syntax, Unstable	Random	3216 (-44%)	240	0.0015 (-56%)	0.694
All	Random	3216 (-44%)	240	0.0015 (-56%)	2.464

Table 18: Coffee result overview OIL run loop scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	179	22	< 0.0001	0.634
No analysis	Handpicked	104 (-42%)	22	< 0.0001	0.702
Syntax	Handpicked	89 (-50%)	22	< 0.0001	0.721
SMT	Handpicked	89 (-50%)	22	< 0.0001	5.257
Syntax, Concern	Handpicked	89 (-50%)	22	< 0.0001	0.658
Syntax, SMT, Concern	Handpicked	89 (-50%)	22	< 0.0001	3.516
Syntax, Unstable	Handpicked	38 (-78%)	22	< 0.0001	0.638
All	Handpicked	38 (-78%)	22	< 0.0001	3.436
Naive	Random	8518	890	0.0026	0.634
No analysis	Random	6831 (-20%)	890	0.0020 (-23%)	0.702
Syntax	Random	5341 (-37%)	890	0.0015 (-42%)	0.721
SMT	Random	5341 (-37%)	890	0.0015 (-42%)	5.257
Syntax, Concern	Random	5341 (-37%)	890	0.0015 (-42%)	0.658
Syntax, SMT, Concern	Random	5341 (-37%)	890	0.0015 (-42%)	3.516
Syntax, Unstable	Random	2166 (-75%)	890	0.0008 (-69%)	0.638
All	Random	2166 (-75%)	890	0.0008 (-69%)	3.436

Table 19: Game result overview OIL run loop scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
No analysis	Handpicked	315 (-24%)	28	< 0.0001	0.766
Syntax	Handpicked	148 (-64%)	28	< 0.0001	0.792
SMT	Handpicked	148 (-64%)	28	< 0.0001	4.533
Syntax, Concern	Handpicked	148 (-64%)	28	< 0.0001	0.744
Syntax, SMT, Concern	Handpicked	148 (-64%)	28	< 0.0001	2.446
Syntax, Unstable	Handpicked	107 (-74%)	28	< 0.0001	0.610
All	Handpicked	107 (-74%)	28	< 0.0001	2.328
Naive	Random	42125	2423	0.0064	0.695
No analysis	Random	39449 (-6%)	2423	0.0055 (-14%)	0.766
Syntax	Random	14505 (-65%)	2423	0.0024 (-62%)	0.792
SMT	Random	14505 (-65%)	2423	0.0024 (-62%)	4.533
Syntax, Concern	Random	14505 (-65%)	2423	0.0024 (-62%)	0.744
Syntax, SMT, Concern	Random	14505 (-65%)	2423	0.0024 (-62%)	2.446
Syntax, Unstable	Random	10564 (-74%)	2423	0.0020 (-69%)	0.610
All	Random	10564 (-74%)	2423	0.0020 (-69%)	2.328

Table 20: EPC result overview OIL run loop scheduler

A.1.2 Results recursive OIL scheduler

In the tables below an overview can be found of the results from the different combinations of causal relation gathering approaches together with the recursive OIL scheduler.

The execution times are again not accurate. Take into account that small differences are not always the result of the generated code.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	127	9	< 0.0001	0.710
No analysis	Handpicked	82 (-35%)	9	< 0.0001	0.764
Syntax	Handpicked	68 (-46%)	9	< 0.0001	0.725
SMT	Handpicked	73 (-43%)	9	< 0.0001	5.622
Syntax, Concern	Handpicked	71 (-44%)	9	< 0.0001	0.752
Syntax, SMT, Concern	Handpicked	58 (-54%)	9	< 0.0001	2.768
Syntax, Unstable	Handpicked	67 (-47%)	9	< 0.0001	0.730
All	Handpicked	58 (-54%)	9	< 0.0001	2.615
Naive	Random	5726	240	0.0034	0.710
No analysis	Random	3536 (-38%)	240	0.0017 (-50%)	0.764
Syntax	Random	3188 (-44%)	240	0.0015 (-56%)	0.725
SMT	Random	3292 (-43%)	240	0.0015 (-56%)	5.622
Syntax, Concern	Random	3188 (-44%)	240	0.0015 (-56%)	0.752
Syntax, SMT, Concern	Random	2892 (-49%)	240	0.0013 (-62%)	2.768
Syntax, Unstable	Random	3132 (-45%)	240	0.0015 (-56%)	0.730
All	Random	2892 (-49%)	240	0.0013 (-62%)	2.615

Table 21: Coffee result overview recursive OIL scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	179	22	< 0.0001	0.634
No analysis	Handpicked	179 (-0%)	22	< 0.0001	0.789
Syntax	Handpicked	67 (-63%)	22	< 0.0001	0.681
SMT	Handpicked	67 (-63%)	22	< 0.0001	5.243
Syntax, Concern	Handpicked	67 (-63%)	22	< 0.0001	0.694
Syntax, SMT, Concern	Handpicked	67 (-63%)	22	< 0.0001	3.245
Syntax, Unstable	Handpicked	34 (-81%)	22	< 0.0001	0.805
All	Handpicked	34 (-81%)	22	< 0.0001	3.051
Naive	Random	8518	890	0.0026	0.634
No analysis	Random	8518 (-0%)	890	0.0026 (-0%)	0.789
Syntax	Random	4201 (-51%)	890	0.0012 (-54%)	0.681
SMT	Random	4201 (-51%)	890	0.0012 (-54%)	5.243
Syntax, Concern	Random	4201 (-51%)	890	0.0012 (-54%)	0.694
Syntax, SMT, Concern	Random	4201 (-51%)	890	0.0012 (-54%)	3.245
Syntax, Unstable	Random	2022 (-76%)	890	0.0008 (-69%)	0.805
All	Random	2022 (-76%)	890	0.0008 (-69%)	3.051

Table 22: Game result overview recursive OIL scheduler

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	$<$ 0.0001	0.695
No analysis	Handpicked	413 (-0%)	28	$<$ 0.0001	0.760
Syntax	Handpicked	122 (-70%)	28	$<$ 0.0001	0.748
SMT	Handpicked	122 (-70%)	28	$<$ 0.0001	4.611
Syntax, Concern	Handpicked	122 (-70%)	28	$<$ 0.0001	0.712
Syntax, SMT, Concern	Handpicked	122 (-70%)	28	$<$ 0.0001	2.601
Syntax, Unstable	Handpicked	96 (-76%)	28	$<$ 0.0001	0.791
All	Handpicked	96 (-76%)	28	$<$ 0.0001	2.567
Naive	Random	42125	2423	0.0064	0.695
No analysis	Random	42125 (-0%)	2423	0.0064 (-0%)	0.760
Syntax	Random	11454 (-73%)	2423	0.0021 (-67%)	0.748
SMT	Random	11454 (-73%)	2423	0.0021 (-67%)	4.611
Syntax, Concern	Random	11454 (-73%)	2423	0.0021 (-67%)	0.712
Syntax, SMT, Concern	Random	11454 (-73%)	2423	0.0021 (-67%)	2.601
Syntax, Unstable	Random	8937 (-79%)	2423	0.0019 (-70%)	0.791
All	Random	8937 (-79%)	2423	0.0019 (-70%)	2.567

Table 23: EPC result overview recursive OIL scheduler

A.1.3 Results prioritizing events

In the tables below the results can be found for different prioritization strategies for all the test cases. The recursive OIL scheduler with all causal relation gathering approaches enabled is used for these test cases.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	127	9	< 0.0001	0.710
Keep order	Handpicked	40 (-69%)	9	< 0.0001	2.746
Round-robin	Handpicked	58 (-54%)	9	< 0.0001	2.615
Calculate score	Handpicked	40 (-69%)	9	< 0.0001	2.404
Naive	Random	5726	240	0.0034	0.710
Keep order	Random	2082 (-64%)	240	0.0009 (-74%)	2.746
Round-robin	Random	2892 (-49%)	240	0.0013 (-62%)	2.615
Calculate score	Random	2082 (-64%)	240	0.0009 (-74%)	2.404

Table 24: Coffee result overview prioritizing events

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	179	22	< 0.0001	0.634
Keep order	Handpicked	34 (-81%)	22	< 0.0001	3.051
Round-robin	Handpicked	34 (-81%)	22	< 0.0001	2.975
Calculate score	Handpicked	34 (-81%)	22	< 0.0001	3.051
Naive	Random	8518	890	0.0026	0.634
Keep order	Random	2022 (-76%)	890	0.0008 (-69%)	3.051
Round-robin	Random	2022 (-76%)	890	0.0008 (-69%)	2.975
Calculate score	Random	2022 (-76%)	890	0.0008 (-69%)	2.844

Table 25: Game result overview prioritizing events

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive	Handpicked	413	28	< 0.0001	0.695
Keep order	Handpicked	95 (-77%)	28	< 0.0001	2.498
Round-robin	Handpicked	96 (-76%)	28	< 0.0001	2.567
Calculate score	Handpicked	95 (-77%)	28	< 0.0001	2.456
Naive	Random	42125	2423	0.0064	0.695
Keep order	Random	8937 (-79%)	2423	0.0019 (-70%)	2.489
Round-robin	Random	8937 (-79%)	2423	0.0019 (-70%)	2.567
Calculate score	Random	8937 (-79%)	2423	0.0019 (-70%)	2.456

Table 26: EPC result overview prioritizing events

A.1.4 Results compiler optimizations

In the tables below the results can be found for the test cases with different compiler optimization levels enabled.

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive O0	Random	5726	240	0.0034	0.710
Naive O2	Random	5726	240	0.0014 (-59%)	0.710
Improved run loop O0	Random	3216 (-44%)	240	0.0015 (-56%)	2.464
Improved run loop O2	Random	3216 (-44%)	240	0.0004 (-88%)	2.464
Recursive O0	Random	2082 (-64%)	240	0.0009 (-74%)	2.404
Recursive 02	Random	2082 (-64%)	240	0.0003 (-91%)	2.404

Table 27: Coffee result overview compiler optimizations

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive O0	Random	8518	890	0.0026	0.634
Naive O2	Random	8518	890	0.0007 (-73%)	0.634
Improved run loop O0	Random	2166 (-75%)	890	0.0008 (-69%)	3.436
Improved run loop O2	Random	2166 (-75%)	890	0.0002 (-92%)	3.436
Recursive O0	Random	2022 (-76%)	890	0.0008 (-69%)	2.844
Recursive O2	Random	2022 (-76%)	890	0.0002 (-92%)	2.844

Table 28: Game result overview compiler optimizations

Test	Case	#Tried	#Prod	Exec time[s]	Gen time[s]
Naive O0	Random	42125	2423	0.0064	0.695
Naive O2	Random	42125	2423	0.0021 (-67%)	0.695
Improved run loop O0	Random	10564 (-74%)	2423	0.0020 (-69%)	2.328
Improved run loop O2	Random	10564 (-74%)	2423	0.0006 (-91%)	2.328
Recursive O0	Random	8937 (-79%)	2423	0.0019 (-70%)	2.456
Recursive O2	Random	8937 (-79%)	2423	0.0004 (-94%)	2.456

Table 29: EPC result overview compiler optimizations

A.2 Charts comparing execution time and amount tries

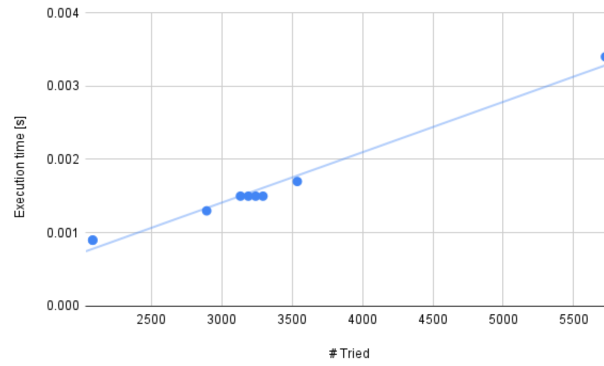

Figure 24: Chart results from Coffee

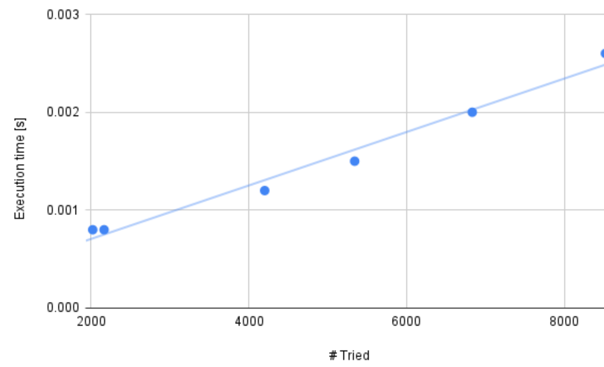

Figure 25: Chart results from Game

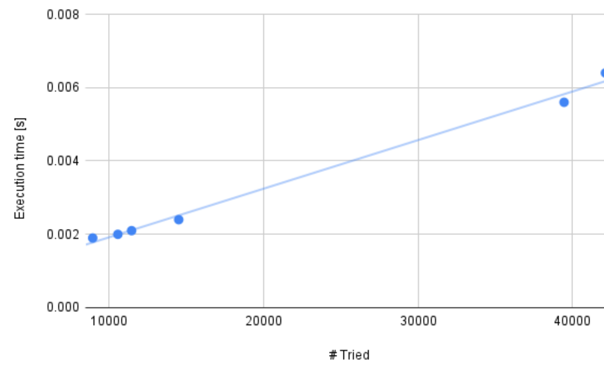

Figure 26: Chart results from EPC

A.3 Proofs

In this chapter, a closer look is taken at the different formal definitions that were supplied earlier. These are used to prove the lemmas for the gathering approaches. This is done for both the transitions associated with T_S and T_{NS}. It is also proven that transitions associated with both T_S^* and T_{NS}^* do not overlap with transitions associated with T_{PS}^*.

A.3.1 Syntax analysis implied transitions

Successor transitions Earlier a definition was supplied for T_S; Definition 1.

Let $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ and $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_S(t) = \{t' \in T_P | \forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))))\}$$

The gathered transitions associated with T_S^{syntax} are defined as follows; Definition 3.

$$T_S^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | ta \sqsubseteq^* so' \wedge gu' = true \wedge \#(\mathcal{CO}(T_{e'})) \leq 1 \wedge\}$$

With this information, we need to show that Lemma 1 always holds.

$$\forall_{t \in T} T_S(t) \supseteq T_S^{syntax}(t)$$

This is logically the same as the following implication.

$$\forall_{(t',t) \in (T_p \times T)}((ta \sqsubseteq^* so' \wedge gu' = true \wedge \#(\mathcal{CO}(T_{e'})) \leq 1 \implies (\forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))))))$$

Assume we have two transitions t and t'. The syntax analysis found $t' \in T_S^{syntax}(t)$. The syntax analysis ensures the following three properties do hold; $ta \sqsubseteq^* so'$, $gu' = true$ and $\#(\mathcal{CO}(T_{e'})) \leq 1$. These are placed in conjunction so they hold individually. Next it is known $t' \in T_P$. With this information, the following needs to be shown to prove that $t' \in T_S(t)$.

$$\forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))))$$

To accomplish this the arbitrary states $s \in S$, $s' \in S$ and parameter evaluation is taken $p \in \mathbb{V}^{\mathcal{PAR}(e)}$ and we assume $s \xrightarrow{e(p)} s'$ and $[\![\mathcal{PRC}(t)]\!](s \cup p)$ holds. Now we need to show the following.

$$\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))$$

Now the arguments of transition t' are taken for the parameter evaluation p'. The defined expressions are acquired using the partial function $\mathcal{ARG}'(\mathcal{PAR}(e'))$. We also pick an $s'' \in S_{Ⓕ}$.

Now we need to show the following.

$$s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p')$$

The next step can be made by looking at the acceptor semantics of a transition in the OIL specification using IOLTS. We know that s' is not the *failure* state as S does not include this state. It follows from the acceptor semantics that $s' \xrightarrow{e'(p')} \iff [\![\mathcal{CC}(e')]\!](s' \cup p')$. Where the evaluations from both the state and the parameters are combined to v' with $v' = p' \cup s'$.

$$[\![\mathcal{CC}(e') \wedge \mathcal{PRC}(t')]\!]v'$$

Which is equal to the following

$$[\![\bigwedge_{c' \in \mathcal{CO}(T_{e'})} \bigvee_{t'' \in T_{e',c'}} \mathcal{PRC}(t'') \wedge \mathcal{PRC}(t')]\!]v'$$

The syntax analysis ensured $\#(\mathcal{CO}(T_{e'})) \leq 1$. So there are two cases, either there are no concerns on the labels of transitions labeled with the event e'. In this case, the conjunction over the concerns holds and we still need to show that the precondition of t' holds. In the other case, there is only one concern on the labels of transitions labeled with the event e'. In this case, we need to show that the disjunction of preconditions holds. However we already need to show that the precondition of t' holds and we know that $t' \in T_{e',c'}$. So we need to show.

$$[\![\mathcal{PRC}(t')]\!]v'$$

This is equal to the following.

$$[\![\mathcal{AC}(so') \wedge gu' \bigwedge \{p = \mathcal{ARG}'(p) | p \in dom(\mathcal{ARG}')\}]\!]v'$$

We already know that $gu' = true$ and we know that for each defined parameter $p = \mathcal{ARG}'(p)$ holds as we picked p' with this in mind. So we need to show.

$$[\![\mathcal{AC}(so')]\!]v'$$

The selected transitions of the syntax analysis ensure $ta \sqsubseteq^* so'$. This $\sqsubseteq^*$ relation is used by $\mathcal{AC}$ as can be seen in Chapter 2.2. The $\sqsubseteq^*$ relation is transitive so the sets of boolean expressions used by $\mathcal{AC}(so')$ is always a subset of the sets gathered for $\mathcal{AC}(ta)$. So $[\![\mathcal{AC}(ta) \implies \mathcal{AC}(so')]\!]v'$. Now we need to show.

$$[\![\mathcal{AC}(ta)]\!]v'$$

It was assumed earlier that $s \xrightarrow{e(p)} s'$ holds. Looking at the acceptor semantics and taking into account that s' is not the failure state as it was selected from S we can assume the following; $[\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$ and $s' = v[\mathcal{U}(T_e^v)]$ holds where $v = s \cup p$. We know that $t \in T_e^v$ as we saw that $[\![\mathcal{PRC}(t)]\!]v$ holds. So we know that $[\![\mathcal{POC}(t)]\!]^v_{s'}$ holds. These postconditions for t look as follows; $[\![\mathcal{AC}(ta) \wedge ar]\!]^v_{s'}$. The area condition does not use old variables so we know that $[\![\mathcal{AC}(ta)]\!]s'$ holds. At last we know that the evaluations in v' include the evaluations in s' therefore $[\![\mathcal{AC}(ta)]\!]v'$ has to hold and the syntax analysis only collects transitions located in T_S.

Therefore Lemma 1 holds.

No successor transitions The following definition was supplied for T_{NS}; see Definition 2.

Let $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ and $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_{NS}(t) = \{t' \in T_P | \forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies \neg(\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p')))))\}$$

The gathered transitions associated with T_{NS}^{syntax} are defined as follows; see Definition 4.

$$T_{NS}^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \exists_{state \in AS(ta)}(\exists_{state' \in AS(so')}(\mathcal{RE}(state) = \mathcal{RE}(state') \wedge \neg(\mathcal{EXP}(state) = \mathcal{EXP}(state'))))\}$$

With this information, we need to show that Lemma 2 always holds.

$$\forall_{t \in T} T_{NS}(t) \supseteq T_{NS}^{syntax}(t)$$

This is logically the same as the following implication.

$$\forall_{((so', gu', e', \mathcal{ARG}', AG', ta', ar'), (so, gu, e, \mathcal{ARG}, AG, ta, ar)) \in (T_p \times T)} ((\exists_{state \in AS(ta)}(\exists_{state' \in AS(so')}(\mathcal{RE}(state) = \mathcal{RE}(state') \wedge \neg(\mathcal{EXP}(state) = \mathcal{EXP}(state'))))) \implies (\forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies \neg(\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p')))))$$

Assume we have two transitions t and t'. The syntax analyis found $t' \in T_{NS}^{syntax}(t)$. The syntax analysis ensures $\exists_{state \in AS(ta)}(\exists_{state' \in AS(so')}(\mathcal{RE}(state) = \mathcal{RE}(state') \wedge \neg(\mathcal{EXP}(state) = \mathcal{EXP}(state')))$ holds. Next it is known $t' \in T_P$. With this information, the following needs to be shown to prove that $t' \in T_{NS}(t)$.

$$\forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies \neg(\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))))$$

To accomplish this arbitrary states $s \in S$, $s' \in S$ and parameter evaluation is taken $p \in \mathbb{V}^{\mathcal{PAR}(e)}$ and we assume $s \xrightarrow{e(p)} s'$ and $[\![\mathcal{PRC}(t)]\!](s \cup p)$ holds. Now we need to show the following.

$$\neg(\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p')))$$

This is logically the same as the following.

$$\forall_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(\neg(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p')))$$

Now we take an arbitrary parameter evaluation $p' \in \mathbb{V}^{\mathcal{PAR}(e)}$ and state $s'' \in S_{Ⓕ}$. The evaluations from both the state and the parameters are combined to v' with $v' = p' \cup s'$.

$$\neg(s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!]v')$$

The next step can be made by looking at the acceptor semantics of a transition in the OIL specification using IOLTS. We know that s' is not the *failure* state as S does not include this state. It follows from the acceptor semantics that $s' \xrightarrow{e'(p')} \iff [\![\mathcal{CC}(e')]\!](s' \cup p')$. Where the evaluations from both the state and the parameters are combined to v' with $v' = p' \cup s'$.

$$\neg([\![\mathcal{CC}(e') \wedge \mathcal{PRC}(t')]\!]v')$$

Which is equal to the following.

$$\neg([\![\bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t'' \in T_{e,c}} \mathcal{PRC}(t'') \wedge \mathcal{PRC}(t')]\!]v')$$

To show this it is sufficient to show that the following does hold.

$$\neg([\![\mathcal{PRC}(t')]\!]v')$$

The syntax analysis finds mismatches in instance variables and values they have to have for a specific area. The function $\mathcal{AC}$ returns a boolean expression that if evaluated shows if an area is active. As there is a mismatch in values for a specific instance variable we know that $\forall_{v \in \mathbb{V}^X}([\![\mathcal{AC}(ta) \implies \neg\mathcal{AC}(so')]\!]v$. It also means that this implication can be expanded to the preconditions; $\forall_{v \in \mathbb{V}^X}([\![\mathcal{AC}(ta) \implies \neg\mathcal{PRC}(t')]\!]v$. This is allowed because $\mathcal{AC}(so')$ is a requirement for $\mathcal{PRC}(t')$ to hold. This implication can be used in our proof so we only have to show the area condition has to hold.

$$[\![\mathcal{AC}(ta)]\!]v'$$

It was assumed earlier that $s \xrightarrow{e(p)} s'$ holds. Looking at the acceptor semantics and taking into account that s' is not the failure state as it was selected from S we can assume the following; $[\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$ and $s' = v[\mathcal{U}(T_e^v)]$ holds where $v = s \cup p$. We know that $t \in T_e^v$ as we saw that $[\![\mathcal{PRC}(t)]\!]v$ holds. So we know that $[\![\mathcal{POC}(t)]\!]^v_{s'}$ holds. These postconditions for t look as follows; $[\![\mathcal{AC}(ta) \wedge ar]\!]^v_{s'}$. The area condition does not use old variables so we know that $[\![\mathcal{AC}(ta)]\!]s'$. At last we know that the evaluations in v' include the evaluations in s' therefore $[\![\mathcal{AC}(ta)]\!]v'$ has to hold and the syntax analysis only collects transitions located in T_{NS}.

Therefore Lemma 2 holds.

No overlap in transition sets The next thing we have to show is that there is no overlap between the transitions associated with T_S, T_{NS}, and T_{PS}. This is denoted with the Lemma 3 and 4.

$$\forall_{t \in T} T_S^{syntax}(t) \cap T_{PS}^{syntax}(t) = \emptyset$$

$$\forall_{t \in T} T_{NS}^{syntax}(t) \cap T_{PS}^{syntax}(t) = \emptyset$$

In the syntax analysis on implied transitions, the different functions were defined as follows in Definition 3, 4 and 5.

$$T_S^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \\ ta \sqsubseteq^* so' \wedge gu' = true \wedge \#(\mathcal{CO}(T_{e'})) \leq 1\}$$

$$T_{NS}^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P \\ \exists_{(targetRegion, targetValue) \in RC(ta)}(\\ \exists_{(sourceRegion, sourceValue) \in RC(so')}(\\ (targetRegion = sourceRegion) \wedge \neg(targetValue = sourceValue)))\}$$

$$T_{PS}^{syntax}((so, gu, e, \mathcal{ARG}, AG, ta, ar)) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \\ ta \sqsubseteq^* so' \wedge \neg(gu' = true \wedge \#(\mathcal{CO}(T_{e'})) \leq 1)\}$$

So we need to show that there is no overlap between the associated transitions. First, we show transitions associated with T_S^{syntax} and T_{PS}^{syntax} do not overlap. This is because either $gu' = true \wedge \#(\mathcal{CO}(T_{e'})) \leq 1$ holds or does not hold. If this part holds the transition is associated with T_S^{syntax} and if it does not hold the transition is associated with T_{PS}^{syntax}. Therefore the associated transitions do not overlap and Lemma 3 holds.

Next, we show that transitions associated with T_{NS}^{syntax} and T_{PS}^{syntax} do not overlap. The syntax analysis for T_{NS}^{syntax} finds mismatches in instance variables and values they have to have for a specific area. The function $\mathcal{AC}$ returns a boolean expression that if evaluated shows if an area is active. As there is a mismatch in values for a specific instance variable we know that $\forall_{v \in \mathbb{V}^X}(\llbracket \mathcal{AC}(ta) \implies \neg \mathcal{AC}(so') \rrbracket v)$. The formal definition of $\mathcal{AC}$ shows that if an area is active all its ancestor areas also have to be active. The ta area is never active at the same time as the so' area. This means that $ta \sqsubseteq^* so'$ never holds for these transitions. Note that T_{PS}^{syntax} requires this $ta \sqsubseteq^* so'$ relation to hold. Therefore the associated transitions do not overlap and Lemma 4 holds.

A.3.2 SMT analysis implied transitions

Successor transitions The following definition was supplied for T_S; see Definition 1.
Let $t = (so, gu, e, \mathcal{ARG}, AG, ta, ar)$ and $t' = (so', gu', e', \mathcal{ARG}', AG', ta', ar')$.

$$T_S(t) = \{t' \in T_P | \forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \land [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \land [\![\mathcal{PRC}(t')]\!](s' \cup p'))))\}$$

The gathered transitions for T_S^{smt} are defined as follows; see Definition 6.

$$T_S^{smt}(t) = \{t' \in T_P | \varphi_S(t, t') \textit{is unsatisfiable} \land \#(\mathcal{CO}(T_{e'})) \leq 1\}$$

With this information, we need to show that Lemma 5 always holds.

$$\forall_{t \in T} T_S(t) \supseteq T_S^{smt}(t)$$

This is logically the same as the following implication.

$$\forall_{((so',gu',e',\mathcal{ARG}',AG',ta',ar'),(so,gu,e,\mathcal{ARG},AG,ta,ar)) \in (T \times T_p)} ((\varphi_S((so, gu, e, \mathcal{ARG}, AG, ta, ar), (so', gu', e', \mathcal{ARG}', AG', ta', ar'))\textit{is unsatisfiable}) \land \#(\mathcal{CO}(T_{e'})) \leq 1) \implies (\forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \land [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \land [\![\mathcal{PRC}(t')]\!](s' \cup p')))))$$

The satisfiability check can be denoted using an existential quantifier together with the semantic brackets to mirror the behavior of the SMT solver. The evaluation u is used which has both evaluations for the old and present instance state. The solver returns satisfiable when there exists a combination of values for the variables that make the satisfiability problem hold. So negating this denotes that a problem is not satisfiable.

$$\forall_{((so',gu',e',\mathcal{ARG}',AG',ta',ar'),(so,gu,e,\mathcal{ARG},AG,ta,ar)) \in (T \times T_p)} ((\neg \exists_{u \in \mathbb{V}^X}([\![\mathcal{AC}(ta) \land ar \land \sigma_{old}(\mathcal{AC}(so) \land gu) \land Assign(AG) \land \neg(\mathcal{AC}(so') \land gu')]\!]u) \land \#(\mathcal{CO}(T_{e'})) \leq 1)) \implies (\forall_{s \in S}(\forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}}((s \xrightarrow{e(p)} s' \land [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{Ⓕ}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}}(s' \xrightarrow{e'(p')} s'' \land [\![\mathcal{PRC}(t')]\!](s' \cup p')))))$$

In the next step, we can now also pick an evaluation v that only contains the evaluations from u in the old instance state and refer to the present instance state by using the evaluation $v[\mathcal{U}(T_e^v)]$. The σ function is removed and the evaluations are used from the old instance state v. The assert can refer to both values from the old instance state and the present one. Therefore a double evaluation is used to also include the old variables. So we end up with the following.

$$\forall_{((so',gu',e',\mathcal{ARG}',AG',ta',ar'),(so,gu,e,\mathcal{ARG},AG,ta,ar))\in(T\times T_p)} \\ ((\neg\exists_{v\in\mathbb{V}^X}((\llbracket \mathcal{AC}(ta)\wedge ar\rrbracket^{v}_{v[\mathcal{U}(T^v_e)]}\wedge\llbracket \mathcal{AC}(so)\wedge gu\rrbracket v)\wedge\neg(\llbracket \mathcal{AC}(so')\wedge gu'\rrbracket v[\mathcal{U}(T^v_e)]))\wedge \\ \#(\mathcal{CO}(T_{e'}))\leq 1)\implies \\ (\forall_{s\in S,s'\in S,p\in\mathbb{V}^{\mathcal{PAR}(e)}}((s\xrightarrow{e(p)}s'\wedge\llbracket \mathcal{PRC}(t)\rrbracket(s\cup p))\implies \\ (\exists_{s''\in S_{Ⓕ},p'\in\mathbb{V}^{\mathcal{PAR}(e')}}(s'\xrightarrow{e'(p')}s''\wedge\llbracket \mathcal{PRC}(t')\rrbracket(s'\cup p')))))$$

The negated existential quantifier can be rewritten to a universal quantifier with a negation in it. Next, this conjunction can then be rewritten to an implication as these are logically the same. Note that there is already a negation located on the right-hand side of the conjunction.

$$\forall_{((so',gu',e',\mathcal{ARG}',AG',ta',ar'),(so,gu,e,\mathcal{ARG},AG,ta,ar))\in(T\times T_p)} \\ ((\forall_{v\in\mathbb{V}^X}(\llbracket \mathcal{AC}(ta)\wedge ar\rrbracket^{v}_{v[\mathcal{U}(T^v_e)]}\wedge\llbracket \mathcal{AC}(so)\wedge gu\rrbracket v\implies\llbracket \mathcal{AC}(so')\wedge gu'\rrbracket v[\mathcal{U}(T^v_e)])\wedge \\ \#(\mathcal{CO}(T_{e'}))\leq 1)\implies \\ (\forall_{s\in S}(\forall_{s\in S,s'\in S,p\in\mathbb{V}^{\mathcal{PAR}(e)}}((s\xrightarrow{e(p)}s'\wedge\llbracket \mathcal{PRC}(t)\rrbracket(s\cup p))\implies \\ (\exists_{s''\in S_{Ⓕ},p'\in\mathbb{V}^{\mathcal{PAR}(e')}}(s'\xrightarrow{e'(p')}s''\wedge\llbracket \mathcal{PRC}(t')\rrbracket(s'\cup p')))))$$

Assume we have two transitions t and t'. The SMT analyis found $t'\in T^{smt}_S(t)$. The SMT analysis only finds causal relations for which t implies the following properties for t'; for all $v\in\mathbb{V}^X$ the following holds $\llbracket \mathcal{AC}(so')\wedge gu'\rrbracket v[\mathcal{U}(T^v_e)]$. Next it ensures $\#(\mathcal{CO}(T_{e'}))\leq 1$ and $t\in T_P$. With this information the following needs to be shown to prove that $t'\in T_S(t)$.

$$\forall_{s\in S,s'\in S,p\in\mathbb{V}^{\mathcal{PAR}(e)}}((s\xrightarrow{e(p)}s'\wedge\llbracket \mathcal{PRC}(t)\rrbracket(s\cup p))\implies \\ (\exists_{s''\in S_{Ⓕ},p'\in\mathbb{V}^{\mathcal{PAR}(e')}}(s'\xrightarrow{e'(p')}s''\wedge\llbracket \mathcal{PRC}(t')\rrbracket(s'\cup p'))))$$

To accomplish this an arbitrary state $s\in S$, $s'\in S$ and parameter evaluation is taken $p\in\mathbb{V}^{\mathcal{PAR}(e)}$ and we assume $s\xrightarrow{e(p)}s'$ and $\llbracket \mathcal{PRC}(t)\rrbracket w$ holds where $w=(s\cup p)$. Now we need to show the following.

$$\exists_{s''\in S_{Ⓕ},p'\in\mathbb{V}^{\mathcal{PAR}(e')}}(s'\xrightarrow{e'(p')}s''\wedge\llbracket \mathcal{PRC}(t')\rrbracket(s'\cup p'))$$

Now the arguments of transition t' are taken for the parameter evaluation p'. The defined expressions are acquired using the partial function $\mathcal{ARG}'(\mathcal{PAR}(e'))$. We also pick an $s''\in S_{Ⓕ}$.

Now we need to show the following.

$$s'\xrightarrow{e'(p')}s''\wedge\llbracket \mathcal{PRC}(t')\rrbracket(s'\cup p')$$

The next step can be made by looking at the acceptor semantics of a transition in the OIL specification using IOLTS. We know that s' is not the $failure$ state as S does not include this state. It follows from the acceptor semantics that $s'\xrightarrow{e'(p')}\iff\llbracket \mathcal{CC}(e')\rrbracket(s'\cup p')$. Where the evaluations from both the state and the parameters are combined to w' with $w'=p'\cup s'$.

$$\llbracket \mathcal{CC}(e')\wedge\mathcal{PRC}(t')\rrbracket w'$$

Which is equal to the following

$$[\![\bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t'' \in T_{e,c}} \mathcal{PRC}(t'') \wedge \mathcal{PRC}(t')]\!] w'$$

The SMT analysis ensured $\#(\mathcal{CO}(T_{e'})) \leq 1$. So there are two cases, either there are no concerns on the labels of transitions labeled with the event e'. In this case, the conjunction over the concerns holds and we still need to show that the precondition of t' holds. In the other case, there is only one concern on the labels of transitions labeled with the event e'. In this case, we need to show that the disjunction of preconditions holds. the event e'. In this case, we need to show that the disjunction of preconditions holds. However we already need to show that the precondition of t' holds and we know that $t' \in T_{e',c'}$. So we need to show.

$$[\![\mathcal{PRC}(t')]\!] w'$$

This is equal to the following.

$$[\![\mathcal{AC}(so') \wedge gu' \bigwedge \{p = \mathcal{ARG}'(p) | p \in dom(\mathcal{ARG}')\}]\!] w'$$

We know that for each defined parameter $p = \mathcal{ARG}'(p)$ holds as we picked p' with this in mind. So we need to show.

$$[\![\mathcal{AC}(so') \wedge gu']\!] w'$$

To show this we take a step back and take a look at the gathered transitions from the SMT analysis. The following was ensured by this SMT analysis.

$$[\![\mathcal{AC}(ta) \wedge ar]\!]^{v}_{v[\mathcal{U}(T^v_e)]} \wedge [\![\mathcal{AC}(so) \wedge gu]\!] v \implies [\![\mathcal{AC}(so') \wedge gu']\!] v[\mathcal{U}(T^v_e)]$$

It was assumed earlier that $s \xrightarrow{e(p)} s'$ holds. Looking at the acceptor semantics and taking into account that s' is not the failure state as it was selected from S we can assume the following; $s' = w[\mathcal{U}(T^w_e)]$, $[\![\mathcal{POC}(T^w_e)]\!]^w_{s'}$ and $[\![\mathcal{CC}(e)]\!] w$. We can use this information to simplify our implication a little bit. We now pick the valuation w as v could be any valuation from $\mathbb{V}^X$.

$$[\![\mathcal{AC}(ta) \wedge ar]\!]^{w}_{s'} \wedge [\![\mathcal{AC}(so) \wedge gu]\!] w \implies [\![\mathcal{AC}(so') \wedge gu']\!] s'$$

First, we earlier assumed that $[\![\mathcal{PRC}(t)]\!] w$ holds. This means the following has to hold $[\![\mathcal{AC}(so) \wedge gu]\!] w$. So we can reduce our implication even further.

$$[\![\mathcal{AC}(ta) \wedge ar]\!]^{w}_{s'} \implies [\![\mathcal{AC}(so') \wedge gu']\!] s'$$

Now we focus on the postcondition that was found earlier $[\![\mathcal{POC}(T^w_e)]\!]^w_{s'}$. We know that $t \in T^w_e$ as we saw that $[\![\mathcal{PRC}(t)]\!] w$ holds. So we know that $[\![\mathcal{POC}(t)]\!]^w_{s'}$ holds. These postconditions for t look as follows; $[\![\mathcal{AC}(ta) \wedge ar]\!]^w_{s'}$. This information can be used to determine the following from our implication.

$$[\![\mathcal{AC}(so') \wedge gu']\!] s'$$

We know that w' has the same evaluations as s' with some additional evaluations from the parameters. So therefore we may conclude that the following also holds finishing our proof.

$$[\![\mathcal{AC}(so') \wedge gu']\!] w'$$

Therefore Lemma 5 holds.

No successor transitions The following definition was supplied for T_{NS}; see Definition 2.

$$T_S(t) = \{t' \in T_P | \forall_{s \in S, s' \in S, p \in \mathbb{V}^{\mathcal{PAR}(e)}} ((s \xrightarrow{e(p)} s' \wedge [\![\mathcal{PRC}(t)]\!](s \cup p)) \implies (\exists_{s'' \in S_{(F)}, p' \in \mathbb{V}^{\mathcal{PAR}(e')}} (s' \xrightarrow{e'(p')} s'' \wedge [\![\mathcal{PRC}(t')]\!](s' \cup p'))))\}$$

The gathered transitions for T_{NS}^{smt} are defined as follows; see Definition 7.

$$T_{NS}^{smt}(t) = \{t' \in T_P | \varphi_{NS}(t, t') \textit{is unsatisfiable}\}$$

With this information, we need to show that Lemma 6 always holds.

$$\forall_{t \in T} T_{NS}(t) \supseteq T_{NS}^{smt}(t)$$

The proof that this lemma is correct is nearly the same as the proof for T_S^{smt}, see Lemma 5. There are two differences as the result of the negation around the existential quantifier in the definition of T_{NS} which carries on through the proof.

First, the following step changes.

$$\neg([\![\bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t'' \in T_{e,c}} \mathcal{PRC}(t'') \wedge \mathcal{PRC}(t')]\!] w')$$

to

$$\neg([\![\mathcal{PRC}(t')]\!] w')$$

Which is now trivial as either $\neg([\![\bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t'' \in T_{e,c}} \mathcal{PRC}(t'')]\!] w'$ or $\neg([\![\mathcal{PRC}(t')]\!] w')$ can result in the complete conjunction not holding.

Next there is the need for proving $\neg[\![\mathcal{AC}(so') \wedge gu']\!] w$. This can be accomplished by using the gathered transitions of the SMT solver because φ_{NS} also gathers the negation compared to φ_S.

$$\varphi_S((so, gu, e, \mathcal{ARG}, AG, ta, ar), (so', gu', e', \mathcal{ARG}', AG', ta', ar')) = \mathcal{AC}(ta) \wedge ar \wedge \sigma_{old}(\mathcal{AC}(so) \wedge gu) \wedge Assign(AG) \wedge \neg(\mathcal{AC}(so') \wedge gu')$$

$$\varphi_{NS}((so, gu, e, \mathcal{ARG}, AG, ta, ar), (so', gu', e', \mathcal{ARG}', AG', ta', ar')) = \mathcal{AC}(ta) \wedge ar \wedge \sigma_{old}(\mathcal{AC}(so) \wedge gu) \wedge Assign(AG) \wedge (\mathcal{AC}(so') \wedge gu')$$

In φ_{NS} the negation is removed around $\mathcal{AC}(so') \wedge gu'$ in comparison to φ_S.

Therefore Lemma 6 holds.

A.3.3 Concern analysis

To show that the concern analysis is correct a portion of the proofs has to be altered. The concern analysis removes the restriction $\#(\mathcal{CO}(T_{e'})) \leq 1$ and adds a post-analysis to ensure the complete analysis is still correct.

Successor transitions The concern analysis ensures that only transitions with the following properties are associated with T_S^{co}; see Definition 8. The $T_S^{*'}$ notation is used to get the gathered transitions without the concern restrictions from either the syntax or SMT analysis on implied transitions.

$$T_S^{co}(t) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \forall_{c \in \mathcal{CO}(T_{e'})} (\exists_{t' \in T_{e',c}} (t' \in T_S^{*'}(t)))\}$$

In the syntax and SMT analysis the $\#(\mathcal{CO}(T_{e'})) \leq 1$ was used in the proofs during the following step. To simplify

$$[\![\bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t'' \in T_{e,c}} \mathcal{PRC}(t'') \wedge \mathcal{PRC}(t')]\!] v'$$

To

$$[\![\mathcal{PRC}(t')]\!] v'$$

So now we need to show that the indicated step in the proof can still be proven with the use of the concern analysis. When this is done correctly we can reuse the remaining parts of the syntax and SMT analysis on implied transitions to prove Lemma 7.

$$\forall_{t \in T} T_S(t) \supseteq T_S^{co}(t)$$

The existential quantifier and universal quantifier in the definition of T_S^{co} closely mimic the conjunction and disjunction in the concern condition. When a transition is found in $T_S^{*'}$ it is known that the preconditions of those transitions hold. The universal and existential quantifier make sure there is at least one transition for each concern that has a precondition that holds. Therefore we know that $[\![\bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t'' \in T_{e,c}} \mathcal{PRC}(t'')]\!] v'$ holds. From this point, the proofs from the syntax and SMT analysis can be used again.

Therefore Lemma 7 holds.

No successor transitions The concern analysis introduces transitions associated with T_{NS}^{co} that have the following properties; see Definition 9.

$$T_{NS}^{co}(t) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \exists_{c \in \mathcal{CO}(T_{e'})} (\forall_{t' \in T_{e',c}} (t' \in T_{NS}^{*}(t)))\}$$

We need to show that Lemma 8 holds.

$$\forall_{t \in T} T_{NS}(t) \supseteq T_{NS}^{co}(t)$$

The previous proofs for both the syntax and SMT analysis on implied transitions can still be used to show that transitions in T_{NS}^{*} are all located in T_{NS}. However, some additional transitions can now be identified. The additionally found transitions are correctly identified as the existential and universal quantification closely mimics the conjunction and disjunction in the concern condition. In the proofs for the syntax and SMT analysis, this looked as follows.

$$\neg([\![\bigwedge_{c \in \mathcal{CO}(T_e)} \bigvee_{t'' \in T_{e,c}} \mathcal{PRC}(t'') \wedge \mathcal{PRC}(t')]\!] v')$$

When the concern analysis identifies a transition in T^{co}_{NS} there is a concern for which this disjunction does not hold meaning that the complete conjunction also does not hold. In this case, the transition is located in T_{NS}.

So therefore Lemma 8 holds.

No overlap in transition sets At last, we need to show that there is no overlap between transitions associated with T^{co}_{S}, T^{co}_{NS} and T^{co}_{PS}. This is denoted with the Lemma 9 and 10.

$$\forall_{t \in T} T^{co}_{S}(t) \cap T^{co}_{PS}(t) = \emptyset$$

$$\forall_{t \in T} T^{co}_{NS}(t) \cap T^{co}_{PS}(t) = \emptyset$$

The definitions are as follows for the gathered transitions for the concern analysis, see Definition 8, 9 and 10.

$$T^{co}_{S}(t) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \forall_{c \in \mathcal{CO}(T_{e'}))} (T_{e',c} \subseteq T^{*}_{S}(t))\}$$

$$T^{co}_{NS}(t) = \{(so', gu', e', \mathcal{ARG}', AG', ta', ar') \in T_P | \exists_{c \in \mathcal{CO}(T_{e'}))} (\forall_{t' \in T_{e',c}} (t' \in T^{*}_{NS}(t)))\}$$

$$T^{co}_{PS}(t) = T^{*}_{S}(t) \setminus (T^{co}_{S}(t) \cup T^{co}_{NS}(t))$$

It is clearly visible that the transitions associated with T^{co}_{PS} have no overlap with either transitions associated with T^{co}_{S} or T^{co}_{NS}. These are explicitly excluded from the transitions associated with T^{co}_{PS}.

Therefore both Lemma 9 and 10 hold.

A.4 Test specification visualizations

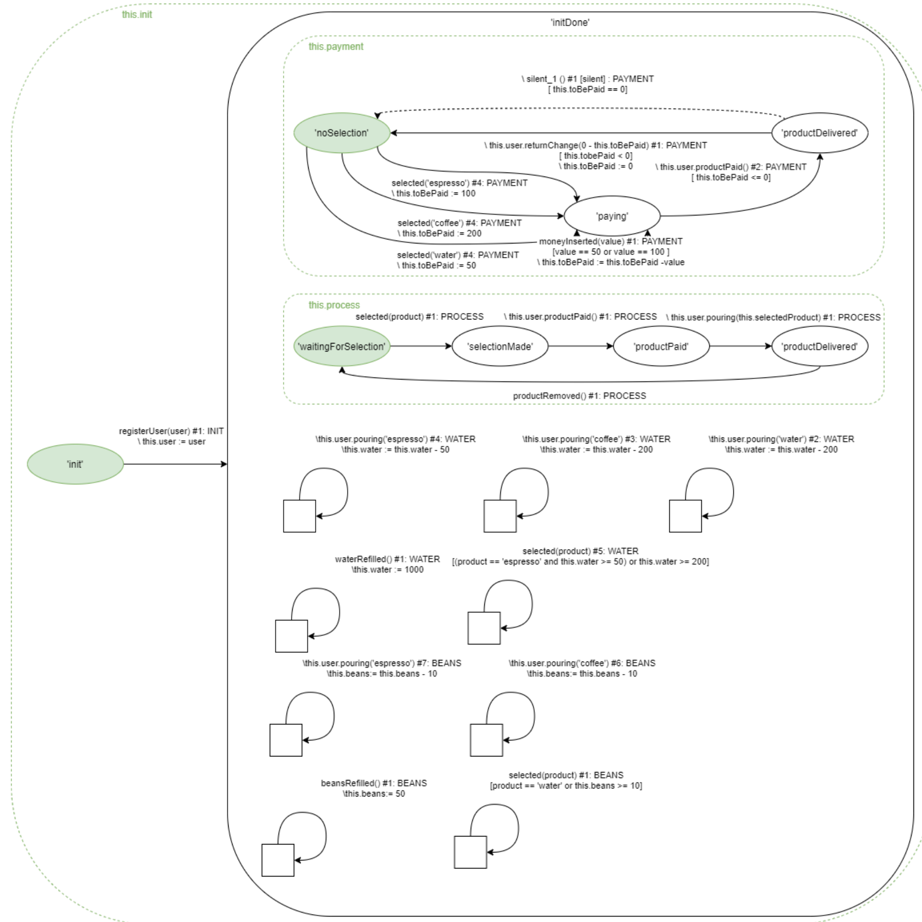

Figure 27: OIL example Coffee

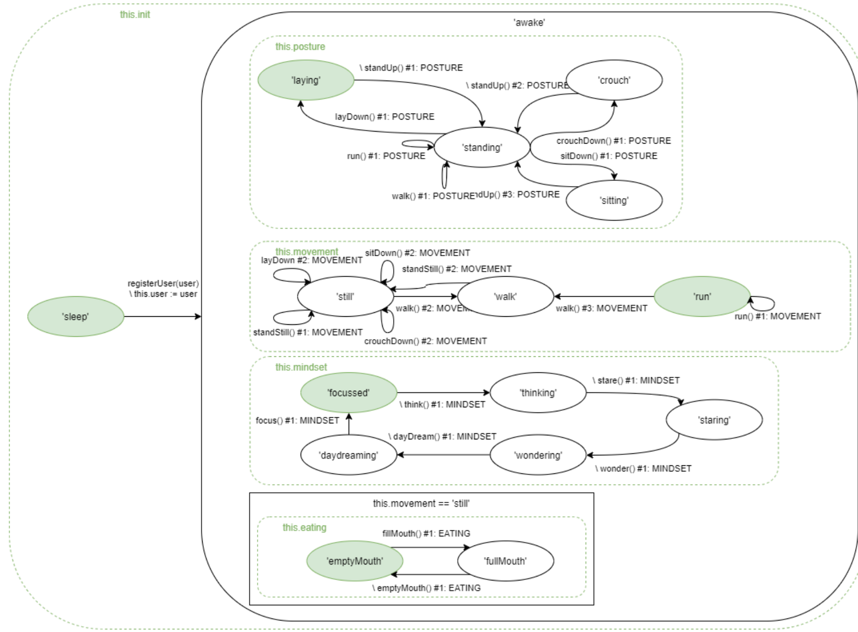

Figure 28: OIL example Game