International Journal on Software Tools for Technology Transfer (2022) 24:441–472
https://doi.org/10.1007/s10009-022-00658-y

GENERAL

Special Issue: FMICS 2019/2020

Formal verification of OIL component specifications using mCRL2

Olav Bunte[1] · Louis C.M. van Gool[2] · Tim A.C. Willemse[1]

Accepted: 8 March 2022 / Published online: 21 April 2022

Abstract
To aid in making software bug-free, several high-tech companies are moving from coding to modelling. In some cases model checking techniques are explored or have already been adopted to get more value from these models. This also holds for Canon Production Printing, where the language OIL was developed for modelling control-software components. In this paper, we present OIL and give its semantics. We define a translation from OIL to mCRL2 to enable the use of model checking techniques. Moreover, we discuss validity requirements on OIL component specifications and show how these can be formalised and verified using model checking. To test the feasibility of these techniques, we apply them to two models of systems used in production.

Keywords Domain specific languages · Model checking · Model transformation · Verification

1 Introduction

To better understand a software system, developers can create abstract models during the design phase. One such model is a behavioural model, which describes the executions of the system. To prove that this model meets the requirements the software should satisfy, one can use model checking, which enables checking of requirements for all executions of the model. While model checking holds great promise, industry so far seems reluctant to adopt the technique. One reason is that most model checking tools build on academic languages, not tailored to the needs of the average engineer.

One company that has shown an interest in using formal methods in the development of their software is Canon Production Printing. Within Canon Production Printing, modelling is a key part of system development for many years already, including the development of domain specific modelling languages [38]. The Open Interaction Language (OIL) is an example of such a language. Its original purpose was to model software interface communication protocols and enable automatic analysis of event log files (trace simulation). Later it has been extended to enable the modelling of control-software components, including the generation of executable code. This has been used to (re)implement several behaviourally complex software components that run on Canon's high-end print systems.

OIL's trace simulation can be used to automatically test a specification by means of a set of pass and fail traces. This is a useful tool to for example reduce risk of regression when the specification evolves. Testing does not always suffice however as several requirements are not feasible to check using testing methods alone. This typically concerns requirements that state the complete absence of some type of undesired behaviour. In this paper, we use OIL as a use case to show how the use of formal methods can help to meet such requirements.

While OIL was designed to have unambiguous semantics, these semantics were previously not formally defined. As a first contribution of this paper, we define a formal operational semantics that corresponds to the behaviour of OIL component specifications. Next we introduce a number of validity requirements over these semantics, ones for which testing is not a feasible approach.

This work was carried out as part of the VOICE-B project, which is funded by Canon Production Printing.

✉ Olav Bunte
o.bunte@tue.nl

Louis C.M. van Gool
louis.vangool@cpp.canon

Tim A.C. Willemse
t.a.c.willemse@tue.nl

[1] Eindhoven University of Technology, Eindhoven, Netherlands

[2] Canon Production Printing, Venlo, Netherlands

Having a formal semantics opens the door to the use of formal methods such as model checking. As our second contribution, we define a translation from OIL component specifications to mCRL2 [22], using the operational semantics as reference. We chose to define both an operational semantics and a translational semantics to separate the concerns of the formalisation of OIL and the translation of OIL to mCRL2. The flexibility of mathematical notation allows the definition of the operational semantics to stay close to the concepts of OIL, while the translational semantics only needs to focus on the translation from mathematical concepts to mCRL2. The target language mCRL2 is supported by a powerful toolset [13] offering model checking and equivalence checking facilities. We have implemented the translation in the Spoofax language workbench [44].

To formally verify the validity requirements on a translated OIL specification, we define the validity requirements in terms of the mu-calculus. For two validity requirements we also define algorithms to check them, as the mu-calculus does not fit these requirements very well. To test the feasibility of our methods, we apply these techniques to some OIL specifications of software components that are used in production at Canon Production Printing.

This paper extends [12] as follows. We previously only described the semantics of OIL, the validity requirements and the translation to mCRL2 informally. In this extended version we define these formally as well. Also, we provide an alternative way for checking two of the validity requirements.

Related work There is a large body of work reporting on the successful application of model checking to industrial cases. These works typically focus on specific business domains, such as for example railway management [3,4,6,9,31,33], automotive [28,29,39,42] and biomedical [26,36]. The modelling languages Statemate, UML and SysML can be used to model systems of any business domain. A lot of research has gone into verification of models written in these languages, see for example [8,17,41], [16,24, 30,35,37,47] and [10,29,45] respectively and the references therein.

Works on modelling control software that are close to ours are those on the FSM language used at CERN [25] and on the Dezyne language developed by the company Verum [7]. The FSM language used at CERN enforces a strict architecture that is tailored to the specific application domain; for general use, this architecture is often too rigid. Using the Dezyne language, a software engineer can model a software system and automatically verify that such a model adheres to the interfaces it uses or implements. Compared to Dezyne, OIL is primarily a modelling language, focussing on ease of use, flexibility and an unambiguous visualisation, whereas Dezyne was designed with verification as the primary focus.

Outline In Sect. 2 we first introduce OIL informally. Then in Sect. 3 we fix some definitions which we then use to define the formal semantics of an OIL specification in Sect. 4. In Sect. 5 we define what it means for an OIL specification to be valid. We give a translation from OIL to mCRL2 specifications in Sect. 6 and show how validity of an OIL specification can be verified in Sect. 7. In Sect. 8 we show the results of some experiments on OIL models of systems used in production. Lastly, we discuss our techniques and results in Sect. 9 and conclude in Sect. 10.

2 An introduction to OIL

OIL (Open Interaction Language) was created by Van Gool as a language to specify, analyse and visualise the (communication) behaviour of control-software systems, partly based on [21]. Using dedicated tooling, one can visualise and analyse OIL specifications. OIL is a textual language, originally based on XML. However, as XML is not very user friendly due its verbosity, a DSL has been designed by Denkers and syntactic sugar was added to OIL [18]. Both the syntax definitions of the XML (OILXML) and the DSL (OILDSL) variants of OIL and the desugaring steps have been implemented in the Spoofax language workbench [44].

While printing is the primary business domain of Canon Production Printing, OIL contains no logic or language constructs specifically tailored to this domain and can therefore also be used in other business domains. Moreover, OIL follows a philosophy of separation of concerns, which helps the engineer to cope with complex behaviour by enabling one to model separate aspects of the system separately in a concise way. This philosophy also allows for a readable and unambiguous visual representation, which is often deemed an indispensable tool in discussions among engineers.

With OIL one can create both *component* and *protocol* specifications. A component specification models the behaviour of a software component, whereas a protocol specification models the desired communication behaviour between components. Although the semantics of both types of specifications is similar, we only focus on component specifications in this paper.

See Fig. 1 for the visualisation of an example OIL specification that models a printer with overheating issues. See A.1 for the corresponding desugared textual OILDSL specification. In the rest of this section we give an informal view of OIL and intuitively explain its main concepts using this specification as running example.

Each OIL component specification consists of a number of *instance variables*, *areas* and *transitions*.

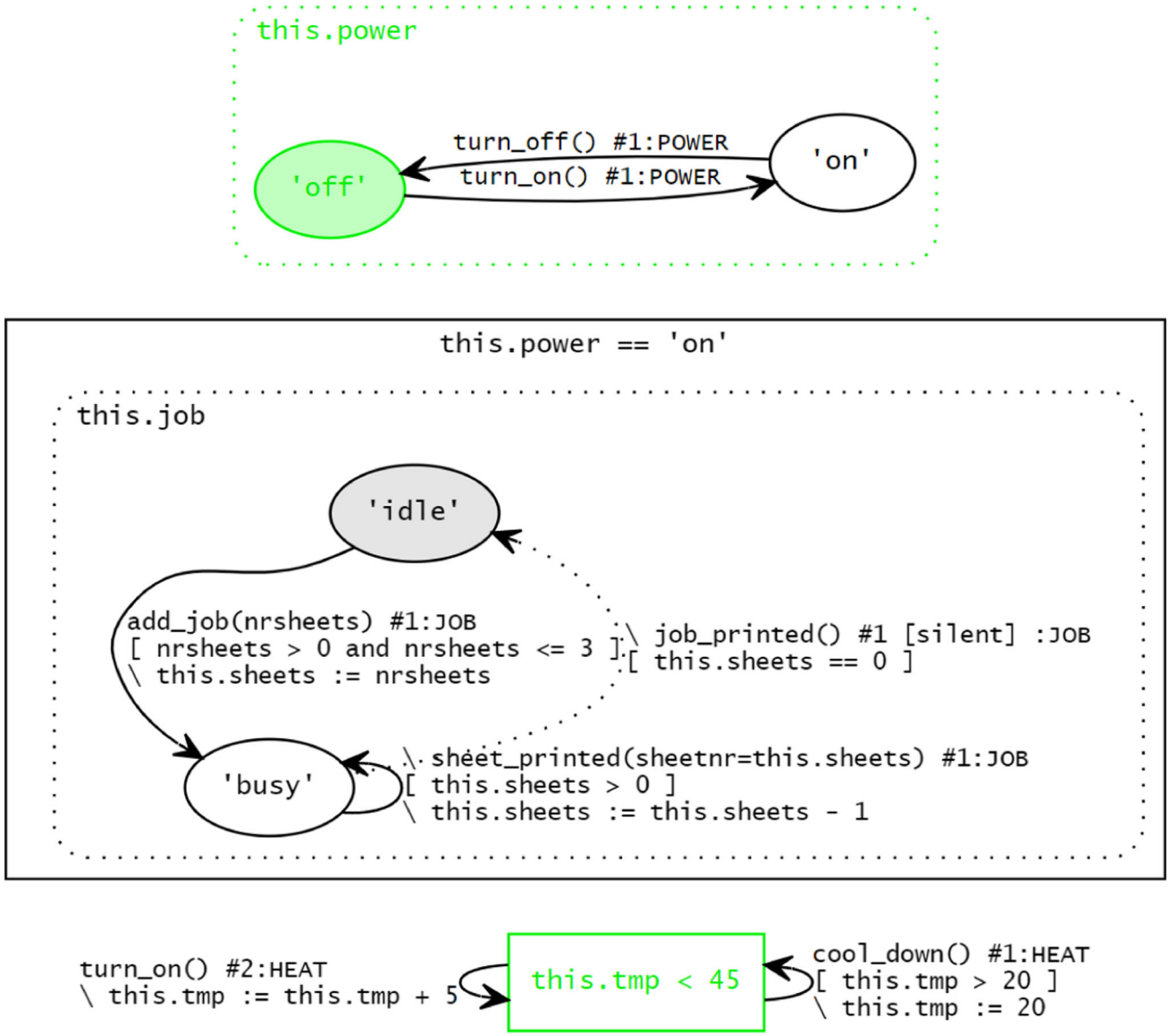

Fig. 1 The visualisation of an example OIL specification that models a simple printer with overheating issues

2.1 Instance variables

Instance variables store the state information of an OIL component specification. We call an instance of such state information an *instance state*, which associates every instance variable with a value. Each instance variable has an initial value, resulting in an initial instance state. In OIL component specifications, instance variables are prepended with the keyword '`this`' to indicate that these belong to the scope of the modelled component instance.

Example 1 The running example defines four instance variables, namely `power`, `job`, `tmp` and `sheets`. They can be found in the visualisation prepended with the keyword `this`, which indicates that this variable is part of the instance state. Instance variable `power` stores whether the component is on using enum values $'off'$ and $'on'$. Instance variable `job` stores whether the component is busy with a print job using enum values $'idle'$ and $'busy'$. Instance variable `tmp` stores the temperature of the component as an integer value. Instance variable `sheets` stores how many sheets are left to print as an integer value. The initial instance state maps `power` to $'off'$, `job` to $'idle'$, `tmp` to 20 and `sheets` to 0. For brevity of notation, we denote such an instance state by $\langle'off', 'idle', 20, 0\rangle$.

2.2 Areas

OIL has three types of areas: *regions*, *states* and *scopes*. A region corresponds to an instance variable and is used to

model behaviour for this variable. Each region contains a number of states which represent values that this variable can have. In the context of the state we refer to this variable as the *variable for this state*. A scope contains a boolean expression that serves as an invariant and is typically used to restrict possible behaviour. Areas are organised as multiple (directed) trees, so an area is either a *root area* or has a *parent*. An area may also have so-called *super areas*, which introduce more parent-child like relations. Super areas relax the strict tree structure to a directed acyclic graph and are typically used for the creation of areas that represent a collection of other areas.

Example 2 The running example has eight areas: two regions, each containing two states, and two scopes. Regions are drawn as dotted boxes, states as ovals and scopes as solid boxes. Areas are directly contained in their parent area. No area in this example has super areas. The two regions refer to the instance variables `power` and `job` and contain states for each value in the domain of these variables. The scope in the middle models that the component may only handle jobs when it is switched on. An alternative way of modelling this restriction would be to make the region that refers to `job` a child of state $'on'$ in the tree structure. The scope on the bottom models that the temperature should stay below 45.

In the visualisation, a state is filled with a colour if the current instance state maps the variable for this state to the value of this state. The visualisation shows the initial instance state and therefore the states with values $'off'$ and $'idle'$ are filled.

Every area is associated with a condition (the *area condition*) and an update (the *area update*). The area condition of an area is a boolean formula. It is true for a given instance state iff it is a root area or the area condition of its parent area is true, in conjunction with the area conditions of its super areas and

- In case the area is a state: the variable for this state equals the value of this state.
- In case the area is a scope: its invariant.

We say that an area is *active* given an instance state iff its area condition is true for this instance state. The area update of an area is a set of assignments to instance variables. It is empty if it is a root area or equal to the area update of its parent, in union with the area updates of its super areas and in case the area is a state, the value of this state is assigned to the variable for this state.

Example 3 In the running example there are three active areas in the initial instance state, coloured green. The region referring to `power` is active since it is a root area. The state with value $'off'$ is active since its parent area is active and the initial instance state maps `power` to $'off'$. The bottom scope is active since it is a root area and its invariant is true for the initial instance state. An example of an area update is the one for state $'off'$, which consists of only one assignment, namely `this.power` $:= 'off'$.

2.3 Events and transitions

An *event* represents the visible behaviour of the system and typically corresponds to a method call. In the context of an OIL component, there are two types of events: *reactive* events, which are received from the environment, and *proactive* events, which are produced by the component itself, either sent to the environment or kept internally (the latter are called *silent* events). Proactive events are also known as *locally controlled events* in the world of IO automata [32]. Like typical methods, events can have parameters which can be used to exchange data between components.

Example 4 The running example has six distinct events: `turn_off`, `turn_on`, `add_job`, `\sheet_printed`, `\job_printed` and `cool_down`. Only events `\sheet_printed` and `\job_printed` are proactive, indicated by the backslash that precedes the event name. Event `\job_printed` is also silent, indicated by `[silent]` in the visualisation. Events `add_job` and `\sheet_printed` have integer parameters `nrsheets` and `sheetnr`, respectively.

Transitions have a *source* and *target* area and an event. Although it is possible, regions are typically not used as source or target since it does not have any added value to do so. Optionally, a transition can have a *guard*, a collection of assignments and an *assert*. If the event of the transition has parameters, the transition may also have *arguments*, which specify fixed values for these parameters.

Example 5 The running example has seven transitions, each drawn as an arrow from its source area to its target area. The event of a transition is the first element in the transition label. This event is followed by a number preceded by a hash symbol, which is used to be able to identify transitions by their event and this number alone. This number is not part of the OIL specification, but generated. The arc of the arrow is dotted if the event of the transition is silent, otherwise it is solid. Below the event, guards are shown between square brackets and assignments are shown following a backslash. The assignments in this example are used to update the instance variables `tmp` and `sheets`. There are no transitions with asserts in this example. Only the transition with event `\sheet_printed` has an argument, which specifies that `sheetnr` must be equal to `this.sheets`.

With every transition we associate a *transition precondition*, a *transition update* and a *transition postcondition*. The transition precondition determines whether the transition can fire and is true iff its source area is active, its guard is true and the values for the event parameters are consistent with the transition's arguments. The transition update defines how the instance state changes whenever this transition fires and consists of the area update of its target area and its assignments. The transition postcondition determines whether the firing of the transition was successful and is true iff its target area is active and its assert is true. If the transition postcondition is false after the transition has fired, we say that the transition has *failed*.

Example 6 For the transition in the running example with event `\sheet_printed`, the transition precondition is $\texttt{this.power} = 'on' \wedge \texttt{this.job} = 'busy' \wedge \texttt{this.sheets} > 0 \wedge \texttt{sheetnr} = \texttt{this.sheets}$, the transition update is $\{\texttt{this.job} := 'busy', \texttt{this.sheets} := \texttt{this.sheets} - 1\}$ and the transition postcondition is $\texttt{this.power} = 'on' \wedge \texttt{this.job} = 'busy'$.

2.4 Updating the instance state

An update of an instance state is triggered by the occurrence of an event. Whenever an event occurs, *all* transitions with this event that can fire, do fire. All updates of transitions that fire are applied simultaneously, resulting in a new instance state. Afterwards, the postconditions of the transitions that fired are checked in this new instance state. If any postcondition is not met (a transition failed), we say that the event fails, resulting in an inconsistent instance state (typically a crash of the component). An event also fails if the transition updates of firing transitions are incompatible, that is if two assignments in these updates assign different values to the same variable.

Note that having two OIL transitions with the same source state and event does not indicate a non-deterministic choice: if both can fire and the event occurs, they fire simultaneously.

Example 7 Suppose that in the initial instance state of the running example the event `turn_on` occurs. This event corresponds to two transitions, identified as `turn_on #1` and `turn_on #2`. Both transitions fire since both transitions' preconditions are true. This causes `turn_on #1` to update `power` to $'on'$ and `turn_on #2` to update `tmp` to $\texttt{tmp}+5$, resulting in instance state $\langle 'on', 'idle', 25, 0\rangle$. In this instance state both transitions' postconditions are true and therefore the event succeeds.

It is possible for an event to fail in the running example. When `turn_on` occurs in instance state $\langle 'off', 'idle', 40, 0\rangle$, both transitions fire which results in instance state $\langle 'on', 'idle', 45, 0\rangle$. Since in this resulting instance state it does not hold that $\texttt{tmp} < 45$, transition `turn_on #2` (and therefore the event `turn_on`) fails. This failure models a crash of the component due to overheating. To make this restriction more explicit to the user of the component, a guard `[this.temp < 40]` can be added to `turn_on #2`.

There are no events with incompatible transition updates in the running example.

2.5 Concerns

As mentioned in the introduction, OIL follows the *separation of concerns* philosophy. This philosophy enables one to model different aspects of a system separately, which helps keeping OIL specifications of complex systems compact. The running example shows this philosophy. There are three different parts visible in the visualisation of the specification that each model a different aspect of the component: the top part models the power aspect, the middle part models the job aspect and the bottom part models the temperature aspect. The separation of concerns philosophy also allows one to easily change the specification if an aspect of the system changes. For instance, if more detailed job handling is required for the running example, the middle part that models the handling of jobs can be easily replaced with a more refined one.

Such separate parts of an OIL specification can interact with each other by means of references to instance variables. For instance, instance variable `power` is referred to by both the region in the top part and the scope in the middle part. Parts can also interact with each other by synchronising on the same event. Synchronisation can occur whenever separate parts of an OIL specification contain transitions with the same event. When these transitions can fire and the corresponding event occurs, the transitions fire simultaneously, causing these separate parts to proceed simultaneously.

We can force such synchronisation, that is make sure that separate parts only proceed with an event if all involved parts can proceed, by restricting the possible combinations of transitions for an event that can fire simultaneously. In OIL this is done by giving transitions one or more *concerns*. Typically, every separate part of an OIL model is associated with a unique concern. We say that an event is part of a concern if one of its transitions has that concern. Then an event may only occur if for each concern this event is part of, at least one of its transitions with that concern can fire. We refer to this as the *concern condition*.

Example 8 In the running example there are three concerns defined, namely `POWER`, `JOB` and `HEAT`, shown after the event in the transition label. The two transitions of event `turn_on` have different concerns, namely `POWER` and `HEAT`, which makes event `turn_on` only allowed if both transitions can fire. This synchronisation enforces that the temperature increases every time the component is turned

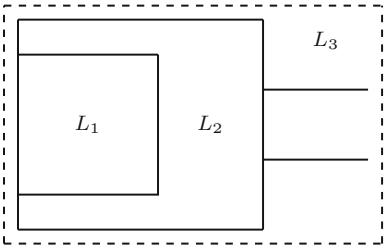

Fig. 2 The three layers of an OIL component. Layer L_1 is the possible behaviour of the component as described by the corresponding OIL specification, layer L_2 is the behaviour of a run-to-completion scheduler that executes L_1 and layer L_3 is the externally visible behaviour of the component

on. If we would not have had these concerns, both transitions could have fired independently of each other. A `turn_on` event could then occur while the component is already on and only increase the temperature of the component.

2.6 Scheduling and communication of events

The execution of an OIL specification is done by a scheduler, which prioritises proactive events over reactive events. Only when there are no proactive events to execute, the scheduler considers reactive events received from the environment. We call this *run-to-completion* semantics. To check which proactive events can be produced by the component, the scheduler checks the concern conditions of all proactive events. If this results in more than one possible proactive event, the scheduler chooses arbitrarily.

Example 9 Since only events `\sheet_printed` and `\job_printed` are proactive, no other event is considered while any of these two are possible when the running example is executed with a run-to-completion scheduler. This causes the printer to not listen to the environment whenever it is busy printing a job. If we would not have a run-to-completion scheduler, it would for instance be possible to turn the printer off while it is busy printing. Note that if we would put scopes around the top region and bottom scope with the invariant $\texttt{job} = 'idle'$, the behaviour with or without run-to-completion scheduler would be the same.

Communication between components is done asynchronously. To realise this, each component has an input FIFO queue in which reactive events are stored that the component receives from the environment. Whenever the scheduler is ready to receive a reactive event, it picks the next one from this queue.

We can view a component as having three layers; see Fig. 2 for a visualisation. The first layer L_1 defines the behaviour that the component is capable of as described by the OIL specification. The second layer L_2 defines the behaviour of the run-to-completion scheduler that receives and executes events consistent with the behaviour defined in layer L_1. Note that L_2 actually has less behaviour than L_1 as run-to-completion only puts restrictions on the behaviour of L_1. The third layer L_3 defines the behaviour of the component as seen from the outside, which includes an input queue to store reactive events and supply them to layer L_2. As we primarily focus on OIL components in isolation, we will only consider layers L_1 and L_2 for the remainder of this paper.

3 Formal preliminaries

Before we introduce the semantics of OIL formally, we need to introduce some definitions concerning updates and transition systems, which we only mentioned informally in the preceding section.

3.1 Valuations and updates

We define $\mathbb{V}$ as the set of all values. Given a set X of variables, a *valuation* over X is a function $X \to \mathbb{V}$ that associates each variable in X with a value. We denote $\mathbb{V}^X$ as the set of all valuations over X.

Definition 1 Let $v \in \mathbb{V}^X$ and $w \in \mathbb{V}^Y$ be valuations over some disjoint sets of variables X and Y. Then the union of v and w is a valuation $v \cup w \in \mathbb{V}^{X \cup Y}$, defined as:

$$(v \cup w)(x) = v(x) \text{ if } x \in X$$
$$(v \cup w)(x) = w(x) \text{ if } x \in Y$$

Definition 2 Let X and X' be sets of variables such that $X' \subseteq X$ and let $v \in \mathbb{V}^X$ be a valuation. Then, we define the restriction $v|_{X'} \in \mathbb{V}^{X'}$ as:

$$v|_{X'}(x) = v(x) \text{ for } x \in X'$$

In an OIL specification one can create expressions from constants, variables and operators to define for instance invariants or guards.

Definition 3 Let X be a set of variables. Then, we define an *expression* f with the following grammar:

$$f ::= c \mid x \mid op(f, \ldots, f)$$

where c is a constant, $x \in X$ a variable and op an n-ary operator for $n > 0$. We define EXP_X as the set of all expressions over variables X.

Given a valuation over the variables in an expression and the interpretation of constants and operators (in boldface), the expression can be evaluated to a single value.

Definition 4 Let X be a set of variables. Then, the *evaluation of an expression* $f \in EXP_X$ given valuation $v \in \mathbb{V}^X$, denoted by $[\![f]\!]v$, is defined as follows:

$$\begin{aligned} [\![c]\!]v &= \mathbf{c} \\ [\![x]\!]v &= v(x) \\ [\![op(f_1, \ldots, f_n)]\!]v &= \mathbf{op}([\![f_1]\!]v, \ldots, [\![f_n]\!]v) \end{aligned}$$

where c is some constant, $\mathbf{c} \in \mathbb{V}$ is the interpretation of c, $x \in X$ is some variable, op is some n-ary operator for $n > 0$ and $\mathbf{op} : \mathbb{V}^n \to \mathbb{V}$ is the interpretation of op.

An expression is *ground* if it does not contain variables. When we evaluate a ground expression, we can leave the valuation out of the notation. For instance, the evaluation of a constant c can be written as $[\![c]\!]$.

A valuation can be changed with an *update*, which is a set of assignments to variables. We assume type correctness of all expressions and assignments in this paper.

Definition 5 Let X be a set of variables. Then, an *update* U over variables X is a set of assignments of the form $x := f$ for $x \in X$ and $f \in EXP_X$.

There is no restriction on how many assignments an update can have for the same variable. However, the application of an update on a valuation can only result in a single value for each variable in the domain of the valuation. If two assignments to the same variable would result in different values, we say that the update is incompatible with the valuation.

Definition 6 Let X be a set of variables, $v \in \mathbb{V}^X$ a valuation and U an update. Update U is *compatible* with v, denoted by $CP(v, U)$, iff for every two assignments $x := f, x := g \in U$ for $x \in X$, it holds that $[\![f]\!]v = [\![g]\!]v$.

For example, the update $\{x := 1, x := 2\}$ is incompatible with any valuation. The update $\{x := x + 2, x := x * 2\}$ is incompatible with $v(x) = 0$ since $0 + 2 \neq 0 * 2$, but it is compatible with $v(x) = 2$ since $2 + 2 = 2 * 2$.

When an update is compatible with a valuation we can apply it to obtain a new valuation.

Definition 7 Let X be a set of variables, $v \in \mathbb{V}^X$ a valuation and U an update compatible with v. Then applying update U on v by means of *simultaneous assignment*, denoted by $v[U]$, results in a new valuation $w \in \mathbb{V}^X$ such that for all $x \in X$:

- for all assignments $x := f \in U$: $w(x) = [\![f]\!]v$,
- in case there exists no assignment $x := f \in U$ for variable x: $w(x) = v(x)$.

To be able to define asserts on transitions in OIL specifications that can reason over both the state before and after the occurrence of an event, we extend expressions with 'old' variables. In such expressions, variables x refer to the state after the event occurred and variables x^{old} refer to the state before the event occurred.

Definition 8 Let X be a set of variables, $f \in EXP_X$ be some expression and $v, w \in \mathbb{V}^X$ two valuations. Then $[\![f]\!]^v_w$ is the evaluation of f using valuation w and the 'old' valuation v, defined as:

$$\begin{aligned} [\![c]\!]^v_w &= \mathbf{c} \\ [\![x]\!]^v_w &= w(x) \\ [\![x^{old}]\!]^v_w &= v(x) \\ [\![op(f_1, \ldots, f_n)]\!]^v_w &= \mathbf{op}([\![f_1]\!]^v_w, \ldots, [\![f_n]\!]^v_w) \end{aligned}$$

where c is some constant, $\mathbf{c} \in \mathbb{V}$ is the interpretation of c, $x \in X$ is some variable, op is some n-ary operator for $n > 0$ and $\mathbf{op} : \mathbb{V}^n \to \mathbb{V}$ is the interpretation of op.

For example, to check whether an integer variable x has increased after applying an update U on a valuation v one can check whether $[\![x > x^{old}]\!]^v_{v[U]}$ results in true. We define $EXP^{old}_X \supseteq EXP_X$ as the set of all expressions over variables X that may include 'old' variables.

3.2 Transition systems

A transition system is a model with states and transitions that models the behaviour of a system.

Definition 9 A *labelled transition system* (LTS) is a tuple $\langle S, s_0, L, \to \rangle$ where S is the set of states, $s_0 \in S$ is the initial state, L is the set of actions and $\to \subseteq S \times L \times S$ is the set of transitions.

For simplicity of notation, we denote $(s, a, s') \in \to$ as $s \xrightarrow{a} s'$. We write $s \xrightarrow{a}$ iff there exists an $s' \in S$ such that $s \xrightarrow{a} s'$ and we write $s \xrightarrow{L'}$ for some $L' \subseteq L$ iff there exists an $a \in L'$ such that $s \xrightarrow{a}$.

We define L^* as the set of *sequences* of actions in L. We write ϵ for the empty sequence and concatenate two sequences with $+$.

Definition 10 Let $\langle S, s_0, L, \to \rangle$ be an LTS. We define $\to^* \subseteq S \times L^* \times S$ as the transition relation over sequences such that for states $s, s' \in S$, action $a \in L$ and sequence $w \in L^*$:

$$s \xrightarrow{\epsilon}{}^* s$$

$$s \xrightarrow{a+w}{}^* s' \text{ iff } \exists_{t \in S} : s \xrightarrow{a} t \wedge t \xrightarrow{w}{}^* s'$$

Often the only states of an LTS that are of interest are states that can be reached via transitions starting from the initial state.

Definition 11 Let $\langle S, s_0, L, \rightarrow \rangle$ be an LTS, $s \in S$ a state and $L' \subseteq L$ a set of action labels. Then, a state $t \in S$ is *reachable from s along* L' iff $\exists_{w \in L'^*} : s \xrightarrow{w}{}^* t$. We define $S_R^{s,L'} \subseteq S$ as the set of all reachable states from s along L'. In case $s = s_0$ and $L' = L$ we abbreviate to S_R.

When one considers a system to be communicating with an environment, it is useful to distinguish between actions that are sent to the system and actions that the system sends itself. To model this distinction, we introduce the notion of an IOLTS. Any definitions on LTSs so far also hold for IOLTSs.

Definition 12 An *input-output labelled transition system* (IOLTS) $\langle S, s_0, I, O, H, \rightarrow \rangle$ is an LTS $\langle S, s_0, I \cup O \cup H, \rightarrow \rangle$ where I is a set of input actions, O is a set of output actions and H is a set of internal actions such that I, O and H are disjoint.

To indicate whether a system is stable and thus waiting for an input, there is the notion of quiescence.

Definition 13 Let $\langle S, s_0, I, O, H, \rightarrow \rangle$ be an IOLTS. A state s is *quiescent* iff $s \xrightarrow{O \cup H}\!\!\!\!\!\!/\;\;$, that is only input actions are enabled in this state. We define $S_\delta \subseteq S$ as the set of all quiescent states.

A single threaded system can only do one thing at a time: either wait for input or create outputs. Such a system is known as an internal choice system [46].

Definition 14 An *internal choice input-output labelled transition system* (IOLTS$^\sqcap$) is an IOLTS where $\forall_{s \in S} : s \xrightarrow{I} \Rightarrow s \in S_\delta$, that is input actions are only enabled in quiescent states.

We say that two (IO)LTSs with initial states s_0 and s_0' are behaviourally equivalent iff their initial states are *bisimilar*, denoted as $s_0 \leftrightarrows s_0'$.

Definition 15 Let $\langle S, s_0, L, \rightarrow \rangle$ be an LTS and $R \subseteq S \times S$ a relation. We say that R is a *strong bisimulation relation* iff it is symmetric and for every $s, t \in S$ such that sRt and for every $a \in L$, if $s \xrightarrow{a} s'$ for some $s' \in S$, then there must exist a $t' \in S$ such that $t \xrightarrow{a} t'$ and $s'Rt'$. We use $\leftrightarrows$ to denote the largest strong bisimulation relation.

4 Formal OIL semantics

In this section, we formally define the semantics of an OIL (component) specification. In Sect. 4.1 we first lay out the formal definition of an OIL specification after which we define its acceptor semantics in the form of an IOLTS that corresponds with layer L_1 in Fig. 2. Lastly in Sect. 4.2 we define the execution semantics of an OIL specification that corresponds with layer L_2 in Fig. 2, in which a run-to-completion scheduler handles the events. We again use the example OIL specification of Fig. 1 as running example.

4.1 Semantics of an OIL component specification

We first formally define the OIL specification itself. This definition is independent of the syntactical representation used for OIL specifications. We typically use italic capital letters to denote sets and calligraphic capital letters to denote functions.

Definition 16 An OIL specification is defined as a tuple $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ where

- $\mathbb{X} = \langle X, \mathcal{I} \rangle$ concerns the variables of the OIL specification, where
  - X is a set of variables. We partition X into a set of instance variables X_I and a set of parameters X_P.
  - $\mathcal{I} \in \mathbb{V}^{X_I}$ associates each instance variable with its initial value.
- $\mathbb{A} = \langle A, \sqsubset, \mathcal{RE}, \mathcal{EXP} \rangle$ concerns the areas of the OIL specification, where
  - A is a set of areas. We partition A into a set of regions A_{Re}, a set of states A_{St} and a set of scopes A_{Sc}.
  - $\sqsubset$ is a partial order over A such that $a \sqsubset a'$ iff a' is the parent area of a or a' is a super area of a.
  - $\mathcal{RE} : A_{St} \rightarrow A_{Re}$ associates each state with the region it belongs to.
  - $\mathcal{EXP} : A \rightarrow EXP_{X_I}$ associates each area a with an expression, which is a variable in X_I in case $a \in A_{Re}$, a constant in EXP in case $a \in A_{St}$ and a boolean expression in EXP_{X_I} in case $a \in A_{Sc}$.
- $\mathbb{T} = \langle E, \mathcal{PAR}, T, CO, \mathcal{CO} \rangle$ concerns the transitions of the OIL specification, where
  - E is a set of events. We partition E into a set of reactive events E_R and a set of proactive events E_P. Additionally, we define $E_H \subseteq E_P$ as the set of silent events.
  - $\mathcal{PAR} : E \rightarrow \mathbb{P}(X_P)$ associates each event with a set of parameters.
  - $T \subseteq A \times EXP_X \times E \times (X_P \nrightarrow EXP_{X_I}) \times \mathbb{P}(X_I \times EXP_X) \times A \times EXP_X^{old}$ is the set of transitions, where $\nrightarrow$ indicates a partial function. For a transition $\langle so, gu, e, \mathcal{ARG}, AG, ta, ar \rangle \in T$, so is its source area, gu is its boolean guard, e is its event, $\mathcal{ARG}$ defines its arguments for parameters in $\mathcal{PAR}(e)$, AG is its collection of assignments (an update), ta is its target area and ar is its boolean assert.
  - CO is a set of concerns.

– $\mathcal{CO} : T \to \mathbb{P}(CO) \setminus \{\emptyset\}$ associates each transition with a non-empty set of concerns. We define $\mathcal{CO}$ also on sets of transitions: let $T' \subseteq T$, then $\mathcal{CO}(T') = \bigcup_{t \in T'} \mathcal{CO}(t)$.

We define $\sqsubseteq^*$ as the reflexive transitive closure of $\sqsubset$. The function $\mathcal{RE}$ follows from the tree structure of the areas in the OIL specification. A state belongs to a region if this region is the closest ancestor region of the state. We assume that the tree structure of areas in the OIL specification is such that for each state such a region exists.

Example 10 Let a_{off} and a_{on} be the states in the running example with values $'off'$ and $'on'$ respectively and let a_{power} be the region around them. Since this region is the parent of the states, we have that $a_{off} \sqsubset a_{power}$ and $a_{on} \sqsubset a_{power}$. Also, since both states belong to this region, we have that $\mathcal{RE}(a_{off}) = a_{power}$ and $\mathcal{RE}(a_{on}) = a_{power}$. Let a_{heat} be the bottom scope in the running example. The function $\mathcal{EXP}$ associates a_{off}, a_{on}, a_{power} and a_{heat} with the expressions $'off'$, $'on'$, `this.power` and `this.tmp` < 45, respectively.

The two transitions identified as `turn_on #1` and `turn_on#2` are defined as $\langle a_{off}, true, \texttt{turn\_on}, \emptyset, \emptyset, a_{on}, true\rangle$ and $\langle a_{heat}, true, \texttt{turn\_on}, \emptyset, \{\texttt{this.tmp} := \texttt{this.tmp} + 5\}, a_{heat}, true\rangle$ respectively. The concerns associated with both transitions by $\mathcal{CO}$ are {`POWER`} and {`HEAT`}, respectively.

See Appendix A.2.1 for the full formal definition of the running example.

Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T}\rangle$ be an OIL specification. For every area $a \in A$ we define its area condition $\mathcal{AC}(a)$ and area update $\mathcal{AU}(a)$. The area condition determines whether an area is active.

Definition 17 The *area condition* $\mathcal{AC}(a)$ of an area $a \in A$ is a boolean expression defined as $\mathcal{AC}(a) = \bigwedge\{\mathcal{EXP}(\mathcal{RE}(a')) = \mathcal{EXP}(a') \mid a' \in A_{St} \wedge a \sqsubseteq^* a'\} \wedge \bigwedge\{\mathcal{EXP}(a') \mid a' \in A_{Sc} \wedge a \sqsubseteq^* a'\}$.

The area update defines changes to the instance state that are necessary for the area to become active. Note that these changes may not be enough as they do not consider invariants of scopes.

Definition 18 The *area update* $\mathcal{AU}(a)$ of an area $a \in A$ is an update defined as $\mathcal{AU}(a) = \{\mathcal{EXP}(\mathcal{RE}(a')) := \mathcal{EXP}(a') \mid a' \in A_{St} \wedge a \sqsubseteq^* a'\}$.

Example 11 The area conditions of areas a_{off}, a_{on} and a_{heat} are defined as `this.power` $= 'off'$, `this.power` $= 'on'$ and `this.tmp` < 45 respectively. A more interesting area condition is that of the state with value $'idle'$ which is defined as `this.power` $= 'on' \wedge$ `this.job` $= 'idle'$. The area updates of areas a_{off}, a_{on} and a_{heat} are defined as $\{\texttt{this.power} := 'off'\}$, $\{\texttt{this.power} := 'on'\}$ and $\emptyset$, respectively.

For every transition $t \in T$ we define its transition precondition $\mathcal{PRC}(t)$, transition update $\mathcal{U}(t)$ and transition postcondition $\mathcal{POC}(t)$. The transition precondition determines whether a transition can fire, which depends on the source area condition $\mathcal{AC}(so)$, guard gu and arguments $\mathcal{ARG}$ of the transition.

Definition 19 Let $t = \langle so, gu, e, \mathcal{ARG}, AG, ta, ar\rangle \in T$ be a transition. Then its *transition precondition* $\mathcal{PRC}(t)$ is a boolean expression defined as $\mathcal{PRC}(t) = \mathcal{AC}(so) \wedge gu \wedge \bigwedge\{p = \mathcal{ARG}(p) \mid p \in dom(\mathcal{ARG})\}$.

The transition update indicates how the instance state changes when this transition fires, which depends on the target area update $\mathcal{AU}(ta)$ and assignments AG of the transition.

Definition 20 Let $t = \langle so, gu, e, \mathcal{ARG}, AG, ta, ar\rangle \in T$ be a transition. Then its *transition update* $\mathcal{U}(t)$ is an update defined as $\mathcal{U}(t) = \mathcal{AU}(ta) \cup AG$. For $T' \subseteq T$ a set of transitions, we define $\mathcal{U}(T') = \bigcup_{t \in T'} \mathcal{U}(t)$.

The transition postcondition must be true after a transition has fired, otherwise the transition has failed and we have arrived in an inconsistent state. It depends on the target area condition $\mathcal{AC}(ta)$ and assert ar of the transition.

Definition 21 Let $t = \langle so, gu, e, \mathcal{ARG}, AG, ta, ar\rangle \in T$ be a transition. Then its *transition postcondition* $\mathcal{POC}(t)$ is a boolean expression defined as $\mathcal{POC}(t) = \mathcal{AC}(ta) \wedge ar$. For $T' \subseteq T$ a set of transitions, we define $\mathcal{POC}(T') = \bigwedge_{t \in T'} \mathcal{POC}(t)$.

Example 12 The transition preconditions for transitions `turn_on #1` and `turn_on #2` are defined as `this.power` $= 'off'$ and `this.tmp` < 45 respectively. A more interesting transition precondition is that of the transition with event `\job_printed`, which equals `this.power` $= 'on' \wedge$ `this.job` $= 'busy' \wedge$ `this.sheets` $= 0$. The transition updates for transitions `turn_on #1` and `turn_on #2` are defined as $\{\texttt{this.power} := 'on'\}$ and $\{\texttt{this.tmp} := \texttt{this.tmp}+5\}$ respectively. The transition postconditions for transitions `turn_on #1` and `turn_on #2` are defined as `this.power` $= 'on'$ and `this.tmp` < 45, respectively.

The states of the transition system are the instance states of the OIL specification, which are valuations over X_I. A transition in the transition system corresponds to the occurrence of an event. Each event e corresponds to a set of OIL transitions

T_e, defined as $T_e = \{\langle so, gu, e', \mathcal{ARG}, AG, ta, ar, CO\rangle \in T \mid e' = e\}$. For e to be allowed, a transition must be able to fire for each concern that the event is part of. This restriction is enforced by the concern condition.

Definition 22 Let $e \in E$ be an event. Let $T_{e,c} \subseteq T_e$ be the set of transitions of event e that have concern c, defined as $T_{e,c} = \{t \in T_e \mid c \in \mathcal{CO}(t)\}$. Then, the *concern condition* $\mathcal{CC}(e)$ is a boolean expression defined as:

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

Example 13 Let $e =$ `turn_on`. Then T_e is the set with the two transitions identified as `turn_on #1` and `turn_on #2`. As mentioned previously in Example 8 and Example 10, these two transitions have different concerns. For this event the concern condition is then defined as $CC(e) =$ `this.power` $= 'off' \wedge$ `this.tmp` < 45. As the concern condition must be true for an event to be allowed to occur, `turn_on` may only occur if both transitions can fire.

To associate parameters of an event with values we use a valuation over these parameters. Given an event $e \in E$ and a valuation $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, we write $e(p)$ as the event e with values for its parameters according to valuation p. In case $\mathcal{PAR}(e) = \emptyset$ there is only one such p (the empty valuation).

It depends on the current instance state and on values for parameters which transitions of an event e can actually fire. Given a valuation $v \in \mathbb{V}^X$, T_e^v is the set of transitions of event e that can fire, defined as $T_e^v = \{t \in T_e \mid [\![\mathcal{PRC}(t)]\!]v\}$. Whenever event e occurs, all OIL transitions in T_e^v fire and apply their transition updates simultaneously. After the updates have been applied, we need to check whether the event succeeded. If not, we arrive in a *failure state* denoted as Ⓕ. This is described in the acceptor semantics of an OIL specification defined below.

Definition 23 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T}\rangle$ be an OIL specification. Then the *acceptor semantics* of the OIL specification is given by the IOLTS $\langle S, s_0, I, O, H, \rightarrow\rangle$, where

- $S = \mathbb{V}^{X_I}$, $S_Ⓕ = S \cup \{Ⓕ\}$,
- $s_0 = \mathcal{I}$,
- $I = \{e(p) \mid e \in E_R \wedge p \in \mathbb{V}^{\mathcal{PAR}(e)}\}$, $O = \{e(p) \mid e \in E_P \setminus E_H \wedge p \in \mathbb{V}^{\mathcal{PAR}(e)}\}$, $H = \{e(p) \mid e \in E_H \wedge p \in \mathbb{V}^{\mathcal{PAR}(e)}\}$, $L = I \cup O \cup H$,
- $\rightarrow \subseteq S_Ⓕ \times L \cup$ `fail` $\times S_Ⓕ$ such that for all $s, s' \in S$ and $e(p) \in L$, with $v = s \cup p$:

$$s \xrightarrow{e(p)} s' \text{ iff } [\![\mathcal{CC}(e)]\!]v \wedge CP(v, \mathcal{U}(T_e^v)) \wedge s' = v[\mathcal{U}(T_e^v)]|_{X_I} \wedge [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$$

$$s \xrightarrow{e(p)} Ⓕ \text{ iff } [\![\mathcal{CC}(e)]\!]v \wedge (\neg CP(v, \mathcal{U}(T_e^v)) \vee \neg [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]})$$

$$Ⓕ \xrightarrow{\texttt{fail}} Ⓕ$$

The failure of an event is explicitly modelled using failure state Ⓕ with a self loop with action `fail` to indicate that a failure occurred. An event fails if the update is incompatible ($\neg CP(v, \mathcal{U}(T_e^v))$) or if the transition postconditions are not met ($\neg [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$). Note that this IOLTS is deterministic. This is because all transitions in OIL with the same event that can fire are combined into one transition in the IOLTS.

Lemma 1 *Let $\langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS that describes the acceptor semantics of some OIL specification. Then for all $s, s', s'' \in S$ and $a \in L$ where $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$, we have that $s' = s''$.*

Also note that in case an instance variable has an infinite domain (such as an integer variable), the state space may be infinite. Similarly, in case a parameter without an argument has an infinite domain (such as an integer parameter), the transition system may be infinitely branching.

Example 14 The IOLTS that describes the acceptor semantics of the running example has 51 states and 126 transitions. Due to its size we will not show this IOLTS, but in the following subsection we will show the IOLTS of the so-called *execution semantics* of the running example instead.

4.2 Execution semantics

The IOLTS in Definition 23 describes the behaviour a component is capable of, that is the behaviour of layer L_1 in Fig. 2. To execute this behaviour (layer L_2) a scheduler is needed. As mentioned previously in Section 2, the scheduler used for OIL components has run-to-completion semantics, which prioritises proactive events over reactive events. This puts some restrictions on the possible behaviour of the component.

Definition 24 Let $M = \langle S, s_0, I, O, H, \rightarrow\rangle$ be an IOLTS. Then $\gamma(M)$ is the behaviour of a *run-to-completion scheduler* over M, defined as $\gamma(M) = \langle S, s_0, I, O, H, \rightarrow'\rangle$ where $\rightarrow' \subseteq \rightarrow$ such that for all $s, s' \in S_Ⓕ$ and $a \in L \cup \{$`fail`$\}$: $s \xrightarrow{a}{}' s'$ iff $a \in I \Rightarrow s \xrightarrow{O \cup H}{\!\!\!\!\!\!/}$.

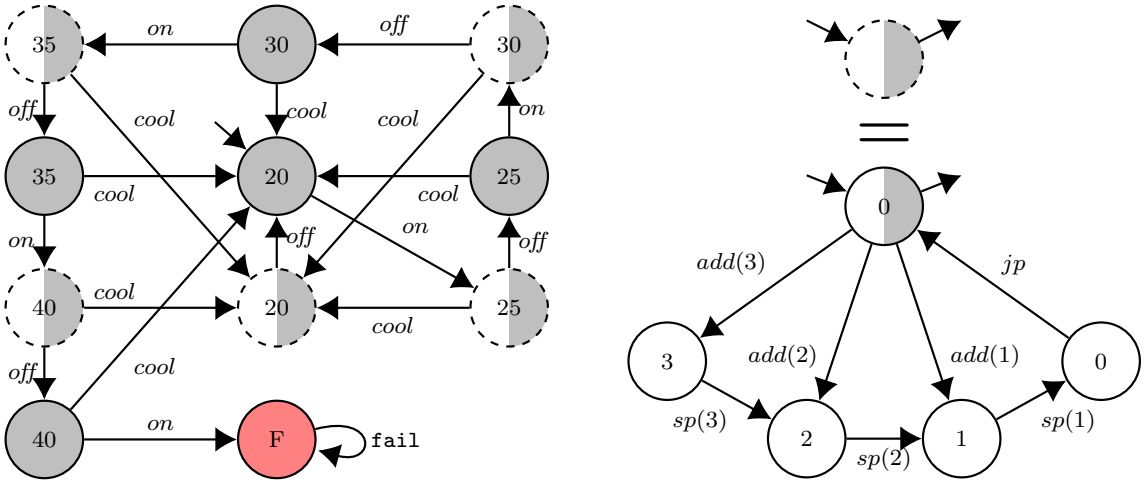

Fig. 3 The transition system that describes the execution semantics of the OIL specification visualised in Fig. 1. The left figure shows the transition system and the right figure expands on dashed states. The left half of a state is gray iff `power` $= 'off'$ and the right half of a state is gray iff `job` $= 'idle'$. In the left figure, the value written in the state is the value of `tmp`. The value of `sheets` in these states equals 0. In the right figure the value written in the state is the value of `sheets`. The value of `tmp` in these states is the same as the state in the left figure that is expanded. The red state with label F is the failure state. Action label *on* refers to event `turn_on`, *off* to `turn_off`, *add* to `add_job`, *sp* to `\sheet_printed`, *jp* to `\job_printed` and *cool* to `cool_down`

In case M is the IOLTS that describes the acceptor semantics of some OIL specification according to Definition 23, we say that $\gamma(M)$ describes the *execution semantics* of this OIL specification.

The execution semantics is internal choice due to the run-to-completion semantics of the scheduler. As the scheduler prioritises proactive over reactive events, reactive events are only enabled whenever no proactive events are enabled, that is when quiescence can be observed, which is according to the definition of an IOLTS$^{\sqcap}$ (Definition 14).

Lemma 2 *Let M be some IOLTS. Then $\gamma(M)$ is an IOLTS$^{\sqcap}$.*

Example 15 The IOLTS$^{\sqcap}$ that describes the execution semantics of the running example has 31 states and 54 transitions. See Fig. 3 for a visualisation of this IOLTS$^{\sqcap}$.

5 Validity of OIL specifications

To avoid undesirable behaviour of the scheduled component, we introduce a number of requirements on an OIL specification. If at least one of these requirements is not met, we say that the OIL specification is *invalid*. Note that since all validity requirements are about the execution of a model, we only (need to) consider the reachable states. Let $\langle S, s_0, I, O, H, \rightarrow \rangle$ be an IOLTS$^{\sqcap}$ that describes the execution semantics of an OIL specification.

When the scheduler checks what proactive events it can produce, it only checks the concern condition of these proactive events (as mentioned at the end of Section 2). Checking whether an event can fail would require to execute the event, check the postcondition and then roll back to the original state. As this might need to be done for many proactive events, this may be computationally very expensive and is therefore undesirable. Still, we would not want that a scheduler may actively crash the system by producing a failing proactive event. We do allow reactive events to fail, as this indicates misuse of the component by the environment. To prevent the scheduler from producing a failing proactive event, we have the following requirement:

Requirement 1 (Safe lookaheadlessness) *Proactive actions cannot fail. More formally, requirement R1 is defined as:*

$$\neg\exists_{s \in S_R, e(p) \in O \cup H} : s \xrightarrow{e(p)} \textcircled{F}$$

Due to the run-to-completion semantics of the scheduler, proactive events have priority over reactive events. If a component would contain an infinite path of proactive events, such as a loop, the scheduler would never consider a reactive event any more once it enters this path. This would result in a component that never reacts to events from the environment. To ensure that a component can eventually engage in communication, we have the following requirement:

Requirement 2 (Finite proactivity) *Any sequence of proactive events must be finite. More formally, requirement R2 is defined as:*

$$\neg\exists_{s \in S_R, u \in (O \cup H)^\omega} : s \xrightarrow{u} {}^{\omega}$$

where $(O \cup H)^\omega$ is the set of infinite sequences over the set of action labels $O \cup H$ and $s \xrightarrow{u}{}^\omega$ iff there exists an infinite path starting in state s that is consistent with sequence u.

When the scheduler has the choice between multiple proactive events, there are multiple routes of proactive events the scheduler can take until it reaches a quiescent state. Since the scheduler chooses between proactive events arbitrarily, the choice between these routes is non-deterministic. If some of these routes would end up in different quiescent states, this non-determinism may permeate the whole component, which is undesired. To prevent the choice of the scheduler affecting the instance state after having run to completion, we have the following requirement:

Requirement 3 (Confluent proactivity) *All possible sequences of proactive events from a state that end up in a quiescent state, end up in the behaviourally same state. More formally, requirement R3 is defined as:*

$$\forall_{s \in S_R, w, w' \in (O \cup H)^*, t, t' \in S_\delta} : s \xrightarrow{w}{}^* t \wedge s \xrightarrow{w'}{}^* t' \Rightarrow t \leftrightarrows t'$$

Lastly, it may be the case that some possible routes that the scheduler can take, consist of different events. This would mean that whether an event is produced or not is determined non-deterministically. This is especially undesired for proactive events, as these may be needed for other components to proceed. The scheduler is free to choose the order in which the event are produced however. To prevent the choice of the scheduler affecting what proactive events will be produced, we have the following requirement:

Requirement 4 (Predictable proactivity) *All possible sequences of proactive events from a state that end up in a quiescent state, consist of the same multiset of events. More formally, requirement R4 is defined as:*

$$\forall_{s \in S_R, w, w' \in (O \cup H)^*, t, t' \in S_\delta} : s \xrightarrow{w}{}^* t \wedge s \xrightarrow{w'}{}^* t' \Rightarrow w \approx w'$$

where $w \approx w'$ iff w and w' have the same multiset of actions.

6 Translating OIL to mCRL2

To verify the above requirements on an OIL specification, or any requirement for that matter, we can make use of model checking techniques [14]. To avoid reimplementing the wheel, we can largely reuse the model checking capabilities of the mCRL2 tool set [13] in the context of OIL by creating a translation from OIL specifications to mCRL2 specifications. We first elaborate on mCRL2, after which we describe how OIL is translated to mCRL2. Afterwards we describe how we have implemented this translation so that it can be applied in practice. The proofs of lemmas and theorems presented in this section can be found in Appendix B.1.

6.1 mCRL2

The language mCRL2 [22] is a behaviour modelling language based on process algebra. Every mCRL2 specification consists of two parts: a data specification and a process specification. The data specification typically contains type definitions and definitions of mappings by means of rewrite rules. The process specification contains definitions of actions and of one or more processes, which use these actions to describe behaviour.

In the context of mCRL2, we typically reason with vectors of variables instead of sets of variables. We denote a vector with a bar on top and use indexing for projection such that $\bar{x} = x_1, .., x_n$. When applicable, we may use the notation $\bar{x}$ to denote the set $\{x_1, .., x_n\}$ of all variables in $\bar{x}$.

Each mCRL2 specification can be (automatically) rewritten to a normal form called a Linear Process Specification (LPS). We use the latter format as the target for our translation. Each LPS contains exactly one process in the form of a linear process equation.

Definition 25 Let L be a set of actions. A Linear Process Equation (LPE) is of the following form:

$$P(\bar{d} : D) = \sum_{i \in I} \sum_{\bar{e_i} : E_i} c_i \rightarrow a_i(\bar{f_i}) \cdot P(\bar{g_i})$$

where P is a process name, I is some index set, $\bar{d}$ and $\bar{e_i}$ are vectors of variables, D and E_i are data types, c_i is a boolean expression, $a_i \in L$ is an action, $\bar{f_i}$ is a vector of expressions that gives values for the parameters for a_i and $\bar{g_i}$ is a vector of expressions that represents the next state. Expression c_i and expressions in $\bar{f_i}$ and $\bar{g_i}$ can depend on variables in $\bar{d}$ and $\bar{e_i}$.

From an LPS an LTS can be easily extracted.

Definition 26 Let there be an LPS with an LPE that defines a process $P(\bar{d} : D)$ as in Definition 25. Let $\bar{id}$ be a vector of ground expressions in EXP that represents the initial value of $\bar{d}$. Then, the process expression $P(\bar{id})$ corresponds to the LTS $\langle S, s_0, L, \rightarrow \rangle$ where

- $S = \mathbb{V}^{\bar{d}}$,
- $s_0 \in \mathbb{V}^{\bar{d}}$ such that $s_0(d_j) = [\![id_j]\!]$ for each $1 \leq j \leq n$,
- $\rightarrow$ is the transition relation such that for all $s \in S, i \in I$ and $p \in \mathbb{V}^{\bar{e_i}}$ with $v = s \cup p$:

$$s \xrightarrow{a_i([\![\bar{f_i}]\!]v)} [\![\bar{g_i}]\!]v \text{ iff } [\![c_i]\!]v$$

The language mCRL2 also comes with a tool set with which one can apply numerous model checking techniques on mCRL2 specifications [13]. See Fig. 4 for the basic work

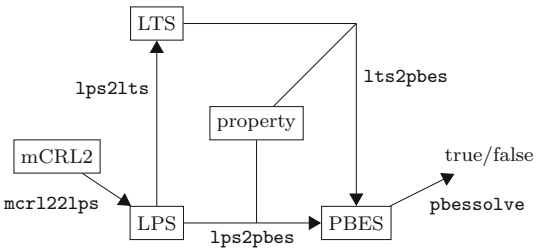

Fig. 4 The basic work flows in the mCRL2 tool set for generating an LTS and for checking a mu-calculus property. The edges are labelled with tool names

flows for generating the LTS and verifying properties defined with the mu-calculus.

6.2 OIL in mCRL2

In this subsection, we define the translation from an OIL specification to an mCRL2 specification. This translation depends on multiple definitions from Section 4 that were also used to define the acceptor semantics of OIL in terms of an IOLTS (Definition 23). We again use the example OIL specification of Fig. 1 as running example.

To be able to represent instance states, we define a structured sort `ISt` in the mCRL2 data specification, which defines a constructor `IS` that accepts a tuple of expressions, representing values for the instance variables. We also add (data) type definitions of (instance) variables where necessary. Along with this structured sort we define projection functions $\texttt{GET}_x$ to query the value of an instance variable x. We call an expression of type `ISt` an *instance struct*. In mCRL2, an instance state is then represented with a ground instance struct, that is an instance struct without variables.

Example 16 For the running example we define the instance state type as follows:

```
ISt = struct IS(
  GET_power :power_type,
  GET_job :job_type,
  GET_tmp :Int, GET_sheets :Int);
```

where the types `power_type` and `job_type` are defined separately. The initial instance state $\langle 'off', 'idle', 20, 0\rangle$ is represented in mCRL2 as the ground instance struct `IS(power_off, job_idle, 20, 0)`.

To translate an expression $f \in EXP_X$ to an mCRL2 expression, we define $\sigma_{\texttt{s}}(f)$ for some instance struct `s`, which translates each constant and operator to its mCRL2 counterpart and each $x \in X_I$ to $\texttt{GET}_x(\texttt{s})$. In case $f \in EXP_X^{old}$ we define $\sigma^{\texttt{s}}_{\texttt{us}}(f)$ for instance structs `s` and `us`, which translates each constant and operator to its mCRL2 counterpart and for each $x \in X_I$, x^{old} to $\texttt{GET}_x(\texttt{s})$ and x to $\texttt{GET}_x(\texttt{us})$. To translate the evaluation of expressions to the context of mCRL2 too, we need to translate an evaluation over instance variables to a valuation over an instance struct variable. Given a valuation $s \in \mathbb{V}^{X_I}$ and an instance struct variable `s`, we define $s_{\texttt{s}}$ as the valuation over `s` such that $[\![\texttt{GET}_x(\texttt{s})]\!]s_{\texttt{s}} = s(x)$ for all $x \in X_I$.

As mentioned before, the global state of an OIL component changes by the application of an update. To formalise this in mCRL2, we first define a setter map $\texttt{SET}_x : \texttt{ISt} \times \texttt{Bool} \times \texttt{T} \to \texttt{ISt}$ with corresponding rewrite rules for each instance variable x, where `T` is the data type of x. The first parameter of type `ISt` is the instance struct to be updated, the second parameter is a boolean expression that indicates whether the change should be applied and the third parameter of type `T` is the new value for x. The boolean parameter effectively makes this a *conditional assignment*. If this boolean parameter evaluates to true, the entry for x in the instance struct is overwritten with the new value, otherwise no changes are made. Why this boolean parameter is useful will be shown later on.

Definition 27 Let the variables in X_I be indexed such that $X_I = \{x_1, .., x_n\}$. Let $x_i \in X_I$ be some instance variable, `s` some instance struct and `f` and $\texttt{g}_1, .., \texttt{g}_n$ some mCRL2 expressions. Then, $\texttt{SET}_{x_i}$ is defined with the following rewrite rules:

$$\texttt{SET}_{x_i}(\texttt{s}, \texttt{false}, \texttt{f}) = \texttt{s}$$
$$\texttt{SET}_{x_i}(\texttt{IS}(\texttt{g}_1, .., \texttt{g}_i, .., \texttt{g}_n), \texttt{true}, \texttt{f}) = \texttt{IS}(\texttt{g}_1, .., \texttt{f}, .., \texttt{g}_n)$$

The result of a setter is an instance struct of expressions. Note that `f` may contain variables and therefore the resulting instance struct too. If it does, it depends on a valuation for these variables which instance state the instance struct actually represents.

Example 17 To update the variable `power` of the running example, we define rewrite rules of the form:

```
SET_power(s, false, u_power) = s;
SET_power(IS(power, job, tmp, sheets),
          true, u_power) =
  IS(u_power, job, tmp, sheets);
```

To update an instance struct `s` with assignment `tmp` := `tmp` + 5 we can use the mCRL2 expression `SET_tmp(s, true, GET_tmp(s) + 5)`.

Whenever an event occurs, all transitions for this event that can fire are involved in updating the instance state. Instead of applying the assignments of the update simultaneously, in mCRL2 we apply them in a sequential way, which means we need an ordering on these assignments. Which transitions for this event can actually fire, and with that, which assignments need to be considered, can only be determined during runtime. Instead of creating updates in mCRL2 for

every possible combination of transitions for an event, we use one update that consists of a conditional assignment for each assignment, whose application depends on the transition precondition $\mathcal{PRC}(t)$ of the transition t, for which the assignment is part of the transition update $\mathcal{U}(t)$.

Definition 28 Let $e \in E$ be an event. Then we define $\hat{U}(e)$ as the list containing all pairs from the set $\{(\mathcal{PRC}(t), u) \mid u \in \mathcal{U}(t), t \in T_e\}$ in some order.

We update an instance struct $\mathtt{s}$ in a sequential way by nesting setter applications on the first parameter of the setters. To properly model a simultaneous update, we need to use the original instance struct $\mathtt{s}$ to retrieve the values for instance variables in the right-hand sides of assignments. In mCRL2, we generate this nesting as follows:

Definition 29 Let $\mathtt{s}$ be some instance struct, l be a list of pairs (b, u), where b is a boolean expression and u is an assignment to an instance variable. Then, the *updated instance struct* $\mathcal{US}(l, \mathtt{s})$ that results from applying the conditional assignments in l on $\mathtt{s}$ is an mCRL2 expression constructed as follows:

$$\mathcal{US}(l, \mathtt{s}) = \begin{cases} \mathtt{s} & \text{if } l = \epsilon \\ \mathtt{SET}_x(\mathcal{US}(l', \mathtt{s}), \sigma_{\mathtt{s}}(b), \sigma_{\mathtt{s}}(f)) & \text{if } l = (b, x := f) + l' \end{cases}$$

Using the above definitions, $\mathcal{US}(\hat{U}(e), \mathtt{s})$ defines the updated instance struct after the occurrence of event e in $\mathtt{s}$. The boolean parameter of the setters are used to make sure that exactly the assignments of transitions that fire are applied.

Example 18 As illustrated in Example 7 in Sect. 2, the transitions of event `turn_on` define the assignments `power` $:=$ $'on'$ and `tmp` $:=$ `tmp` $+ 5$. If `turn_on` would occur in an instance state `s`, the resulting updated instance state, which corresponds to $\mathcal{US}(\hat{U}(\mathtt{turn\_on}), \mathtt{s})$, is described in mCRL2 as follows:

```
SET_tmp(SET_power(s,
  GET_power(s) == power_off, power_on),
        GET_tmp(s) < 45, GET_tmp(s) + 5)
```

This update results in the instance struct equal to `IS(power_on, GET_job(s), GET_tmp(s) + 5, GET_sheets(s))`. In case `s` would be the initial instance struct `IS(power_off, job_idle, 20, 0)`, the updated instance struct can be rewritten to `IS(power_on, job_idle, 25, 0)`.

With the following two lemmas we can compare the definition and application of updates in mCRL2 with the formal OIL semantics defined in Sect. 4. Lemma 3 shows that for every event e, the set of assignments $\mathcal{U}(T_e^v)$ of transitions of e that can fire corresponds to the list of assignments $\hat{U}(e)$. Lemma 4 shows that the value for every variable $x \in X_I$ after the applying the update $\mathcal{U}(T_e^v)$ is the same as after applying the corresponding update expression $\mathcal{US}(\hat{U}(e), \mathtt{s})$.

Lemma 3 *Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$ and $v = s \cup p$. Then $u \in \mathcal{U}(T_e^v) \Leftrightarrow \exists_{b \in EXP_X} : (b, u) \in \hat{U}(e) \wedge [\![b]\!]v$.*

Lemma 4 *Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $v = s \cup p$ and $v_S = s_S \cup p$. If $CP(v, \mathcal{U}(T_e^v))$, then for all $x \in X_I$, $v[\mathcal{U}(T_e^v)](x) = [\![GET_x(\mathcal{US}(\hat{U}(e), s))]\!]v_S$.*

Note that in case more than one assignment to the same instance variable is considered, only the last of these assignments in the order has effect, as it overwrites the previous one. In case the assignments are compatible it does not matter for the end result in which order the assignments are applied, since assignments to the same variable assign the same value. In case the assignments are incompatible, different orders may lead to different resulting instance states. This is not an issue however, since incompatibility should result in failure of the event. To check whether the assignments are compatible, we add compatibility checks after the update has been done. These compatibility checks check for every assignment $x := f$ whether the value of x in the updated instance struct equals f. Since these checks need the updated instance struct, they can be done together with the postconditions.

Definition 30 Let $t \in T$ be some transition and let $\mathtt{s}$ and $\mathtt{us}$ be two instance structs that represent the state before, respectively, after an update. Then, the transition's compatibility checks $\mathcal{CP}(t, \mathtt{s}, \mathtt{us})$, the transition's altered postcondition $\mathcal{POC}(t, \mathtt{s}, \mathtt{us})$ and their combination $\mathcal{PCP}(t, \mathtt{s}, \mathtt{us})$ under the assumption that the transition's precondition holds are mCRL2 expressions constructed as follows:

$$\mathcal{CP}(t, \mathtt{s}, \mathtt{us}) = \bigwedge_{x := f \in \mathcal{U}(t)} \mathtt{GET}_x(\mathtt{us}) = \sigma_{\mathtt{s}}(f)$$

$$\mathcal{POC}(t, \mathtt{s}, \mathtt{us}) = \sigma^{\mathtt{s}}_{\mathtt{us}}(\mathcal{POC}(t))$$

$$\mathcal{PCP}(t, \mathtt{s}, \mathtt{us}) = \sigma_{\mathtt{s}}(\mathcal{PRC}(t)) \Rightarrow (\mathcal{CP}(t, \mathtt{s}, \mathtt{us}) \wedge \mathcal{POC}(t, \mathtt{s}, \mathtt{us}))$$

If two assignments $x := f$ and $x := g$ in an update are incompatible, it depends on the order of the assignments which compatibility check is violated. If the assignment $x := g$ is applied later, it overwrites the application of $x := f$, which violates the compatibility check of $x := f$.

Example 19 For the assignments that correspond to event `turn_on`, we add the compatibility checks `GET_power(us) == power_off` and `GET_tmp(us) == GET_tmp(s)`

+ 5, where s and us are the instance structs before, respectively, after updating.

The main reason we check compatibility after the update is due to the complexity of checking it before the update. Since in general we need to accommodate any instance state, we do not know beforehand which combinations of transitions can fire. If in the worst case each transition of an event has an assignment to the same variable, we would need to check compatibility for this variable for every pair of transitions, which results in a number of checks quadratic to the amount of transitions of an event. If we check after the update as part of the postconditions we do not need to compare between transitions. Note that in either case the transition preconditions are needed to only check the compatibility checks and postconditions of transitions that can fire or have fired.

With the following two lemmas, and the corollary that follows from them, we can compare checking compatibility and checking postconditions as done in the operational semantics (Definition 23) to checking them in mCRL2. Lemma 5 shows that checking compatibility with $CP(v, \mathcal{U}(T_e^v))$ corresponds to checking compatibility with $\mathcal{CP}(t, \mathtt{s}, \mathcal{US}(\hat{U}(e), \mathtt{s}))$. Lemma 5 shows that checking postconditions with $\mathcal{POC}(T_e^v)$ corresponds to checking postconditions with $\mathcal{POC}(t, \mathtt{s}, \mathcal{US}(\hat{U}(e), \mathtt{s}))$.

Corollary 1 combines the two lemmas.

Lemma 5 *Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $v = s \cup p$ and $v_S = s_S \cup p$. Then $CP(v, \mathcal{U}(T_e^v)) \Leftrightarrow [\![\bigwedge_{t \in T_e} \sigma_S(\mathcal{PRC}(t)) \Rightarrow \mathcal{CP}(t, s, \mathcal{US}(\hat{U}(e), s))]\!] v_S$.*

Lemma 6 *Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $v = s \cup p$ and $v_S = s_S \cup p$. If $CP(v, \mathcal{U}(T_e^v))$, then $[\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]} \Leftrightarrow [\![\bigwedge_{t \in T_e} \sigma_S(\mathcal{PRC}(t)) \Rightarrow \mathcal{POC}(t, s, \mathcal{US}(\hat{U}(e), s))]\!] v_S$.*

Corollary 1 *Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $v = s \cup p$ and $v_S = s_S \cup p$. Then $CP(v, \mathcal{U}(T_e^v)) \wedge [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]} \Leftrightarrow [\![\bigwedge_{t \in T_e} \mathcal{PCP}(t, s, \mathcal{US}(\hat{U}(e), s))]\!] v_S$.*

In the process specification, the behaviour of an OIL model is encoded using a single monolithic process P with an instance struct parameter to record the instance state and a boolean parameter which is false iff an event has failed. The body of process P is a non-deterministic choice between a number of summands, one for each event in the OIL specification. Additionally, to model the failure state, we have a summand with a self-loop labelled with action fail. We define $\bar{p^e}$ as the vector of variables in $\mathcal{PAR}(e)$ and τ^e as the data type of this vector for some event e.

Definition 31 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification and let is be a ground instance struct that represents the initial instance state as defined by $\mathcal{I}$. Then the *acceptor semantics described in mCRL2* of this OIL specification is defined as the process expression P(is, true) where P is a process defined with the LPE:

$$\begin{aligned}&\mathtt{P}(\mathtt{s} : \mathtt{ISt}, \mathtt{b} : \mathtt{Bool}) = \\ &\quad \sum_{e \in E} \sum_{\bar{p^e} : \tau^e} (\mathtt{b} \wedge \sigma_{\mathtt{s}}(\mathcal{CC}(e))) \rightarrow e(\bar{p^e}) \cdot \\ &\qquad\qquad \mathtt{P}(\mathtt{us}, \bigwedge_{t \in T_e} \mathcal{PCP}(t, \mathtt{s}, \mathtt{us})) + \\ &\quad \neg \mathtt{b} \rightarrow \mathtt{fail} \cdot \mathtt{P}(\mathtt{s}, \mathtt{b})\end{aligned}$$

where $\mathtt{us} = \mathcal{US}(\hat{U}(e), \mathtt{s})$.

For the purpose of testing the translation to mCRL2, a version of the translation was created that defined auxiliary variables in each summand, one for every transition precondition and one for the updated state. This was done to make the generated mCRL2 specification more readable. Somewhat to our surprise, experiments showed that this version required considerably more time for model checking because more rewriting effort was needed. The tool lpssumelm from the mCRL2 toolset can eliminate these auxiliary variables.

An adjustment that did improve the efficiency for model checking was not adding unnecessary compatibility checks. From experience, it is often the case that incompatibility is not possible in the context of an event for an instance variable, because there is only one assignment that assigns to it. Adding the compatibility check for this assignment only adds more unnecessary rewriting effort for the mCRL2 toolset. Therefore, we first analyse the OIL specification to check for possible incompatibilities, that is whether there is more than one assignment to the same instance variable in transitions of an event, and then we only add the incompatibility checks for such assignments in the translation.

Example 20 See Fig. 5 for part of the main process P of the running example, showing only the summand for the event turn_on with auxiliary variables. On line 4 we define auxiliary variables f1, f2 and us, which represent the transition preconditions of the transitions turn_on #1 and turn_on #2 and the updated instance state respectively. This is done using the sum-operator to declare the variables, followed by conditions to fix their values (lines 4-5). The variables f1 and f2 are supplied to the setters so that only the updates of transitions that can fire are applied. The boolean b and the concern condition are checked on line 6. On line 7 the action turn_on is done and then the process recurses with the updated instance state and the postconditions. One could expect a compatibility check GET_tmp(us) ==

```
proc
  P(s : ISt, b : Bool) =
    ...
    sum f1, f2 : Bool, us : ISt.(f1 == (GET_power(s) == power_off) && f2 == (GET_tmp(s) < 45) &&
    us == SET_tmp(SET_power(s, f1, power_on), f2, GET_tmp(s) + 5) &&
    b && (f1 && f2)) ->
    turn_on.P(us, (f1 => GET_power(us) == power_on) && (f2 => GET_tmp(us) < 45)) +
    ...;
```

Fig. 5 Part of process `P` of the mCRL2 specification generated from the OIL specification visualised in Fig. 1, showing only the summand for the event `turn_on` with auxiliary variables

`GET_tmp(s) + 5` here due to the update `SET_tmp(.., f2, GET_tmp(s) + 5)`, but since there is only one assignment to `tmp` defined for event `turn_on`, this check is not necessary so it is left out.

Given an OIL specification, its acceptor semantics described as an IOLTS and its acceptor semantics described in mCRL2 have the same behaviour.

Theorem 1 *Let* $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ *be an OIL specification. Let* $\langle S, s_0, I, O, H, \rightarrow \rangle$ *be the IOLTS that describes the acceptor semantics of this OIL specification* (Definition 23)*. Let* $\langle S', s'_0, L', \rightarrow' \rangle$ *be the LTS that corresponds to the LPE of* `P` (Definition 26) *where* `P(is, true)` *describes the acceptor semantics of this OIL specification in mCRL2* (Definition 31)*. Then* $s_0 \leftrightarrows s'_0$.

6.2.1 Execution semantics

In Definition 24 the execution semantics is acquired from the acceptor semantics by prioritisation of proactive events. Such prioritisation is however at the time of writing not available in the mCRL2 tool set. Therefore we choose to create a direct translation from an OIL component specification to its execution semantics in mCRL2. For this we define the proactive priority condition, which, given an event and an instance state, is true iff the event is proactive or there are no proactive events possible in the given instance state. Whether proactive events are possible is checked by checking the concern conditions of each proactive event.

Definition 32 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification, $e \in E$ an event and `s` some instance struct. Then the mCRL2 expression encoding the *proactive priority condition* $\mathcal{PPC}(e, \mathtt{s})$ is constructed as follows:

$$\mathcal{PPC}(e, \mathtt{s}) = \begin{cases} \neg \bigvee_{e' \in E_P} \exists_{\bar{p^{e'}}:\tau^{e'}} : \sigma_{\mathtt{s}}(\mathcal{CC}(e')) & \text{if } e \in E_R \\ \mathtt{true} & \text{if } e \in E_P \end{cases}$$

Example 21 The two events `\sheet_printed` and `\job_printed` are the only proactive events in the running example. Therefore, the proactive priority condition for the running example in some instance struct `s` is defined in mCRL2 as:

```
!((exists snr : Int.GET_job(s)
     == job_busy &&
   GET_sheets(s) > 0 && snr
    == GET_sheets(s))
  || (GET_job(s) == job_busy &&
      GET_sheets(s) == 0))
```

Along with the concern condition, the proactive priority condition must also hold for an event to be allowed, as described below.

Definition 33 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification and let `is` be a ground instance struct that represents the initial instance state as defined by $\mathcal{I}$. Then the *execution semantics described in mCRL2* of this OIL specification is defined as the process expression $P(\mathtt{is}, \mathtt{true})$ where P is a process defined with the LPE:

$$\begin{aligned} &\mathtt{P}(\mathtt{s} : \mathtt{ISt}, \mathtt{b} : \mathtt{Bool}) = \\ &\quad \sum_{e \in E} \sum_{\bar{p^e}:\tau^e} (\mathtt{b} \wedge \sigma_{\mathtt{s}}(\mathcal{CC}(e)) \wedge \mathcal{PPC}(e, \mathtt{s})) \rightarrow e(\bar{p^e}) \cdot \\ &\qquad\qquad \mathtt{P}(\mathtt{us}, \bigwedge_{t \in T_e} \mathcal{PCP}(t, \mathtt{s}, \mathtt{us})) + \\ &\quad \neg \mathtt{b} \rightarrow \mathtt{fail} \cdot \mathtt{P}(\mathtt{s}, \mathtt{b}) \end{aligned}$$

where $\mathtt{us} = \mathcal{US}(\hat{U}(e), \mathtt{s})$.

Given an OIL specification, its execution semantics described as an IOLTS and its execution semantics described in mCRL2 have the same behaviour.

Theorem 2 *Let* $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ *be an OIL specification. Let* $\langle S, s_0, I, O, H, \rightarrow \rangle$ *be the IOLTS that describes the execution semantics of this OIL specification (Definition 24). Let* $\langle S', s'_0, L', \rightarrow' \rangle$ *be the LTS that corresponds to the LPE of* `P` *(Definition 26) where* `P(is, true)` *describes the execution semantics of this OIL specification in mCRL2 (Definition 33). Then* $s_0 \leftrightarrows s'_0$.

Example 22 See Appendix A.3 for the full mCRL2 specification that describes the execution semantics of the running example.

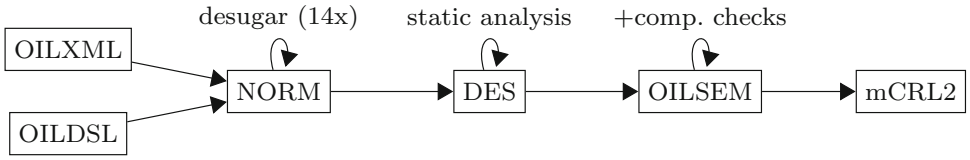

Fig. 6 The transformation pipeline implemented in Spoofax from OIL specification to mCRL2 specification. NORM refers to the normalised AST and DES refers to the desugared AST

6.3 Implementation of the translation to mCRL2

The translation from OIL to mCRL2 has been implemented in the Spoofax language workbench [44] using the model transformation language Stratego [11]. It makes use of the already available Spoofax implementations of OIL by Denkers [18] and mCRL2 by Van Antwerpen[1]. A total of 20 separate consecutive transformations are used to translate an OIL specification to an mCRL2 specification. See Fig. 6 for a visualisation of this pipeline. An OIL specification is first transformed to the normalised AST, which serves as a middle ground between OILXML and OILDSL. On this normalised AST a number of desugaring and explication transformations have been defined, which are required for the transformation to the desugared AST. This desugared AST is semantically equivalent to the normalised AST, reduced to basic constructs. To annotate variables with types, static analysis is applied on the desugared AST. Inspired by the work of Frenken [20] on a C++ code generator for OIL in Spoofax, an additional intermediate representation is generated before generating mCRL2, called OILSEM. This intermediate representation is designed to correspond closely to the formal semantics of OIL. On this representation we add compatibility checks to the postconditions of transitions. Lastly, we transform the OILSEM representation to mCRL2.

The transformations consist of about 1200 lines of code and 400 transformation rules in total. Most desugar transformations are fairly small with at most 40 lines of code and 10 transformation rules. The transformation from OILSEM to mCRL2 is the most complex one with 300 lines of code and 130 transformation rules.

Although the formal definition of the semantics of OIL described in mCRL2 is proven to correctly correspond to the formal semantics of OIL, the same is not guaranteed for the translation to mCRL2 implemented in Spoofax. Nevertheless, we are confident that it is correct. Firstly, the OILSEM representation was designed to contain the same data used in the definition of the operational semantics (Definition 23). For instance, each transition defines its precondition, update and postcondition (Definition 19-21). Because Stratego is a functional language, the definitions in Stratego correspond closely to the formal definitions. Additionally, all transformations up to OILSEM are quite small and straightforward and most desugaring transformations are equipped with postconditions that check whether the desugaring was applied correctly.

During the development of the more complex transformation from OILSEM to mCRL2 we have relied on the mCRL2 toolset to check for regressions and correctness of the translation. Whenever a new concept of OIL was added to the translation, an OIL specification illustrating this concept was translated to mCRL2. Then the corresponding LTS was generated using the mCRL2 toolset to check whether the implementation of the new concept resulted in expected behaviour. Also, we used equivalence checking to test whether a refactoring in the translation to mCRL2, such as the one that adds auxiliary variables to summands, did not change the behaviour of generated mCRL2 specifications. This was done by comparing the LTS before with the LTS after the refactoring, for a test set of OIL specifications. In a few occasions this has revealed subtle errors in refactorings that might have been overlooked otherwise. Equivalence checking was also applied to test whether mCRL2 specifications generated from the current translation and from one written in Python, developed in an exploratory phase of this project, have the same behaviour. This showed that there was a subtle mistake in the original Python translation that resulted in faulty behaviour in some generated mCRL2 specifications. In general, the use of formal methods during the development process has given us more confidence regarding the correctness of the translation implemented in Spoofax.

7 Validation of OIL specifications

To verify whether an OIL specification is valid, that is whether all four requirements defined in Sect. 5 are met, we can express these requirements in terms of mu-calculus formulae and check them on the corresponding mCRL2 specification described in Definition 31 using the mCRL2 tool set. The proofs of lemmas presented in this section can be found in Appendix B.2.

[1] See https://github.com/MetaBorgCube/metaborg-mcrl2

7.1 Mu-calculus

The mu-calculus is an algebra used to define properties over an LTS. In this document we only consider a subset of the mu-calculus as defined in [22].

Definition 34 Let $\langle S, s_0, L, \rightarrow\rangle$ be an LTS. Then a mu-calculus formula ϕ has the following grammar:

$$\begin{aligned}\phi :=\ & b \mid Z(\bar{e}) \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \langle a\rangle\phi \mid [a]\phi \\ & \mid \exists_{\bar{d}:D}.\phi \mid \forall_{\bar{d}:D}.\phi \\ & \mid \mu Z(\bar{d} : D := \bar{e}).\phi \mid \nu Z(\bar{d} : D := \bar{e}).\phi\end{aligned}$$

where b is a boolean expression, Z is a fixpoint variable, $\bar{e}$ is a vector of expressions, $a \in L$ is an action, $\bar{d}$ is a vector of data variables and D is data type. In case a fixpoint variable or an action does not have any parameters, the parentheses are omitted. To ensure monotonicity of fixpoint operators, we do not allow fixpoint variables in formulae on the left-hand side of an implication operator $\Rightarrow$.

To ease notation, we extend the modal operators over a set of actions $L' \subseteq L$ or a set of sequences of these actions L'^*. We define the following short-hand notations:

$$\langle L'\rangle\phi = \bigvee_{a\in L'} \langle a\rangle\phi \qquad [L']\phi = \bigwedge_{a\in L'} [a]\phi$$

$$\langle L'^*\rangle\phi = \mu Z.(\langle L'\rangle Z \vee \phi) \qquad [L'^*]\phi = \nu Z.([L']Z \wedge \phi)$$

Given an LTS and a mu-calculus formula, one can extract the set of states in the LTS on which this formula is true, which is defined as follows:

Definition 35 Let Φ be the set of all mu-calculus formulae, η a valuation over fixpoint variables, v a valuation over data variables and $\langle S, s_0, L, \rightarrow\rangle$ an LTS. Then, the semantics $[\![\phi]\!]\eta v \subseteq S$ of a mu-calculus formula ϕ is defined as:

$$\begin{aligned}[\![b]\!]\eta v &= \begin{cases} S \text{if } [\![b]\!]v = true \\ \emptyset \text{if } [\![b]\!]v = false \end{cases} \\ [\![Z(e)]\!]\eta v &= \eta(Z)([\![e]\!]v) \\ [\![\phi_1 \vee \phi_2]\!]\eta v &= [\![\phi_1]\!]\eta v \cup [\![\phi_2]\!]\eta v \\ [\![\phi_1 \wedge \phi_2]\!]\eta v &= [\![\phi_1]\!]\eta v \cap [\![\phi_2]\!]\eta v \\ [\![\phi_1 \Rightarrow \phi_2]\!]\eta v &= (S \setminus [\![\phi_1]\!]\eta v) \cup [\![\phi_2]\!]\eta v \\ [\![\langle a\rangle\phi]\!]\eta v &= \{s \in S \mid \exists_{s'\in S} : s \xrightarrow{a} s' \wedge s' \in [\![\phi]\!]\eta v\} \\ [\![[a]\phi]\!]\eta v &= \{s \in S \mid \forall_{s'\in S} : s \xrightarrow{a} s' \Rightarrow s' \in [\![\phi]\!]\eta v\} \\ [\![\exists_{\bar{d}:D}.\phi]\!]\eta v &= \bigcup_{c\in\mathbb{D}} [\![\phi]\!]\eta v[\bar{d} := c] \\ [\![\forall_{\bar{d}:D}.\phi]\!]\eta v &= \bigcap_{c\in\mathbb{D}} [\![\phi]\!]\eta v[\bar{d} := c] \\ [\![\mu Z(\bar{d} : D := \bar{e}).\phi]\!]\eta v &= \mu(f^{\eta,v}_{Z,\bar{d}})([\![\bar{e}]\!]v) \\ [\![\nu Z(\bar{d} : D := \bar{e}).\phi]\!]\eta v &= \nu(f^{\eta,v}_{Z,\bar{d}})([\![\bar{e}]\!]v)\end{aligned}$$

$$f^{\eta,v}_{Z,\bar{d}} = \lambda Y.\lambda c.[\![\phi]\!]\eta[Z := Y]v[\bar{d} := c]$$

where $\mathbb{D}$ is the set of all values that correspond to data type D, $\mu(f) = \bigsqcap\{x \mid x = f(x)\}$ and $\nu(f) = \bigsqcup\{x \mid x = f(x)\}$. The operators $\bigsqcap$ and $\bigsqcup$ are the infimum, respectively, supremum operators corresponding to the subset order lifted to functions in a pointwise fashion.

A state s *satisfies* a mu-calculus formula ϕ, denoted as $s \models \phi$, iff $s \in [\![\phi]\!]\eta v$ for all valuations η and v. We say a mu-calculus formula is *closed* iff every variable reference is within the scope of its declaration. We only consider closed mu-calculus formulae in this paper. Note that for a closed mu-calculus formula it holds that $[\![\phi]\!]\eta v = [\![\phi]\!]\eta' v'$ for all valuations η, η', v and v'.

When checking a mu-calculus formula on an LTS, one is usually only interested in whether the initial state satisfies the formula. As mentioned in [1], to check whether a mu-calculus formula ϕ is satisfied by all states reachable from some state s, one can check the formula $[L^*]\phi$ on s, where L is the set of all actions in the LTS, see Lemma 7 below.

Lemma 7 *Let $\langle S, s_0, L, \rightarrow\rangle$ be an LTS, $s \in S$ some state, $L' \subseteq L$ some set of action labels and ϕ be some closed mu-calculus formula. Then, $s \models [L'^*]\phi \Leftrightarrow \forall_{t\in S_R^{s,L'}} : t \models \phi$.*

Whenever we say that a mu-calculus formula ϕ is true on a transition system, we mean that $s_0 \models \phi$ where s_0 is the initial state of the transition system.

7.2 Checking the validity requirements

In this section, we show how each validity requirement can be checked on an OIL specification by formalising them in the mu-calculus. For Requirement 3 and 4 we also define algorithms to check them directly on an IOLTS, since the mu-calculus isn't well suited for these requirements. Let $\langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS$^\sqcap$ that describes the execution semantics of an OIL specification. We assume that S and $\rightarrow$ are finite.

Safe lookaheadlessness

Requirement 1 disallows the existence of any trace that has a proactive event followed by the failure action `fail`. This requirement can be formalised in the mu-calculus with the formula $\phi_{R1} = [L^*][O \cup H][\texttt{fail}]false$.

Lemma 8 *Let $M = \langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS$^\sqcap$ that describes the execution semantics of an OIL specification. Then R1 is met on M iff $s_0 \models \phi_{R1}$.*

Finite proactivity

Requirement 2 requires that no sequence of proactive events is infinite. Using the construct $\mu Z.[L']Z$ which is true iff all L'^* sequences are finite for some set of actions L' (as shown in [14]), this requirement can be formalised in the mu-calculus with the formula $\phi_{R2} = [L^*]\mu Z.[O \cup H]Z$.

Lemma 9 *Let $M = \langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS$^\sqcap$ that describes the execution semantics of an OIL specification. Then R2 is met on M iff $s_0 \models \phi_{R2}$.*

Confluent proactivity

For Requirement 3 we need to know whether sequences end up in the behaviourally same (quiescent) state, which is something that cannot be expressed with the mu-calculus directly. We can work around this by first reducing the resulting transition system modulo strong bisimulation and then marking every quiescent state with a unique action a from some action set Q by means of a self-loop. Then Requirement 3 can be formalised in the mu-calculus with the formula:

$$\phi_{R3} = [L^*] \bigvee_{a \in Q} [(O \cup H)^*]([O \cup H]false \Rightarrow \langle a\rangle true)$$

This formula checks for every reachable state ($[L^*]$) if there exists a quiescent state, identified by an action a in Q ($\bigvee_{a\in Q}$), such that after every sequence of proactive events ($[(O \cup H)^*]$) that ends up in a quiescent state ($[O \cup H]false$), this quiescent state is the one marked with a ($\langle a\rangle true$).

Lemma 10 *Let $M = \langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS$^\sqcap$ that describes the execution semantics of an OIL specification. Then R is met on M if $s_0 \models \phi_{R3}$.*

Predictable proactivity

For Requirement 4 we need to know what sequences of proactive events are possible, which can be collected by adding data to fixpoint variables. This requirement can be formalised in the mu-calculus with the formula:

$$\phi_{R4} = [L^*]\exists_{w:Bag(O\cup H)} : \nu X(w' : Bag(O \cup H) := \emptyset).$$
$$\bigwedge_{a\in O\cup H} [a]X(w' + \{a\}) \wedge ([O \cup H]false \Rightarrow w = w')$$

The structure of this mu-calculus formula is similar to that of the mu-calculus formula of confluent proactivity. Instead of checking for the existence of a particular quiescent state (action in Q), in this formula we check for the existence of a particular multiset w, also known as a "bag" in mCRL2 ($\exists_{w\in Bag(O\cup H)}$). To collect all possible sequences of proactive events, we start with the empty multiset ($\nu X(w' : Bag(O \cup H) := \emptyset)$) and then add (unique representations of) events one by one while following the sequences ($[a]X(w' + \{a\})$). When we reach a quiescent state ($[O \cup H]false$), we require that the constructed multiset w' equals w ($w = w'$).

Lemma 11 *Let $M = \langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS$^\sqcap$ that describes the execution semantics of an OIL specification. Then R4 is met on M iff $s_0 \models \phi_{R4}$.*

Note that checking this mu-calculus formula does not terminate due to the existential quantification over an infinite domain. This can be solved by adding information (in the form of a self loop) to each non-quiescent state in the IOLTS that provides a multiset of actions corresponding to a possible proactive sequence. This information can then be used in the mu-calculus formula by adding a diamond operator to give the existential quantifier a value to pick. It is also possible to check this requirement without the need to adapt the IOLTS, namely by checking multiset equality for every pair of proactive sequences. First, using a fixpoint operator and a universal quantifier over proactive actions, we can recursively compute all possible proactive sequences. Then, for each proactive sequence w found, we can go through all possible proactive sequences again and compare them to w, similarly to ϕ_{R4}. However, this is very inefficient since a quadratic number of comparisons are done. This can be improved by defining an ordering on actions, which induces a topological ordering on sequences. Then, we can compute the "largest" (or "smallest") possible proactive sequence initially to compare to all other sequences. Note that finite proactivity is required to make sure that computing possible proactive sequences terminates.

Alternative methods of checking confluent and predictable proactivity

To be able to effectively check Requirement 3 and 4 using the mu-calculus, the IOLTS needs to be adapted or a complex mu-calculus formula is needed, which indicates that the mu-calculus is not a great fit for these requirements. Therefore we propose an alternate way of checking these requirements, namely by means of an algorithm that works directly on the IOLTS. See Algorithm 1 for the algorithm to check Requirement 3 and Algorithm 2 for the algorithm to check Requirement 4.

Both algorithms are very similar in structure. They both have an initialisation phase followed by a recursive depth first search function that returns false if a violation of the requirement has been found. For confluent proactivity we first reduce the IOLTS modulo strong bisimulation (line 2) to make sure that two states are bisimilar iff they are equal. Then we declare a map to store values for states. In Algorithm 1 we declare a map R to store reachable quiescent states (line

Algorithm 1 Checking confluent proactivity

```
1: function IsConf(M = ⟨S, s0, I, O, H, →⟩)
2:     Reduce M modulo strong bisimilation
3:     Declare R as an empty map from S to Sδ
4:     R[s] = s for every s ∈ Sδ
5:     P = Sδ
6:     for s ∈ SR do
7:         if ¬IsConfDFS(s) then
8:             return false
9:     return true
10:
11: function IsConfDFS(s)
12:     if s ∉ P then
13:         for (s, a, s′) ∈ → do
14:             if ¬IsConfDFS(s′) then
15:                 return false
16:             if s ∉ R then
17:                 R[s] := R[s′]
18:             else if R[s] ≠ R[s′] then
19:                 return false
20:         P := P ∪ {s}
21:     return true
```

Algorithm 2 Checking predictable proactivity

```
1: function IsPred(M = ⟨S, s0, I, O, H, →⟩)
2:     Declare B as an empty map from S
3:                 to multisets over O ∪ H
4:     B[s] = ∅ for every s ∈ Sδ
5:     P = Sδ
6:     for s ∈ SR do
7:         if ¬IsPredDFS(s) then
8:             return false
9:     return true
10:
11: function IsPredDFS(s)
12:     if s ∉ P then
13:         for (s, a, s′) ∈ → do
14:             if ¬IsPredDFS(s′) then
15:                 return false
16:             if s ∉ B then
17:                 B[s] := B[s′] + a
18:             else if B[s] ≠ B[s′] + a then
19:                 return false
20:         P := P ∪ {s}
21:     return true
```

3) and in Algorithm 2 we declare a map B to store multisets that represents possible proactive sequences (lines 2-3). Both maps are initialised for quiescent states (line 4). We also initialise a set P, which stores all states that have already been processed. Then for every reachable state (line 6) we call the recursive function (line 7). If some recursive call has found a violation of the requirement we return false (line 8), otherwise we return true (line 9).

In the recursive functions (line 11), we first check if state s was not already processed (line 12). Then for every transition from s to some state s' with some action a (line 13), we call the recursive function on s' (line 14). If the recursive call returns false because a violation was found, we propagate this back up immediately (line 15). Otherwise, if we do not have a value for s (line 16), namely in the first iteration, we create and store a value based on the value for s', that was just computed by the recursive call (line 17). In Algorithm 1 the new value for s is the quiescent state that s' can reach and in Algorithm 2 the new value for s is the multiset computed for s' with an increment for action a that was needed to reach s' from s. If there was a value stored for s already, we compare this to the new value (line 18) and return false in case they are not equal (line 19), since this is a violation of the requirement. After all outgoing transition have been considered and no violation was found, we add s to the set P of processed states (line 20) and return $true$ to indicate that no violation was found (line 21).

Correctness of Algorithm 1 can be shown by adding post-conditions to IsConfDFS(s) depending on what it returns. In case IsConfDFS(s) returns true, we have the following postcondition:

$$s \in P \wedge \forall_{t \in P, w \in (O \cup H)^*, u \in S_\delta} : t \xrightarrow{w}{}^* u \Rightarrow \mathrm{R}[t] = u$$

At the end of the algorithm, if no violations have been found, we know that IsConfDFS(s) was called and returned true for every $s \in S_R$. From the postcondition it then follows that $P \supseteq S_R$ and that Requirement 3 is fulfilled. In case a violation is found and IsConfDFS(s) returns false, we have the following postcondition:

$$\exists_{w, w' \in (O \cup H)^*, u, u' \in S_\delta} : s \xrightarrow{w}{}^* u \wedge s \xrightarrow{w'}{}^* u' \wedge u \neq u'$$

which says that state s violates the requirement. The correctness for Algorithm 2 can be shown similarly with the postcondition

$$s \in P \wedge \forall_{t \in P, w \in (O \cup H)^*, u \in S_\delta} : t \xrightarrow{w}{}^* u \Rightarrow w \approx\!\!\!| \; B[t]$$

if IsPredDFS(s) returns true, where $w \approx\!\!\!| \; B[t]$ iff w as a multiset of actions equals $B[t]$, and the postcondition

$$\exists_{w, w' \in (O \cup H)^*, u, u' \in S_\delta} : s \xrightarrow{w}{}^* u \wedge s \xrightarrow{w'}{}^* u' \wedge w \not\approx w'$$

if IsPredDFS(s) returns false, where $w \not\approx w'$ iff w and w' do not have the same multiset of actions.

Finite proactivity (Requirement 2) is required for these algorithms to terminate. We show termination for Algorithm 1; the same arguments can be made for Algorithm 2. The function IsConfDFS(s) terminates if $s \in S_\delta$ since $S_\delta \subseteq P$ due to line 5. In case $s \notin S_\delta$, IsConfDFS(s) may call IsConfDFS(s') for successor states s' on line 14. Given that finite proactivity holds, we know that there are no infinite sequences of proactive actions and therefore the recursion always eventually ends in a quiescent state $t \in S_\delta$, for which

IsConfDFS(t) is known to terminate. From this it follows that line 14 always terminates. Since we assume that S and $\rightarrow$ are finite, we know that the for loops on lines 6 and 13 always terminate, from which we can conclude that Algorithm 1 always terminates.

Apart from the reduction modulo strong bisimulation, which can be done in $O(|\rightarrow| \log |S|)$ [43], both algorithms run in $O(|S| + |\rightarrow|)$. We will show this for Algorithm 1; the arguments are the same for Algorithm 2. The initialisation on line 4 runs in $O(|S|)$ in the worst case. The function IsConfDFS(s) runs in constant time if $s \in P$, which is always the case for $s \in S_\delta$ due the initialisation on line 4. If $s \notin P$, we call IsConfDFS(s') for every outgoing transition $s \xrightarrow{a} s'$. After such a call returns, in the worst case, a value is assigned to $R[s]$ right after. Since finite proactivity holds, we know that a call of IsConfDFS(s) cannot eventually lead to another call of IsConfDFS(s) and must eventually lead to a call IsConfDFS(t) for some $t \in S_\delta$. Therefore, for each $s \in S$, IsConfDFS(s) runs linear to the number of its outgoing transitions at most once. Since each transition can only have one source state, it follows that each transition in $\rightarrow$ is only considered once, in all calls of IsConfDFS combined. This implies that the loop on lines 5-7 runs in $O(|\rightarrow|)$, so the total algorithm (excluding the reduction modulo strong bisimulation) runs in $O(|S| + |\rightarrow|)$.

8 Experiments

To test the feasibility of our techniques, we have applied them on two OIL models representing systems used in production at Canon Production Printing. We refer to these two models as EPC and AGA. In the rest of this section we will give some results and experiences regarding experiments done on these models.

To obtain the size of the instance state space, we generate the LTS from the generated mCRL2 specification. This LTS is then reduced modulo bisimulation to remove any superfluous behaviour. See Fig. 4 for the tools used to generate an LTS and to check a property expressed in the mu-calculus. Since the generated mCRL2 specification is already an LPS, we can skip the use of `mcrl22lps` (and use `txt2lps` instead).

The experiments are done on a laptop with Windows 10, an Intel Core i7-56500U 2.50 GHz processor and 16 GB of RAM. Although the mCRL2 toolset tends to run slower on Windows machines, it is the main operating system used within Canon Production Printing. This way we can test whether we can achieve acceptable performance within the default engineering environment. With regard to time needed for translation, we split the transformation pipeline in two: the transformation from OIL specification to analysed desugared AST and from analysed desugared AST to mCRL2. This is done because the analysed desugared AST can easily be reused for translations. For all timings mentioned we have taken the average of at least five runs.

Table 1 The time in seconds needed to check each requirement on the reduced LTS for both the EPC and the AGA case. For R1 and R2 we used the mu-calculus formulae ϕ_{R1} respectively ϕ_{R2} and for R3 and R4 we used Algorithm 1 respectively Algorithm 2

	R1	R2	R3	R4
EPC	0.4	0.4	0.4	0.4
AGA	13	12	21	22

The EPC case

The EPC model is an OIL specification with a total of 10 instance variables, 5 regions, 1 scope, 26 states, 29 transitions and 27 events. It starts with an initialisation phase, then enters a loop and from this loop it can return to the initial state via a termination phase. It models a system used in production, but the code generated from the model itself is not used in production. The analysed desugared AST of the EPC OIL specification is generated in about 7 seconds. From this analysed model the mCRL2 specification is generated in about 2.7 seconds. The LTS can be generated from the mCRL2 specification in about 4.5 seconds. This LTS has 6466 states, 94 actions and 11491 transitions. After reduction modulo strong bisimulation, the LTS has 1178 states and 3207 transitions.

All four validity requirements are met on this model. See Table 1 for the time needed to check each validity requirement on the reduced LTS.

The AGA case

The AGA model is an OIL specification with a total of 55 instance variables, 18 regions, 2 scopes, 179 states, 220 transitions and 185 events. It starts with an initialisation phase and then enters a loop. It models a system used in production and, unlike the EPC model, it is used to generate the actual code for this system. The analysed desugared AST of the AGA OIL specification is generated in about 26 seconds. From this analysed model the mCRL2 specification is generated in about 90 seconds. To be able to generate the LTS for this model within a reasonable amount of time, some changes needed to be made to the OIL specification:

– We gave event parameters of reactive events with an infinite domain a fixed value. These parameters represent values received from the environment. In case such a parameter has an infinite domain, there would be an infinite number of transitions possible in the LTS, which causes the generation of the LTS to not terminate. Since

the values of these parameters were only used to be passed on to other components, this change does not affect the control flow behaviour of the model.
- We removed the assignments to instance variables that are at most only used to pass information on to other components. This keeps these variables at their initial values, which avoids creating multiple branches in the LTS for each value. Note that this effectively abstracts away some event parameters in proactive events, used to pass this information back to the environment. This is not an issue, since these branches are behaviourally the same except for the value for the instance variable and such event parameters and since we are (for now) only concerned with the behaviour of a single component.
- We added assignments to reset instance variables to their initial value after their value becomes irrelevant. This makes the branches in the LTS that represent different values for this variable converge earlier.

After these changes, the LTS can be generated in about 14.6 minutes[2]. The resulting LTS has 113844 states and 177156 transitions. After reduction modulo strong bisimulation, the LTS has 23372 states and 40820 transitions. Some of this reduction is due to non-optimal placement of the resets. However, investigation shows that this is not the only reason for the observed reduction. For instance, we found that the value of a certain instance variable has no effect on the behaviour if another instance variable was set to false.

All validity requirements are met on this model. See Table 1 for the times needed to check each requirement on the reduced LTS.

These validity requirements are of course not the only properties we can check on these models. For instance, we can check deadlock freedom with the μ-calculus formula $[L^*]\langle L\rangle true$, which we can verify to be true on the AGA model. A more interesting property is whether it is always possible to go to the start of the loop in the AGA model. This requirement can be encoded with the μ-calculus formula $[L^*]\langle L^*.\texttt{start}\rangle true$, where `start` represents the event at the beginning of the loop. Checking this formula on the AGA model results in false, which is due to events in the loop that are deliberately put in the model to model a failure in the system. Removing these events from L and checking the formula again results in true. These formulae can be checked on the reduced LTS within a few seconds.

[2] As mentioned earlier, the mCRL2 toolset tends to run slower on Windows machines. This is mostly because the compiling rewriter (passing option `-rjittyc` to `lps2lts`, the state space generation tool), which is typically much faster than the default rewriter, is not available on Windows machines. To experiment what improvement the compiling rewriter could bring we used a virtual machine running Ubuntu 20.04 and using half the laptop's memory. On this virtual machine the LTS can be generated in about 185 seconds from the mCRL2 specification using the option `-rjittyc` for `lps2lts`.

9 Discussion of results

Our translation from OIL to mCRL2 and the subsequent verification of two OIL specifications show that it is possible to model check OIL specifications. The current implementation of this translation comprises a large number of smaller transformations to bridge the large semantical gap between OIL and mCRL2. While this is beneficial for the maintainability and reusability of (parts of) the translation, a monolithic translation would be more efficient. However, the experiments show that for increasingly large models the current translation time is rather insignificant compared to the time needed for model checking.

At the same time, it is clear that improvements are necessary before model checking can be made available to the average engineer. These improvements concern both automating some of the preprocessing of OIL models needed to scale the analysis and enhancements to the back-end verification methodology we currently use.

9.1 Process structure

We have described the semantics of OIL in mCRL2 by using a single monolithic process. A drawback of having a monolithic approach over a compositional approach would be the inability to reuse processes whenever only a part of an OIL specification changes. In the monolithic approach, the whole process specification needs be generated anew. Also, the separate composable processes could be reduced before being combined which could speed up the state space generation of the whole model. Another typical benefit of a compositional approach is maintainability. OIL seems to be quite suitable for a compositional approach due to the separation of concerns. However, we think that a compositional approach for describing the semantics of OIL in mCRL2 would be more complex than the current monolithic approach, mainly for two reasons.

Firstly, processes defined in mCRL2 lack a notion of shared variables and can only exchange information via communication of actions. Since from every part in an OIL specification any instance variable can be read or assigned to, the instance state would need to be synchronised between all processes frequently. A possible alternative would be to model the instance state as a separate process, but such solutions typically scale poorly due to the overhead induced by the extra communications needed by the main process with this additional parallel process.

Secondly, it is complex to model the atomicity of simultaneously firing OIL transitions in mCRL2 in a compositional

manner. Communications of actions in mCRL2 seem suitable to describe synchronisation on an event by means of concerns by creating a process for each concern. However, this synchronisation also requires updating the instance state, if these updates are found to be compatible, and checking whether the event fails. To share results and prevent race conditions between processes when checking compatibility, updating the instance state and checking the postconditions, additional communication would be needed.

9.2 Automating Preprocessing

As the AGA case clearly shows, the state space of an OIL specification has the potential to explode if it has many instance variables. To help the state space generator, we manually analysed the usage of these variables and adapted the OIL specification. This is both tedious and error-prone, and therefore a candidate for automation. We note that there is a wealth of literature on such static analysis; see for instance research in the fields of program slicing [40] and live variable analysis [19]. A more interesting challenge, however, is to investigate whether it is possible to implement such static analysis techniques at the meta-level in a language workbench such as Spoofax, so that such techniques become available to all languages defined in such a workbench.

We remark that the mCRL2 toolset already contains some tools that help reduce the state space by removing variables that have no effect on behaviour, such as `lpsparelm` and `lpsstategraph` [34]. However, experiments have shown that these tools are not very effective on mCRL2 specifications generated from OIL specifications. This is due to our monolithic representation of the instance state. To make these tools more effective, the structure of the generated mCRL2 will have to be redesigned or the tools have to be improved.

9.3 Enhanced Back-end

As shown in Sect. 7, the mu-calculus is a good fit for encoding Requirement 1 and 2, but not for Requirement 3 and 4. We do remark that this is the first time that we have come across a functional property that cannot be expressed in the first-order modal mu-calculus without adding non-trivial information to the model. It may be necessary to resort to an even more expressive logic, such as a higher-order fixed point logic [2] or some hybrid logic [27], to encode such properties in a logic without modifying the model. The downside of using such logics is that, as far as we are aware of, no toolset supports such logics. Alternatively, it may be possible to check these requirements more efficiently by encoding them directly in a Parameterised Boolean Equation System [23] (see PBES in Fig. 4), thereby sidestepping the limitations of the mu-calculus.

Another aspect that could be exploited is that specifications such as the AGA model have a number of instance variables set during the initialisation phase. These basically create configurations for the behaviour that is defined in the loop after the initialisation phase. This could be exploited by modelling them as features instead and apply techniques in the context of software product lines [15]. Some research has already been done regarding model checking software product lines in the context of mCRL2 [5].

9.4 Using model checking for OIL in practice

OIL still is in an early stage of development, so the number of cases where it has been applied for systems used in production within Canon Production Printing is limited. Currently, only two of these cases have been used for model checking experiments, namely the EPC and AGA case described in Sect. 8. We do feel that these two cases are sufficiently representative: the EPC case uses separation of concerns to its fullest extent, while the AGA case models one of the behaviourally most complex components available.

We envision that when OIL is used on a larger scale, we need to hide the complexities of the use of mCRL2 from the engineer. Checking the validity requirements would be done by the click of a button or automatically, and preferably return a counter example when violated. An engineer should also be able to specify custom requirements in a language that is simpler than the mu-calculus, and check these on an OIL specification in a similar fashion. Other uses of the generated mCRL2 would be conformance checking or regression checking, but this is future work.

10 Conclusion

We have presented the Open Interaction Language (OIL), a language for modelling the behaviour of software systems. By means of an example OIL component specification we have explained the semantics of OIL informally. We have also defined the operational semantics of OIL component specifications formally. This considers two layers: the first layer that defines the behaviour that a component is capable of and the second layer that defines the behaviour of a run-to-completion scheduler that executes the component. Both are defined in the form of an input-output labelled transition system. On the execution semantics we have introduced four validity requirements, which aim to prevent undesirable behaviour.

We have defined a translation from OIL to mCRL2, based on the formal operational semantics, to enable the use of model checking techniques on OIL specifications. The mCRL2 specifications generated with this translation have been shown to correspond to the operational semantics of

OIL. Thanks to the definition of the operational semantics, the definition of the translation is rather straightforward; the main difficulties are how to apply the updates and how to represent the instance variables. Another benefit of having this separation is that we can easily experiment with alternate definitions for the translation to mCRL2. The translation has been implemented using the model transformation language Stratego in the language workbench Spoofax.

We have defined the four validity requirements in terms of the mu-calculus so that they can be verified using the mCRL2-toolset. For the last two validity requirements we have also proposed an algorithmic approach, which we find more suitable. We have checked these validity requirements on two OIL specifications of systems used in production at Canon Production Printing and with this showed that the application of model checking techniques on OIL specifications is feasible.

Acknowledgements We thank the reviewers for their helpful feedback.

Availability of data, material and code The implementation of the translation from OIL to mCRL2 cannot be made available currently due to confidentiality, but progress is being made to make it publicly available. The code for Algorithm 1 and 2 used for the experiments can be made available upon request. The industrial models used in the experiments cannot be made available due to confidentiality.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

A Running example

A.1 OILDSL

See below for the OILDSL specification of the OIL specification visualised in Fig. 1.

```
component heat2

enum power {off, on}

enum job {idle, busy}

var power: power = 'off'
var job: job = 'idle'
var tmp: integer = 20
var sheets: integer = 0

region power(this.power)
{
  state off('off')
  state on('on')
}

scope power_on[this.power == 'on']
{
  region job(this.job)
  {
    state idle('idle')
    state busy('busy')
  }
}

scope heat[this.tmp < 45]

in off on turn_on() go on
  concern POWER end
in on on turn_off() go off
    concern POWER end

in idle if nrsheets > 0 and nrsheets <= 3
 on add_job() assign this.sheets
  := nrsheets
 go busy concern JOB end
in busy if this.sheets == 0
 do[silent] job_printed() go idle
  concern JOB end
in busy if this.sheets > 0
 do sheet_printed(sheetnr = this.sheets)
 assign this.sheets := this.sheets - 1
 go busy concern JOB end

in heat on turn_on() assign this.tmp
  := this.tmp + 5
 go heat concern HEAT end
in heat if this.tmp > 20 on cool_down()
 assign this.tmp := 20 go heat
  concern HEAT end
```

A.2 Formal semantics

A.2.1 Formal definition of the OIL specification

The OIL specification given in appendix A.1, visualised in Fig. 1, is formally defined as the tuple $\langle\langle X, \mathcal{I}\rangle, \langle A, \sqsubset, \mathcal{RE}, \mathcal{EXP}\rangle, \langle E, \mathcal{PAR}, T, CO, \mathcal{CO}\rangle$ where

- $X = X_I \cup X_P$ where $X_I = \{\texttt{power}, \texttt{job}, \texttt{tmp}, \texttt{sheets}\}$ and $X_P = \{\texttt{nrsheets}, \texttt{sheetnr}\}$,
- $\mathcal{I}(\texttt{power}) = 'off'$, $\mathcal{I}(\texttt{job}) = 'idle'$, $\mathcal{I}(\texttt{tmp}) = 20$ and $\mathcal{I}(\texttt{sheets}) = 0$,
- $A = A_{Re} \cup A_{St} \cup A_{Sc}$ where $A_{Re} = \{a_{power}, a_{job}\}$, $A_{St} = \{a_{off}, a_{on}, a_{idle}, a_{busy}\}$ and $A_{Sc} = \{a_{power\_on}, a_{heat}\}$,
- $a_{job} \sqsubset a_{power\_on}, a_{off} \sqsubset a_{power}, a_{on} \sqsubset a_{power}, a_{idle} \sqsubset a_{job}$ and $a_{busy} \sqsubset a_{job}$,
- $\mathcal{RE}(a_{off}) = a_{power}$, $\mathcal{RE}(a_{on}) = a_{power}$, $\mathcal{RE}(a_{idle}) = a_{job}$, and $\mathcal{RE}(a_{busy}) = a_{job}$,
- $E = E_R \cup E_P$ where $E_R = \{\texttt{turn\_on}, \texttt{turn\_off}, \texttt{add\_job}, \texttt{cool\_down}\}$, $E_P = \{\backslash\texttt{sheet\_printed}\} \cup E_H$ and $E_H = \{\backslash\texttt{job\_printed}\}$,
- $\mathcal{PAR}(\texttt{add\_job}) = \{\texttt{nrsheets}\}$, $\mathcal{PAR}(\backslash\texttt{sheet\_printed}) = \{\texttt{sheetnr}\}$ and $\mathcal{PAR}(e) = \emptyset$ for all other events e,
- $T = \{t_1 = \langle a_{off}, true, \texttt{turn\_on}, \emptyset, \emptyset, a_{on}, true\rangle$, $t_2 = \langle a_{on}, true, \texttt{turn\_off}, \emptyset, \emptyset, a_{off}, true\rangle$, $t_3 = \langle a_{idle}, \texttt{nrsheets} > 0 \wedge \texttt{nrsheets} \leq 3, \texttt{add\_job}, \emptyset, \{\texttt{this.sheets} := \texttt{nrsheets}\}, a_{busy}, true\rangle$, $t_4 = \langle a_{busy}, \texttt{this.sheets} = 0, \backslash\texttt{job\_printed}, \emptyset, \emptyset, a_{idle}, true\rangle$, $t_5 = \langle a_{busy}, \texttt{this.sheets} > 0, \backslash\texttt{sheet\_printed}, \{(\texttt{sheetnr}, \texttt{this.sheets})\}, \{\texttt{this.sheets} := \texttt{this.sheets} - 1\}, a_{busy}, true\rangle$, $t_6 = \langle a_{heat}, true, \texttt{turn\_on}, \emptyset, \{\texttt{this.tmp} := \texttt{this.tmp} + 5\}, a_{heat}, true\rangle$, $t_7 = \langle a_{heat}, \texttt{this.tmp} > 20, \texttt{cool\_down}, \emptyset, \{\texttt{this.tmp} := 20\}, a_{heat}, true\rangle\}$,
- $CO = \{\texttt{POWER}, \texttt{JOB}, \texttt{HEAT}\}$ and
- $\mathcal{CO}(t_1) = \{\texttt{POWER}\}$, $\mathcal{CO}(t_2) = \{\texttt{POWER}\}$, $\mathcal{CO}(t_3) = \{\texttt{JOB}\}$, $\mathcal{CO}(t_4) = \{\texttt{JOB}\}$, $\mathcal{CO}(t_5) = \{\texttt{JOB}\}$, $\mathcal{CO}(t_6) = \{\texttt{HEAT}\}$, $\mathcal{CO}(t_7) = \{\texttt{HEAT}\}$.

A.3 mCRL2

See below for the mCRL2 specification generated from the OIL specification visualised in Fig. 1.

```
sort
  ISt = struct
    IS(GET_power : power_type, GET_job :
      job_type,
       GET_tmp : Int, GET_sheets : Int);
  power_type = struct power_off |
    power_on;
  job_type = struct job_no | job_ok;

map
  SET_power : ISt # Bool # power_type
      -> ISt;
  SET_job : ISt # Bool # job_type -> ISt;
  SET_tmp : ISt # Bool # Int -> ISt;
  SET_sheets : ISt # Bool # Int -> ISt;
var
  s : ISt, b : Bool, pow : power_type,
  job : job_type, tmp : Int, she : Int,
  u_pow : power_type, u_job : job_type,
  u_tmp : Int, u_she : Int;
eqn
  SET_power(s, false, u_pow) = s;
  SET_power(IS(pow, job, tmp, she),
   true, u_pow) =
    IS(u_pow, job, tmp, she);
  SET_job(s, false, u_job) = s;
  SET_job(IS(pow, job, tmp, she),
      true, u_job) =
    IS(pow, u_job, tmp, she);
  SET_tmp(s, false, u_tmp) = s;
  SET_tmp(IS(pow, job, tmp, she),
     true, u_tmp) =
    IS(pow, job, u_tmp, she);
  SET_sheets(s, false, u_she) = s;
  SET_sheets(IS(pow, job, tmp, she),
   true, u_she) =
    IS(pow, job, tmp, u_she);

map
  PPC : ISt -> Bool;
var
  s : ISt;
eqn
  PPC(s) = !((exists f1 : Bool, us : ISt.
      f1 == (GET_power(s) == power_on
                  &&
             GET_job(s) == job_ok &&
             GET_sheets(s) == 0)
                  && f1) ||
    exists f1 : Bool, us : ISt, sheetnr:
                        Int.
      f1 == (GET_power(s) == power_on &&
             GET_job(s) == job_ok &&
             GET_sheets(s) > 0 &&
             sheetnr == GET_sheets(s))
                 && f1);

act
```

```
  fail ;
  turn_off ;
  add_job: Int;
  job_printed ;
  sheet_printed: Int;
  turn_on ;
  cool_down ;

proc
  P(s : ISt, b : Bool) =
    !b -> fail . P(s, b) +

    sum f1 : Bool, us : ISt.
    (f1 == (GET_power(s) == power_on) &&
    us == SET_power(s, f1, power_off) &&
    b && f1 && PPC(s)) ->
    turn_off.P(us,
      f1 => GET_power(us)
          == power_off) +

    sum f1 : Bool, us : ISt, nrsheets:
          Int.
    (f1 == (GET_power(s) == power_on &&
            GET_job(s) == job_no &&
            nrsheets > 0 && nrsheets
                <= 3) &&
    us == SET_sheets(SET_job
                (s, f1, job_ok),
                     f1, nrsheets) &&
    b && f1 && PPC(s)) ->
    add_job(nrsheets).P(us, f1 =>
      GET_power(us) == power_on &&
      GET_job(us) == job_ok) +

    sum f1 : Bool, us : ISt.
    (f1 == (GET_power(s) == power_on &&
            GET_job(s) == job_ok &&
            GET_sheets(s) == 0) &&
    us == SET_job(s, f1, job_no) &&
    b && f1) ->
    job_printed.P(us,
      f1 => GET_power(us) == power_on &&
            GET_job(us) == job_no) +

    sum f1 : Bool, us : ISt, sheetnr :
                Int.
    (f1 == (GET_power(s) == power_on &&
            GET_job(s) == job_ok &&
            GET_sheets(s) > 0 &&
            sheetnr == GET_sheets(s)) &&
    us == SET_sheets(SET_job(s, f1,
              job_ok),
                     f1, GET_sheets(s)
                        - 1) &&
    b && f1) ->
    sheet_printed(sheetnr).P(us,
    f1 => GET_power(us) == power_on &&
          GET_job(us) == job_ok) +

    sum f1, f2 : Bool, us : ISt.
    (f1 == (GET_power(s) == power_off)
              &&
    f2 == (GET_tmp(s) < 45) &&
    us == SET_tmp(SET_power(s, f1,
                power_on),
                  f2, GET_tmp(s) + 5) &&
    b && (f1 && f2) && PPC(s)) ->
    turn_on.P(us,
      (f1 => GET_power(us) == power_on)
             &&
      (f2 => GET_tmp(us) < 45)) +

    sum f1 : Bool, us :
          ISt.
    (f1 == (GET_tmp(s) < 45 &&
          GET_tmp(s) > 20) &&
    us == SET_tmp(s, f1, 20) &&
    b && f1 && PPC(s)) ->
    cool_down.P(us, f1 => GET_tmp(us)
          < 45);

init P(IS(power_off, job_no, 20, 0),
            true);
```

B Proofs

B.1 Proofs for Section 6

Lemma 3 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$ and $v = s \cup p$. Then $u \in \mathcal{U}(T_e^v) \Leftrightarrow \exists_{b \in EXP_X} : (b, u) \in \hat{U}(e) \wedge [\![b]\!]v$.

Proof We prove the two implications separately:

$\Rightarrow$: Let $u \in \mathcal{U}(T_e^v)$ be some assignment. By Definition 20, there is a $t \in T_e^v$ such that $u \in \mathcal{U}(t)$. By the definition of T_e^v, we know that $t \in T_e$. Using this and Definition 28, it follows that $(\mathcal{PRC}(t), u) \in \hat{U}(e)$. By the definition of T_e^v we also know that $[\![\mathcal{PRC}(t)]\!]v$, from which we can conclude that $(b, u) \in \hat{U}(e) \wedge [\![b]\!]v$.

$\Leftarrow$: Let $(b, u) \in \hat{U}(e)$ be some pair. By Definition 28, there is a $t \in T_e$ such that $u \in \mathcal{U}(t)$ and $b = \mathcal{PRC}(t)$. Since $[\![b]\!]v$, we know that $[\![\mathcal{PRC}(t)]\!]v$ and therefore that $t \in T_e^v$

using the definition of T_e^v. Using Definition 20 we can conclude that $u \in \mathcal{U}(T_e^v)$.

Lemma 4 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $v = s \cup p$ and $v_\mathtt{S} = s_\mathtt{S} \cup p$. If $CP(v, \mathcal{U}(T_e^v))$, then $v[\mathcal{U}(T_e^v)](x) = [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S}$ for all $x \in X_I$.

Proof Pick some $x \in X_I$ and assume $CP(v, \mathcal{U}(T_e^v))$ holds. The definitions of $\mathcal{US}$ (Definition 29) and $\mathtt{SET}_x$ (Definition 27) show that setters for different instance variables do not influence each other, so only setters for x can influence the value of x after the update has been applied. Therefore, we only need to consider assignments to x. We distinguish two cases.

- Case: There exists no assignment $x := f \in \mathcal{U}(T_e^v)$. By Definition 5, this means that $v[\mathcal{U}(T_e^v)](x) = v(x)$. Using Lemma 3, we know that there is no $(b, x := f) \in \hat{U}(e)$ such that $[\![b]\!]v = \mathit{true}$. Knowing this, $\mathcal{US}$ does not change the value for x and therefore $[\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\mathtt{GET}_x(\mathtt{S})]\!]v_\mathtt{S}$. Since $v(x) = [\![\mathtt{GET}_x(\mathtt{S})]\!]v_\mathtt{S}$, we can conclude that $v[\mathcal{U}(T_e^v)](x) = [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S}$.
- Case: There exists an $x := f \in \mathcal{U}(T_e^v)$ for some $f \in EXP_X$. Since $CP(v, \mathcal{U}(T_e^v))$, we only need to consider one assignment $x := f$, as all assignments to x in $\mathcal{U}(T_e^v)$ result in the same value. By Definition 5, this means that $v[\mathcal{U}(T_e^v)](x) = [\![f]\!]v$.
  Using Lemma 3, we know that there is a $(b, x := f) \in \hat{U}(e)$ such that $[\![b]\!]v = \mathit{true}$ for each $x := f \in \mathcal{U}(T_e^v)$. In the construction of $\mathcal{US}(\hat{U}(e), \mathtt{S})$ (Definition 29), each right-hand side of an assignment $x := f$ is translated to an mCRL2 expression $\sigma_\mathtt{S}(f)$ and then used to overwrite the entry in the instance struct for x. Since $CP(v, \mathcal{U}(T_e^v))$ and $[\![f]\!]v = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$, we know that in the mCRL2 context we too only need to consider one assignment $x := f$, as all assignments will result in the same value. With only this assignment in mind, it follows that $[\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$. Since $[\![f]\!]v = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$ we can conclude that $v[\mathcal{U}(T_e^v)](x) = [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S}$.

Lemma 5 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $v = s \cup p$ and $v_\mathtt{S} = s_\mathtt{S} \cup p$. Then $CP(v, \mathcal{U}(T_e^v)) \Leftrightarrow [\![\bigwedge_{t \in T_e} \sigma_\mathtt{S}(\mathcal{PRC}(t)) \Rightarrow \mathcal{CP}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S}$.

Proof First we rewrite the right-hand side using that $[\![f]\!]v = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$ for any $f \in EXP_X$, $T_e^v = \{t \in T_e \mid [\![\mathcal{PRC}(t)]\!]v\}$ and using Definition 30 and 20 (in that order):

$$
\begin{aligned}
&[\![\bigwedge_{t \in T_e} \sigma_\mathtt{S}(\mathcal{PRC}(t)) \Rightarrow \mathcal{CP}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} \\
&= \bigwedge_{t \in T_e} [\![\sigma_\mathtt{S}(\mathcal{PRC}(t))]\!]v_\mathtt{S} \Rightarrow [\![\mathcal{CP}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} \\
&= \bigwedge_{t \in T_e^v} [\![\mathcal{CP}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} \\
&= \bigwedge_{t \in T_e^v} \bigwedge_{x := f \in \mathcal{U}(t)} [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S} \\
&= \bigwedge_{x := f \in \mathcal{U}(T_e^v)} [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}
\end{aligned}
$$

What is left to prove is that $CP(v, \mathcal{U}(T_e^v)) \Leftrightarrow \bigwedge_{x := f \in \mathcal{U}(T_e^v)} [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$. We prove the two implications separately:

$\Rightarrow$: Since $CP(v, \mathcal{U}(T_e^v))$ we know for every two assignments $x := f, x := g \in \mathcal{U}(T_e^v)$ that $[\![f]\!]v = [\![g]\!]v$. By Definition 5, this implies that the updated valuation $v[\mathcal{U}(T_e^v)]$ exists such that for every $x := f \in \mathcal{U}(T_e^v)$ we have that $v[\mathcal{U}(T_e^v)](x) = [\![f]\!]v$. Using Lemma 4 and using that $[\![f]\!]v = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$, we can rewrite this to $[\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$ which is what we needed to show.

$\Leftarrow$: We prove the contrapositive. Since $\neg CP(v, \mathcal{U}(T_e^v))$, there must exist two assignments $x := f, x := g \in \mathcal{U}(T_e^v)$ such that $[\![f]\!]v \neq [\![g]\!]v$. Because $x := f, x := g \in \mathcal{U}(T_e^v)$, the conjunction $\bigwedge_{x := f \in \mathcal{U}(T_e^v)} [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$ contains the two terms $[\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$ and $[\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(g)]\!]v_\mathtt{S}$. From $[\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S} = [\![f]\!]v$, $[\![\sigma_\mathtt{S}(g)]\!]v_\mathtt{S} = [\![g]\!]v$ and $[\![f]\!]v \neq [\![g]\!]v$ it follows that $[\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} \neq [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S}$, which is not possible. Therefore, it cannot be that $\bigwedge_{x := f \in \mathcal{U}(T_e^v)} [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$ holds, which is what we needed to show.

We have shown that $CP(v, \mathcal{U}(T_e^v)) \Leftrightarrow \bigwedge_{x := f \in \mathcal{U}(T_e^v)} [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S} = [\![\sigma_\mathtt{S}(f)]\!]v_\mathtt{S}$, from which we can conclude that $CP(v, \mathcal{U}(T_e^v)) \Leftrightarrow [\![\bigwedge_{t \in T_e} \sigma_\mathtt{S}(\mathcal{PRC}(t)) \Rightarrow \mathcal{CP}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S}$.

Lemma 6 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $e \in E$, $s \in \mathbb{V}^{X_I}$, $p \in \mathbb{V}^{\mathcal{PAR}(e)}$, $v = s \cup p$ and $v_\mathtt{S} = s_\mathtt{S} \cup p$. If $CP(v, \mathcal{U}(T_e^v))$, then $[\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]} \Leftrightarrow [\![\bigwedge_{t \in T_e} \sigma_\mathtt{S}(\mathcal{PRC}(t)) \Rightarrow \mathcal{POC}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_\mathtt{S}$.

Proof Using Definition 21 and using the definition of T_e^v (in that order), we can derive:

$$\begin{aligned}[\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]} &= \bigwedge_{t\in T_e^v} [\![\mathcal{POC}(t)]\!]^v_{v[\mathcal{U}(T_e^v)]} \\ &= \bigwedge_{t\in T_e} [\![\mathcal{PRC}(t)]\!]v \Rightarrow [\![\mathcal{POC}(t)]\!]^v_{v[\mathcal{U}(T_e^v)]}\end{aligned}$$

We can rewrite the right-hand side of the lemma to a similar form:

$$\begin{aligned}&[\![\bigwedge_{t\in T_e} \sigma_{\mathtt{s}}(\mathcal{PRC}(t)) \Rightarrow \mathcal{POC}(t,\mathtt{s},\mathcal{US}(\hat{U}(e),\mathtt{s}))]\!]v_{\mathtt{s}} \\ =\ &\bigwedge_{t\in T_e} [\![\sigma_{\mathtt{s}}(\mathcal{PRC}(t))]\!]v_{\mathtt{s}} \Rightarrow [\![\mathcal{POC}(t,\mathtt{s},\mathcal{US}(\hat{U}(e),\mathtt{s}))]\!]v_{\mathtt{s}} \\ =\ &\bigwedge_{t\in T_e} [\![\sigma_{\mathtt{s}}(\mathcal{PRC}(t))]\!]v_{\mathtt{s}} \Rightarrow [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(\mathcal{POC}(t))]\!]v_{\mathtt{s}}\end{aligned}$$

Since $[\![\mathcal{PRC}(t)]\!]v = [\![\sigma_{\mathtt{s}}(\mathcal{PRC}(t))]\!]v_{\mathtt{s}}$, we only need to prove that $[\![\mathcal{POC}(t)]\!]^v_{v[\mathcal{U}(T_e^v)]} \Leftrightarrow [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(\mathcal{POC}(t))]\!]v_{\mathtt{s}}$. We prove this by proving that $[\![\mathcal{POC}(t)]\!]^v_{v[\mathcal{U}(T_e^v)]} = [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(\mathcal{POC}(t))]\!]v_{\mathtt{s}}$ by induction on the structure of $\mathcal{POC}(t)$. In case the expression is a constant c, we can derive that $[\![c]\!]^v_{v[\mathcal{U}(T_e^v)]} = [\![c]\!] = [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(c)]\!]v_{\mathtt{s}}$. In case the expression is a variable $x \in \mathcal{PAR}(e)$, we can derive that $[\![x]\!]^v_{v[\mathcal{U}(T_e^v)]} = v(x) = [\![x]\!]v_{\mathtt{s}} = [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(x)]\!]v_{\mathtt{s}}$. In case the expression is a variable $x \in X_I$, we can derive using Lemma 4 that $[\![x]\!]^v_{v[\mathcal{U}(T_e^v)]} = v[\mathcal{U}(T_e^v)](x) = [\![\mathtt{GET}_x(\mathcal{US}(\hat{U}(e),\mathtt{s}))]\!]v_{\mathtt{s}} = [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(x)]\!]v_{\mathtt{s}}$. In case the expression is a variable x^{old} for some $x \in X_I$, we can derive that $[\![x^{old}]\!]^v_{v[\mathcal{U}(T_e^v)]} = v(x) = [\![\mathtt{GET}_x(\mathtt{s})]\!]v_{\mathtt{s}} = [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(x)]\!]v_{\mathtt{s}}$. As for the inductive step, in case the expression is an operator $op(f_1,..,f_n)$, assuming that $[\![f_i]\!]^v_{v[\mathcal{U}(T_e^v)]} = [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(f_i)]\!]v_{\mathtt{s}}$ for all $1 \le i \le n$, we can derive that $[\![op(f_1,..,f_n)]\!]^v_{v[\mathcal{U}(T_e^v)]} = [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(op(f_1,..,f_n))]\!]v_{\mathtt{s}}$

Since we have shown that $[\![\mathcal{POC}(t)]\!]^v_{v[\mathcal{U}(T_e^v)]} \Leftrightarrow [\![\sigma^{\mathtt{s}}_{\mathcal{US}(\hat{U}(e),\mathtt{s})}(\mathcal{POC}(t))]\!]v_{\mathtt{s}}$, we can conclude that $[\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]} \Leftrightarrow [\![\bigwedge_{t\in T_e} \sigma_{\mathtt{s}}(\mathcal{PRC}(t)) \Rightarrow \mathcal{POC}(t,\mathtt{s},\mathcal{US}(\hat{U}(e),\mathtt{s}))]\!]v_{\mathtt{s}}$.

Theorem 1 Let $\langle \mathbb{X},\mathbb{A},\mathbb{T}\rangle$ be an OIL specification. Let $\langle S,s_0,I,O,H,\rightarrow\rangle$ be the IOLTS that describes the acceptor semantics of this OIL specification (Definition 23). Let $\langle S',s'_0,L',\rightarrow'\rangle$ be the LTS that corresponds to the LPE of `P` (Definition 26) where `P(is, true)` describes the acceptor semantics of this OIL specification in mCRL2 (Definition 31). Then $s_0 \leftrightarrows s'_0$.

Proof For the sake of notation, we denote states in S', which are valuations over the process parameters `s` and `b`, as $(s_{\mathtt{s}},b)$ for any $s \in \mathbb{V}^{X_I}$ and $b \in \mathbb{B}$, where $s_{\mathtt{s}}$ is the evaluation for `s` such that $s(x) = [\![\mathtt{GET}_x(\mathtt{s})]\!]s_{\mathtt{s}}$ and b is the value for `b`. Let $R = \{(s,(s_{\mathtt{s}},true)) \mid s \in \mathbb{V}^{X_I}\} \cup \{((s_{\mathtt{s}},true),s) \mid s \in \mathbb{V}^{X_I}\} \cup \{(Ⓕ,(s_{\mathtt{s}},false)) \mid s \in \mathbb{V}^{X_I}\} \cup \{((s_{\mathtt{s}},false),Ⓕ) \mid s \in \mathbb{V}^{X_I}\}$. Note that R is symmetric and that $s_0 R s'_0$. We show that R is a strong bisimulation relation (see Definition 15). We distinguish three cases.

- Case $sR(s_{\mathtt{s}},true)$ for some $s \in \mathbb{V}^{X_I}$. If there is some $a \in L$ and $s' \in S_Ⓕ$ such that $s \xrightarrow{a} s'$, then according to Definition 23 there must be some $e \in E$ and $p \in \mathbb{V}^{\mathcal{PAR}(e)}$ such that $a = e(p)$ and $[\![CC(e)]\!]v$ where $v = s \cup p$. Since $[\![CC(e)]\!]v = [\![\sigma_{\mathtt{s}}(CC(e))]\!]v_{\mathtt{s}}$ for $v_{\mathtt{s}} = s_{\mathtt{s}} \cup p$, using Definition 26 and 31, there must also be a $(u_{\mathtt{s}},b) \in S'$ such that $(s_{\mathtt{s}},true) \xrightarrow{e([\![\bar{p^e}]\!]v_{\mathtt{s}})}{}' (u_{\mathtt{s}},b)$ for some $u \in \mathbb{V}^{X_I}$ and $b \in \mathbb{B}$. Since $\bar{p^e}$ is a vector of all parameters in $\mathcal{PAR}(e)$, it follows that $[\![\bar{p^e}]\!]v_{\mathtt{s}}$ corresponds to p and therefore $e([\![\bar{p^e}]\!]v_{\mathtt{s}}) = a$. To show that this case satisfies the strong bisimulation condition, all is left to show is that $s'R(u_{\mathtt{s}},b)$. We distinguish between two cases.
  - Case $CP(v,\mathcal{U}(T_e^v)) \wedge [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$. From Definition 23 it then follows that $s' = v[\mathcal{U}(T_e^v)]|_{X_I}$. Using Definition 26 and 31 we can see that $u_{\mathtt{s}}$ corresponds to the evaluated instance struct $[\![\mathcal{US}(\hat{U}(e),\mathtt{s})]\!](s_{\mathtt{s}} \cup p) = [\![\mathcal{US}(\hat{U}(e),\mathtt{s})]\!]v_{\mathtt{s}}$. Then using Lemma 4 we can derive that $u = s'$. Also, since $CP(v,\mathcal{U}(T_e^v)) \wedge [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$, using Corollary 1, it follows that $[\![\bigwedge_{t\in T_e} \mathcal{PCP}(t,\mathtt{s},\mathcal{US}(\hat{U}(e),\mathtt{s}))]\!]v_{\mathtt{s}}$. Then using Definition 26 and 31 it follows that $b = true$. Since $(s',(s'_{\mathtt{s}},true)) \in \{(s,(s_{\mathtt{s}},true)) \mid s \in \mathbb{V}^{X_I}\}$, we can conclude that $s'R(u_{\mathtt{s}},b)$.
  - Case $\neg CP(v,\mathcal{U}(T_e^v)) \vee \neg [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$. From Definition 23 it then follows that $s' = Ⓕ$. Using Corollary 1 we have that $\neg [\![\bigwedge_{t\in T_e} \mathcal{PCP}(t,\mathtt{s},\mathcal{US}(\hat{U}(e),\mathtt{s}))]\!]v_{\mathtt{s}}$. Then using Definition 26 and 31 it follows that $b = false$. Since $(Ⓕ,(u_{\mathtt{s}},false)) \in \{(Ⓕ,(s_{\mathtt{s}},false)) \mid s \in \mathbb{V}^{X_I}\}$, we can conclude that $s'R(u_{\mathtt{s}},b)$.
- Case $(s_{\mathtt{s}},true)Rs$ for some $s \in \mathbb{V}^{X_I}$. If there is some $a \in L$ and $(s'_{\mathtt{s}},b) \in S'$ such that $(s_{\mathtt{s}},true) \xrightarrow{a}{}' (s'_{\mathtt{s}},b)$, according to Definition 26 and 31 there must be some $e \in E$ and $p \in \mathbb{V}^{\mathcal{PAR}(e)}$ such that $a = e([\![\bar{p^e}]\!]v_{\mathtt{s}})$ and $[\![\sigma_{\mathtt{s}}(CC(e))]\!]v_{\mathtt{s}}$ where $v_{\mathtt{s}} = s_{\mathtt{s}} \cup p$. Since $[\![\sigma_{\mathtt{s}}(CC(e))]\!]v_{\mathtt{s}} = [\![CC(e)]\!]v$ for $v = s \cup p$, using Definition 23, there must also be a $u \in S$ such that $s \xrightarrow{e(p')} u$ for some $u \in \mathbb{V}^{X_I}$ and $p' \in \mathbb{V}^{\mathcal{PAR}(e)}$. Since $\bar{p^e}$ is a vector of

all parameters in $\mathcal{PAR}(e)$, it follows that p' corresponds to $[\![\bar{p^e}]\!]v_{\mathtt{S}}$ (and $p = p'$) and therefore $e(p') = a$. To show that this case satisfies the strong bisimulation condition, all is left to show is that $(s'_{\mathtt{S}}, b)Ru$. We distinguish between two cases.

- Case $b = true$. Using Definition 26 and 31 it follows that $[\![\bigwedge_{t\in T_e} \mathcal{PCP}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_{\mathtt{S}}$ and that s' corresponds to the evaluated instance struct $[\![\mathcal{US}(\hat{U}(e), \mathtt{S})]\!](s_{\mathtt{S}} \cup p) = [\![\mathcal{US}(\hat{U}(e), \mathtt{S})]\!]v_{\mathtt{S}}$. Then using Lemma 4 we can derive that $s' = v[\mathcal{U}(T_e^v)]|_{X_I}$. Also, using Corollary 1 we know that $CP(v, \mathcal{U}(T_e^v)) \wedge [\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$. From Definition 23 it then follows that $u = v[\mathcal{U}(T_e^v)]|_{X_I} = s'$. Since $((s'_{\mathtt{S}}, true), s') \in \{((s_{\mathtt{S}}, true), s) \mid s \in \mathbb{V}^{X_I}\}$, we can conclude that $(s'_{\mathtt{S}}, b)Ru$.
- Case $b = false$. From Definition 26 and 31 it follows that $\neg[\![\bigwedge_{t\in T_e} \mathcal{PCP}(t, \mathtt{S}, \mathcal{US}(\hat{U}(e), \mathtt{S}))]\!]v_{\mathtt{S}}$. Using Corollary 1 we then know that $\neg CP(v, \mathcal{U}(T_e^v)) \vee \neg[\![\mathcal{POC}(T_e^v)]\!]^v_{v[\mathcal{U}(T_e^v)]}$. From Definition 23 it then follows that $u = \text{Ⓕ}$. Since $((s'_{\mathtt{S}}, false), \text{Ⓕ}) \in \{((s_{\mathtt{S}}, false), \text{Ⓕ}) \mid s \in \mathbb{V}^{X_I}\}$, we can conclude that $(s'_{\mathtt{S}}, b)Ru$.

- Case $\text{Ⓕ}R(s_{\mathtt{S}}, false)$ or $(s_{\mathtt{S}}, false)R\text{Ⓕ}$ for some $s \in \mathbb{V}^{X_I}$. According to Definition 23 the only transition possible from Ⓕ is $\text{Ⓕ} \xrightarrow{\mathtt{fail}} \text{Ⓕ}$. According to Definition 26 and 31 the only transition possible from $(s_{\mathtt{S}}, false)$ is $(s_{\mathtt{S}}, false) \xrightarrow{\mathtt{fail}}' (s_{\mathtt{S}}, false)$. Since both transitions have the same action and since the target states of these transitions are related again (because they are the same as the source), we can conclude that R is a strong bisimulation relation in this case.

Since we have shown that every element satisfies the strong bisimulation condition, we know that R is a strong bisimulation relation. Since $s_0 R s'_0$, we can conclude that $s_0 \leftrightarrows s'_0$.

Theorem 2 Let $\langle \mathbb{X}, \mathbb{A}, \mathbb{T} \rangle$ be an OIL specification. Let $\langle S, s_0, I, O, H, \rightarrow \rangle$ be the IOLTS that describes the execution semantics of this OIL specification (Definition 24). Let $\langle S', s'_0, L', \rightarrow' \rangle$ be the LTS that corresponds to the LPE of `P` (Definition 26) where `P(is, true)` describes the execution semantics of this OIL specification in mCRL2 (Definition 33). Then $s_0 \leftrightarrows s'_0$.

Proof The only difference from Theorem 1, is that transitions in the IOLTS are additionally restricted by γ (Definition 24) and that transitions from the mCRL2 process are additionally restricted by $\mathcal{PPC}(e)$. Therefore, it suffices to prove that γ and $\mathcal{PPC}(e)$ put the same restrictions on the underlying LTSs. The rest of the proof can be reused from Theorem 1.

Let $s \in \mathbb{V}^X$, let $e \in E$ and let $a = e(p)$ for some $p \in \mathbb{V}^{\mathcal{PAR}(e)}$. Then we need to prove that $(a \in I \Rightarrow s \xrightarrow{O\cup H}\!\!\!\!\!\!/\ ) \Leftrightarrow [\![\mathcal{PPC}(e)]\!]s_{\mathtt{S}}$. In the case that $e \in E_P$, it can trivially be shown that this is true, since both sides are then equal to $true$. In case that $e \in E_R$, we know that $a \in I$ and we can rewrite the left-hand side:

$$s \xrightarrow{O\cup H}\!\!\!\!\!\!/\ = \neg\exists_{a\in O\cup H} : s \xrightarrow{a}$$
$$= \neg\exists_{e\in E_P, p\in\mathbb{V}^{\mathcal{PAR}(e)}} : s \xrightarrow{e(p)}$$

From Definition 23 one can derive that $s \xrightarrow{e(p)}$ iff $[\![\mathcal{CC}(e)]\!](s \cup p)$ and thus we can rewrite further to $\neg\exists_{e\in E_P, p\in\mathbb{V}^{\mathcal{PAR}(e)}} : [\![\mathcal{CC}(e)]\!](s \cup p)$.

We can rewrite the right-hand side of the bi-implication as follows:

$$[\![\mathcal{PPC}(e)]\!]s_{\mathtt{S}} = [\![\neg \bigvee_{e'\in E_P} \exists_{\bar{p^{e'}}:\tau^{e'}} : \sigma_{\mathtt{S}}(\mathcal{CC}(e'))]\!]s_{\mathtt{S}}$$
$$= \neg\exists_{e'\in E_P, p'\in\mathbb{V}^{\mathcal{PAR}(e')}} : [\![\mathcal{CC}(e')]\!](s \cup p')$$

Since we have been able to rewrite both sides of the bi-implication to the same formula, we can conclude that $(a \in I \Rightarrow s \xrightarrow{O\cup H}\!\!\!\!\!\!/\ ) \Leftrightarrow [\![\mathcal{PPC}(e)]\!]s_{\mathtt{S}}$, which was all we needed to prove the theorem.

B.2 Proofs for Section 7

Lemma 7 Let $\langle S, s_0, L, \rightarrow \rangle$ be an LTS, $s \in S$ some state, $L' \subseteq L$ some set of action labels and ϕ be some closed mu-calculus formula. Then $s \models [L'^*]\phi \Leftrightarrow \forall_{t\in S^{s,L'}_R} : t \models \phi$.

Proof See [1], Theorem 6.2.

Lemma 8 Let $\langle S, s_0, I, O, H, \rightarrow \rangle$ be the IOLTS$^\sqcap$ that describes the execution semantics of an OIL specification. Then $R1 \Leftrightarrow s_0 \models \phi_{R1}$.

Proof Using lemma 7 and the definition of the modal box operator we can rewrite:

$$s_0 \models \phi_{R1} = \forall_{s\in S_R} : s \models [O \cup H][\mathtt{fail}]false$$
$$= \forall_{s,t,u\in S_R, a\in O\cup H} : s \xrightarrow{a} t \xrightarrow{\mathtt{fail}} u \Rightarrow u \models false$$
$$= \neg\exists_{s,t,u\in S_R, a\in O\cup H} : s \xrightarrow{a} t \xrightarrow{\mathtt{fail}} u$$

Since `fail` can only be enabled in the failure state Ⓕ, we can rewrite further to $\neg\exists_{s\in S_R, a\in O\cup H} : s \xrightarrow{a} \text{Ⓕ}$, which is the definition of $R1$.

Lemma 9 Let $\langle S, s_0, I, O, H, \rightarrow \rangle$ be the IOLTS$^\sqcap$ that describes the execution semantics of an OIL specification. Then $R2 \Leftrightarrow s_0 \models \phi_{R2}$.

Proof Using lemma 7 we can rewrite:

$$s_0 \models \phi_{R2} = \forall_{s \in S_R} : s \models \mu Z.[O \cup U]Z$$

Now we only need to prove for any state $s \in S_R$ that:

$$\neg\exists_{w \in (O \cup H)^\omega} : s \xrightarrow{w}{}^{\omega} \Leftrightarrow s \models \mu Z.[O \cup U]Z$$

For the rest of the proof we refer to [14], Section 6.1, Lemma 9 through 12, which prove the dual.

Lemma 10 Let $\langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS$^{\sqcap}$ that describes the execution semantics of an OIL specification. Then $R3 \Leftrightarrow s_0 \models \phi_{R3}$.

Proof Let η be some fixpoint valuation and v some data valuation. Using Lemma 7, it is left to show for any $s \in S_R$ that:

$$\begin{aligned}&\forall_{w,w' \in (O \cup H)^*, t, t' \in S_\delta} : s \xrightarrow{w}{}^* t \wedge s \xrightarrow{w'}{}^* t' \Rightarrow t \leftrightarrows t' \\ \Leftrightarrow\ & s \in [\![\bigvee_{a \in Q} [(O \cup H)^*]([O \cup H]false \Rightarrow \langle a\rangle true)]\!]\eta v\end{aligned}$$

The right-hand side can be rewritten as follows:

$$\begin{aligned}&s \in [\![\bigvee_{a \in Q} [(O \cup H)^*]([O \cup H]false \Rightarrow \langle a\rangle true)]\!]\eta v \\ \Leftrightarrow\ & \exists_{a \in Q} : s \in [\![[(O \cup H)^*]([O \cup H]false \Rightarrow \langle a\rangle true)]\!]\eta v\end{aligned}$$

Before the mu-calculus formula is checked, the LTS is first reduced modulo strong bisimulation. Afterwards, each quiescent state is marked with a unique action a by adding a self-loop with this action. Using this information and using that bisimulation is an equivalence, we can rewrite:

$$\begin{aligned}&\forall_{w,w' \in (O \cup H)^*, t, t' \in S_\delta} : s \xrightarrow{w}{}^* t \wedge s \xrightarrow{w'}{}^* t' \Rightarrow t \leftrightarrows t' \\ \Leftrightarrow\ & \exists_{t \in S_\delta} : \forall_{w \in (O \cup H)^*, t' \in S_\delta} : s \xrightarrow{w}{}^* t' \Rightarrow t \leftrightarrows t \\ \Leftrightarrow\ & \exists_{a \in Q} : \forall_{w \in (O \cup H)^*, t' \in S_\delta} : s \xrightarrow{w}{}^* t \Rightarrow t \xrightarrow{a}\end{aligned}$$

What is left to prove is that for any $s \in S_R$ and for some $a \in Q$ that:

$$\begin{aligned}&\forall_{w \in (O \cup H)^*, t \in S_\delta} : s \xrightarrow{w}{}^* t \Rightarrow t \xrightarrow{a} t \\ \Leftrightarrow\ & s \in [\![[(O \cup H)^*]([O \cup H]false \Rightarrow \langle a\rangle true)]\!]\eta v\end{aligned}$$

Using the definition of reachability we can rewrite the left-hand side to:

$$\begin{aligned}&\forall_{w \in (O \cup H)^*, t \in S_\delta} : s \xrightarrow{w}{}^* t \Rightarrow t \xrightarrow{a} \\ \Leftrightarrow\ & \forall_{w \in (O \cup H)^*, t \in S} : s \xrightarrow{w}{}^* t \Rightarrow (t \in S_\delta \Rightarrow t \xrightarrow{a}) \\ \Leftrightarrow\ & \forall_{t \in S_R^{s, O \cup H}} : t \in S_\delta \Rightarrow t \xrightarrow{a}\end{aligned}$$

Using lemma 7, we only need to prove for any $s \in S_R$ for some $a \in Q$ for any $t \in S_R^{s, O \cup H}$ that:

$$(t \in S_\delta \Rightarrow t \xrightarrow{a}) \Leftrightarrow t \in [\![[O \cup H]false \Rightarrow \langle a\rangle true]\!]\eta v$$

We can rewrite the right-hand side to $t \models [O \cup H]false \Rightarrow t \models \langle a\rangle true$. By definition of quiescence, we know that $t \in S_\delta \Leftrightarrow t \models [O \cup H]false$. Then it is only left to prove that $t \xrightarrow{a} \Leftrightarrow t \models \langle a\rangle true$. This follows from the definition of the diamond operator and we can therefore conclude that $R3 \Leftrightarrow s_0 \models \phi_{R3}$.

Lemma 11. Let $\langle S, s_0, I, O, H, \rightarrow\rangle$ be the IOLTS$^{\sqcap}$ that describes the execution semantics of an OIL specification. Then $R4 \Leftrightarrow s_0 \models \phi_{R4}$.

Proof Using Lemma 7, it is left to show for any $s \in S_R$ that:

$$\begin{aligned}&\forall_{w,w' \in (O \cup H)^*, t, t' \in S_\delta} : s \xrightarrow{w}{}^* t \wedge s \xrightarrow{w'}{}^* t' \Rightarrow w \approx w' \\ \Leftrightarrow\ & s \models \exists_{w : Bag(O \cup H)} : \nu X(w' : Bag(O \cup H) := \emptyset). \\ & \bigwedge_{a \in O \cup H} [a]X(w' + \{a\}) \wedge ([O \cup H]false \Rightarrow w = w')\end{aligned}$$

Using that $\approx$ is an equivalence, we can rewrite the left-hand side:

$$\begin{aligned}&\exists_{u \in (O \cup H)^*} : \forall_{w \in (O \cup H)^*, t \in S_\delta} : s \xrightarrow{w}{}^* t \Rightarrow u \approx w \\ \Leftrightarrow\ & \exists_{u \in Bag(O \cup H)} : \forall_{w \in Bag(O \cup H), t \in S_\delta} : s \xrightarrow{w}{}^* t \Rightarrow u = w\end{aligned}$$

where $s \xrightarrow{w}{}^* t$ for $w \in Bag(O \cup H)$ iff there is a sequence $w' \in (O \cup H)^*$ such that $s \xrightarrow{w'}{}^* t$ and w' consists of the same actions as in w. What is left to prove is that for any $s \in S_R$ and for some $u \in Bag(O \cup H)$:

$$\begin{aligned}&\forall_{w \in Bag(O \cup H), t \in S_\delta} : s \xrightarrow{w}{}^* t \Rightarrow u = w \\ \Leftrightarrow\ & s \in [\![\nu X(w' : Bag(O \cup H) := \emptyset). \bigwedge_{a \in O \cup H} [a]X(w' + \{a\}) \wedge \\ & \qquad ([O \cup H]false \Rightarrow w = w')]\!]\eta v[w := u]\end{aligned}$$

We can rewrite the left-hand side further:

$$\forall_{w \in Bag(O \cup H), t \in S} : s \xrightarrow{w}{}^* t \Rightarrow (t \in S_\delta \Rightarrow u = w)$$

Note that the right-hand side has a very similar structure to that of $[(O \cup H)^*]\phi$, except that the fixpoint operator and variable now have data. With a similar proof as that for Lemma 7 we can show for any mu-calculus formula ϕ and for some fixpoint valuation η and data valuation v that:

$$\begin{aligned}&s \in [\![\nu X(w' : Bag(O \cup H) := \emptyset). \\ & \qquad \bigwedge_{a \in O \cup H} [a]X(w' + \{a\}) \wedge \phi]\!]\eta v \\ \Leftrightarrow\ & \forall_{u' \in Bag(O \cup H), t \in S} : s \xrightarrow{u'}{}^* t \Rightarrow t \in [\![\phi]\!]\eta v[w' := u']\end{aligned}$$

The update of v with $w' := u'$ is due to the repeated application of $w' + \{a\}$ by the fixpoint variable X, starting with the empty multiset $\emptyset$, while traversing a path consistent with u'. Using this all left to prove is for any $s \in S_R$ for some $u \in Bag(O \cup H)$ for all $u' \in Bag(O \cup H)$ and $t \in S$ such that $s \xrightarrow{u'}{}^{*} t$ that:

$$(t \in S_\delta \Rightarrow u = u')$$
$$\Leftrightarrow t \in [\![[O \cup H]false \Rightarrow w = w']\!]\eta v[w := u, w' := u']$$

The right-hand side can be rewritten to $t \models [O \cup H]false \Rightarrow t \in [\![w = w']\!]\eta v[w := u, w' := u']$. Using the definition of quiescence we know that $t \in S_\delta \Leftrightarrow t \models [O \cup H]false$. Then it is only left to prove that $u = u' \Leftrightarrow t \in [\![w = w']\!]\eta v[w := u, w' := u']$, which is true since we can rewrite the right-hand side to $u = u'$, and we can therefore conclude that $R4 \Leftrightarrow s_0 \models \phi_{R4}$.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive systems: modelling, specification and verification. Cambridge university press (2007)
2. Axelsson, R., Lange, M., Somla, R.: The complexity of model checking higher-order fixpoint logic. Logical Methods in Comput Sci **3**(2) (2007)
3. Basile, D., ter Beek, M.H., Ferrari, A., Legay, A.: Modelling and analysing ERTMS L3 moving block railway signalling with simulink and uppaal SMC. In: FMICS, Lecture Notes in Computer Science, vol. 11687, pp. 1–21. Springer (2019)
4. ter Beek, M.H., Borälv, A., Fantechi, A., Ferrari, A., Gnesi, S., Löfving, C., Mazzanti, F.: Adopting formal methods in an industrial setting: The railways case. In: FM, Lecture Notes in Computer Science, vol. 11800, pp. 762–772. Springer (2019)
5. ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-based model checking with mCRL2. In: FASE, Lecture Notes in Computer Science, vol. 10202, pp. 387–405. Springer (2017)
6. Berger, U., James, P., Lawrence, A., Roggenbach, M., Seisenberger, M.: Verification of the european rail traffic management system in real-time maude. Sci. Comput. Program. **154**, 61–88 (2018)
7. van Beusekom, R., Groote, J.F., Hoogendijk, P.F., Howe, R., Wesselink, W., Wieringa, R., Willemse, T.A.C.: Formalising the Dezyne modelling language in mCRL2. In: FMICS-AVoCS, Lecture Notes in Computer Science, vol. 10471, pp. 217–233. Springer (2017)
8. Bienmüller, T., Damm, W., Wittke, H.: The STATEMATE verification environment - making it real. In: CAV, Lecture Notes in Computer Science, vol. 1855, pp. 561–567. Springer (2000)
9. Bouwman, M., Janssen, B., Luttik, B.: Formal modelling and verification of an interlocking using mCRL2. In: FMICS, Lecture Notes in Computer Science, vol. 11687, pp. 22–39. Springer (2019)
10. Bouwman, M., Luttik, B., van der Wal, D.: A formalisation of sysml state machines in mcrl2. In: FORTE, Lecture Notes in Computer Science, vol. 12719, pp. 42–59. Springer (2021)
11. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/xt 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1-2), 52–70 (2008)
12. Bunte, O., van Gool, L.C.M., Willemse, T.A.C.: Formal verification of OIL component specifications using mcrl2. In: FMICS, Lecture Notes in Computer Science, vol. 12327, pp. 231–251. Springer (2020)
13. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In: TACAS (2), Lecture Notes in Computer Science, vol. 11428, pp. 21–39. Springer (2019)
14. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
15. Cordy, M., Devroey, X., Legay, A., Perrouin, G., Classen, A., Heymans, P., Schobbens, P., Raskin, J.: A decade of featured transition systems. In: From Software Engineering to Formal Methods and Tools, and Back, Lecture Notes in Computer Science, vol. 11865, pp. 285–312. Springer (2019)
16. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - visual automated transformations for formal verification and validation of UML models. In: ASE, pp. 267–270. IEEE Computer Society (2002)
17. Damm, W., Klose, J.: Verification of a radio-based signaling system using the STATEMATE verification environment. Formal Methods Syst. Des. **19**(2), 121–141 (2001)
18. Denkers, J., van Gool, L., Visser, E.: Migrating custom DSL implementations to a language workbench (tool demo). In: SLE, pp. 205–209. ACM (2018)
19. Fernandez, J., Bozga, M., Ghirvu, L.: State space reduction based on live variables analysis. Sci. Comput. Program. **47**(2–3), 203–220 (2003)
20. Frenken, M.: Code generation and model-based testing in context of oil. Master's thesis, Eindhoven University of Technology (2019)
21. van Gool, L.: Formalising interface specifications. Ph.D. thesis, Eindhoven University of Technology (2006)
22. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press (2014)
23. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. Theor. Comput. Sci. **343**(3), 332–369 (2005)
24. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M.R., van de Pol, J.: Towards model checking executable UML specifications in mCRL2. ISSE **6**(1–2), 83–90 (2010)
25. Hwong, Y., Keiren, J.J.A., Kusters, V.J.J., Leemans, S.J.J., Willemse, T.A.C.: Formalising and analysing the control software of the compact muon solenoid experiment at the Large Hadron Collider. Sci. Comput. Program. **78**(12), 2435–2452 (2013)
26. Islam, M.A., Cleaveland, R., Fenton, F.H., Grosu, R., Jones, P.L., Smolka, S.A.: Probabilistic reachability for multi-parameter bifurcation analysis of cardiac alternans. Theor. Comput. Sci. **765**, 158–169 (2019)
27. Kernberger, D., Lange, M.: Model checking for hybrid branching-time logics. J. Log. Algebraic Methods Program. **110** (2020)
28. Kim, J.H., Larsen, K.G., Nielsen, B., Mikucionis, M., Olsen, P.: Formal analysis and testing of real-time automotive systems using UPPAAL tools. In: FMICS, Lecture Notes in Computer Science, vol. 9128, pp. 47–61. Springer (2015)
29. Kölbl, M., Leue, S.: Automated functional safety analysis of automated driving systems. In: FMICS, Lecture Notes in Computer Science, vol. 11119, pp. 35–51. Springer (2018)
30. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Asp. Comput. **11**(6), 637–664 (1999)
31. Limbrée, C., Cappart, Q., Pecheur, C., Tonetta, S.: Verification of railway interlocking - compositional approach with OCRA. In: RSSRail, Lecture Notes in Computer Science, vol. 9707, pp. 134–149. Springer (2016)
32. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. Laboratory for Computer Science, Massachusetts Institute of Technology (1988)

33. Mitsch, S., Gario, M., Budnik, C.J., Golm, M., Platzer, A.: Formal verification of train control with air pressure brakes. In: RSSRail, Lecture Notes in Computer Science, vol. 10598, pp. 173–191. Springer (2017)
34. van de Pol, J., Timmer, M.: State space reduction of linear processes using control flow reconstruction. In: ATVA, Lecture Notes in Computer Science, vol. 5799, pp. 54–68. Springer (2009)
35. Remenska, D., Templon, J., Willemse, T.A.C., Homburg, P., Verstoep, K., Ramo, A.C., Bal, H.E.: From UML to process algebra and back: An automated approach to model-checking software design artifacts of concurrent systems. In: NASA Formal Methods, Lecture Notes in Computer Science, vol. 7871, pp. 244–260. Springer (2013)
36. Sankaranarayanan, S., Kumar, S.A., Cameron, F., Bequette, B.W., Fainekos, G.E., Maahs, D.M.: Model-based falsification of an artificial pancreas control system. SIGBED Rev. **14**(2), 24–33 (2017)
37. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electron. Notes Theor. Comput. Sci. **55**(3), 357–369 (2001)
38. Schindler, E., Moneva, H., van Pinxten, J., van Gool, L., van der Meulen, B., Stotz, N., Theelen, B.: Jetbrains mps as core dsl technology for developing professional digital printers. In: Domain-Specific Languages in Practice, pp. 53–91. Springer (2021)
39. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: FMICS, Lecture Notes in Computer Science, vol. 9128, pp. 62–77. Springer (2015)
40. Silva, J.: A vocabulary of program slicing-based techniques. ACM Comput. Surv. **44**(3), 12:1–12:41 (2012)
41. Thévenod-Fosse, P., Waeselynck, H.: STATEMATE applied to statistical software testing. In: ISSTA, pp. 99–109. ACM (1993)
42. Toennemann, J., Rausch, A., Howar, F., Cool, B.: Checking consistency of real-time requirements on distributed automotive control software early in the development process using UPPAAL. In: FMICS, Lecture Notes in Computer Science, vol. 11119, pp. 67–82. Springer (2018)
43. Valmari, A.: Bisimilarity minimization in O(m logn) time. In: Petri Nets, Lecture Notes in Computer Science, vol. 5606, pp. 123–142. Springer (2009)
44. Visser, E., Wachsmuth, G., Tolmach, A.P., Neron, P., Vergu, V.A., Passalaqua, A., Konat, G.: A language designer's workbench: a one-stop-shop for implementation and verification of language designs. In: Onward!, pp. 95–111. ACM (2014)
45. Wang, H., Zhong, D., Zhao, T., Ren, F.: Integrating model checking with sysml in complex system safety analysis. IEEE Access **7**, 16561–16571 (2019)
46. Weiglhofer, M., Wotawa, F.: Asynchronous input-output conformance testing. In: COMPSAC (1), pp. 154–159. IEEE Computer Society (2009)
47. Zhang, S.J., Liu, Y.: An automatic approach to model checking UML state machines. In: SSIRI (Companion), pp. 1–6. IEEE Computer Society (2010)