Formal Verification of OIL Component Specifications using mCRL2

Olav Bunte[1](✉), Louis C. M. van Gool[2], and Tim A. C. Willemse[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
o.bunte@tue.nl

[2] Canon Production Printing, Venlo, The Netherlands

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 informal 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.

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 models in the development of control software is Canon Production Printing. To investigate the benefits of a *Model-Driven Engineering* approach to software engineering, a new language for modelling the behaviour of control software, called *Open Interaction Language* (OIL), was developed within the company.

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. With the use of dedicated tooling one can automatically generate efficient executable code from such models. 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

© Springer Nature Switzerland AG 2020
M. H. ter Beek and D. Ničković (Eds.): FMICS 2020, LNCS 12327, pp. 231–251, 2020.
https://doi.org/10.1007/978-3-030-58298-2_10

readable and unambiguous visual representation, which is often deemed an indispensable tool in discussions among engineers.

OIL was not specifically designed to allow for scalable formal analysis on models written in the language. It is therefore unclear to what extent it is feasible to analyse such models. In this paper, we set out to investigate exactly this question. Our contributions are as follows. We define a formal operational semantics for OIL and identify a number of validity requirements on OIL component specifications. These validity requirements ensure that code, generated from the OIL models, behaves reliably and predictably. To enable the use of model checking techniques, we have defined and implemented a translation from OIL component specifications to mCRL2 [17]. The latter is supported by a powerful toolset [10] offering model checking and equivalence checking facilities. Interestingly, the separation of concerns philosophy of OIL poses the biggest challenge in devising this translation. This is mainly due to the large semantical gap between OIL concepts and concepts typical to academic languages such as mCRL2. Our translation from OIL to mCRL2 is implemented in the Spoofax language workbench [36]. We have defined the validity requirements in terms of the μ-calculus so that they can be formally verified on OIL specifications. To test the feasibility of our methods, we have applied these techniques to some models of systems that are used in production at Canon Production Printing. Technical details can be found in [9]; in this paper we focus on the more salient aspects of the work.

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 [2,3,5,7,26,27], automotive [23,24,32,34] and biomedical [21,30]. The modelling languages 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 [7,12,19,24,25,29,31,37] and the references therein.

Works on modelling control software close to ours are those on the FSM language used at CERN [20] and on the Dezyne language developed by the company Verum [6]. 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 introduce OIL and its semantics informally by means of a small example and present the validity requirements. Using the same example, we show in Sect. 3 how OIL specifications are translated to mCRL2 and how to formally verify the validity requirements. In Sect. 4 we show the results of some experiments on OIL models of systems used in production. Lastly, we discuss our techniques and results in Sect. 5 and conclude in Sect. 6.

2 OIL

OIL (Open Interaction Language) was created by Van Gool within Canon Production Printing as a language to analyse and visualise the communication behaviour of software systems, partly based on [16]. Using dedicated tooling, one can visualise an OIL specification, analyse traces on it and generate executable code. Originally the syntax of OIL was based on XML. However, as XML is not very user friendly due its verbosity, a more compact syntax has been designed by Denkers [13]. Although OIL is a textual language, it was designed to have a readable yet unambiguous graphical representation. In the following section we will give an informal description of the core constructs of OIL by means of an example.

2.1 A Brief Introduction to OIL

Each OIL specification consists of a number of *global state variables*, *areas* and *transitions*. A *global state* assigns a value to each global state variable. Updating the global state is done by means of simultaneous assignment. Which updates are performed and when is determined by the areas and transitions.

Areas are organised in a tree structure, so an area is either a *root area* or has a *parent*. OIL distinguishes between three types of areas: *regions*, *states* and *scopes*. A region contains a collection of states and refers to a global state variable. A state in this region represents a value of the referred to global state variable, also called the *variable for this state*. A region is typically used to define an aspect of the behaviour of the system and the different states of that aspect. A scope contains a boolean expression that serves as an invariant. It is typically used to restrict possible behaviour.

Transitions have a *source* and *target* area and are labelled with an *event*. Optionally, a transition can have a guard, a collection of assignments and an assert. A transition between two states typically represents the update of a variable from the value of the source state to the value of the target state.

Example. Figure 1 depicts an example OIL component specification of a system with overheating issues, which will serve as a running example in this section. The example has three global state variables: `power`, `job` and `tmp`. The variable `power` models whether the system is switched off (`'off'`) or on (`'on'`), the variable `job` models whether the system is idle (`'idle'`) or busy handling a job (`'busy'`) and the integer variable `tmp` models the temperature of the system. The initial global state maps `power` to `'off'`, `job` to `'idle'` (as can be seen by a slight colouring of the corresponding states) and `tmp` to 20 (not shown in the figure). For brevity of notation, we will denote such a global state as $\langle$`'off'`, `'idle'`, $20\rangle$. Global state variables are prepended with the keyword '`this`' to indicate that these belong to the scope of the modelled component.

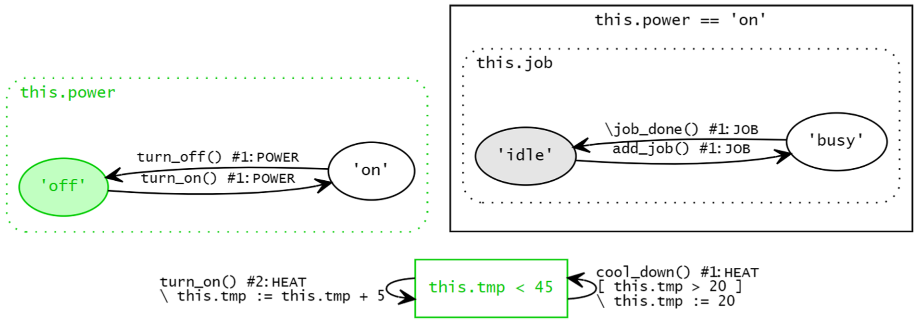

Fig. 1. The visualisation of an OIL specification of a system with overheating issues. (Colour figure online)

The example has eight areas: two regions, each containing two states, and two scopes. Regions are drawn as dotted boxes, states as ovals, scopes as solid boxes and transitions as arrows. Areas are directly contained in their parent area. The two regions refer to the variables `power` and `job` and contain states for each value of these variables. Each transition between two states updates a variable to its other value. The top right scope models that the system 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'`. The value of `tmp` can be updated by the assignments (the expressions preceded by a backslash) of the lower two transitions. The right transition of these two has a guard (the expression between square brackets) that requires the temperature to be more than 20. The scope these transitions have as source and target limit the temperature to be less than 45.

Each area is associated with a condition (the *area condition*) and an update (the *area update*). The area condition of a region is true iff it is a root area or the area condition of its parent area is true. For the area condition of a state we additionally require that the variable for this state has the value of this state, whereas for the area condition of a scope we additionally require that its invariant holds. We say that an area is *active* given a global state iff its area condition is true in this global state. The area update of a region or scope performs no assignments if it is a root area, else it equals the area update of its parent area. For a state the area update is extended with the assignment of the value of this state to the variable for this state.

Example. In the running example there are three active areas in the initial global 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 is active and the initial global state maps `power` to `'off'` (`power` = `'off'`). The bottom scope is active since it is a root area and its invariant is true. An example of an area update is the one for state `'off'`, which is `power` := `'off'`.

An update of the global state is triggered by the occurrence of an event. Whenever an event occurs, all transitions that are labelled with this event that can fire, do fire. A transition can fire if its *transition precondition* is true, that is when its source area is active and its guard is true in the current global state. When a transition fires, its *transition update* is applied, defined by the area update of its target area in combination with the transition's assignments. All updates of transitions that fire are applied simultaneously. If these updates try to change the same variable to two different values, we say that these updates are *incompatible*. This causes the event to fail, resulting in an inconsistent state. An event can also fail if after applying the transition updates the *transition postcondition* of one of its fired transitions is not met, that is if one of the fired transition's target area is inactive or assert is false.

Example. Suppose that in the initial global 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 can fire since their source areas are active. When they fire, `turn_on() #1` updates `power` to `'on'` and `turn_on() #2` updates `tmp` to `tmp` + 5, resulting in the global state ⟨`'on'`, `'idle'`, 25⟩. In this global state both target areas are active, since `power` = `'on'` and `tmp` < 45, and therefore the event succeeds.

It is possible for an event to fail in the running example. When `turn_on()` occurs in the global state ⟨`'off'`, `'idle'`, 40⟩, both transitions fire and result in the global state ⟨`'on'`, `'idle'`, 45⟩. Since in this resulting global state it does not hold that `tmp` < 45, transition `turn_on() #2` (and therefore the event `turn_on()`) fails. This failure models a crash of the system due to overheating. To make this restriction more explicit to the user of the system, a guard `[this.temp < 40]` can be added to `turn_on() #2`.

As mentioned in the introduction, OIL follows the *separation of concerns* philosophy. This philosophy enables one to model different aspects of the system separately, which helps keeping the OIL models of complex systems compact. The running example shows this philosophy well. There are three different parts of the specification that each model a different aspect of the system: the top left region models the power aspect, the top right region 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 top right part of the OIL model that models the handling of jobs can be easily replaced with a more refined one.

Such separate parts of an OIL model can interact with each other by means of references to global state variables, such as `power` referred to by both the top left region and the top right scope. They can also interact with each other by synchronising on the same event. Synchronisation occurs whenever separate parts of an OIL model contain transitions of the same event. When these transitions can fire and the corresponding event occurs, the transitions fire simultaneously, making these separate parts 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 of an event that can fire simultaneously. In OIL this is done by labelling transitions with 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 is labelled with that concern. Then an event may only occur if for each concern this event is part of, at least one of its transitions labelled with that concern can fire. We refer to this as the *concern condition*. If an event occurs and its concern condition is met, all transitions labelled with this event that can fire, do fire.

Example. In the running example there are three concerns defined, namely `POWER`, `JOB` and `HEAT`, shown after the event name on a transition. The two transitions of event `turn_on()` are both labelled with different concerns, namely `POWER` and `HEAT`, which makes event `turn_on()` only allowed if both transitions can fire. This forces synchronisation between the part modelling the power of the system with the part that models the temperature of the system, making it illegal to turn the system on when it is already on.

In OIL component specifications we distinguish between two types of events: *reactive* and *proactive* events. Reactive events are events that the component receives from the environment, whereas proactive events are events that the component produces itself, either internal or sent to the environment. This distinction determines in which way the code generated from an OIL specification is executed: when a component is running in an environment it uses a scheduler with run-to-completion semantics, which prioritises proactive events over reactive events. At the level of OIL's semantics, this effectively partitions the set of global states in a set of *quiescent* global states from which no proactive events are possible and a set of non-quiescent global states from which only (at least one) proactive events are possible. If there is a choice between multiple proactive events, the scheduler chooses arbitrarily.

Example. In the running example only event `job_done()` is proactive, which is indicated in the visualisation with a backslash preceding the event name. Whenever a component with this specification is in a global state where event `job_done()` can be produced, no other event is possible. Any other global state in this example is quiescent.

The formal semantics of OIL is defined by associating a Labelled Transition System (LTS) to each syntactically correct OIL model. The LTS for the example specification has a total of 16 reachable states and 28 reachable transitions. See Fig. 2 for a visualisation of this LTS.

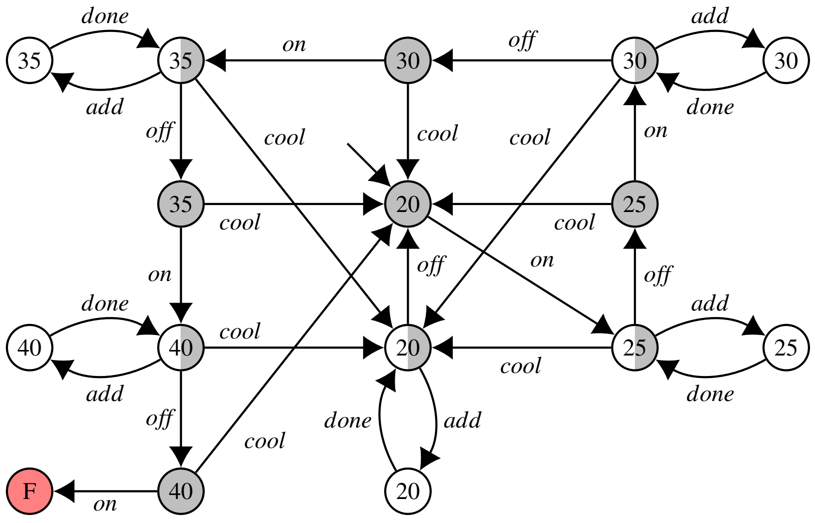

Fig. 2. The LTS that describes the semantics of the running example OIL specification of Fig. 1. The left half of a state is gray iff `power` = `'off'`, the right half of a state is gray iff `job` = `'idle'` and the value of `tmp` is written in the state. The red state with label F indicates that an event has failed. Action label *on* refers to event `turn_on()`, *off* to `turn_off()`, *add* to `add_job()`, *done* to `job_done()` and *cool* to `cool_down()`. (Colour figure online)

2.2 Validity of OIL Component Specifications

The scheduler that is used to execute the code generated from an OIL model is required to run as efficiently as is possible. This means that the scheduler performs only the most basic checks to prevent system crashes. To guarantee error-free code execution, engineers have adopted informal rules to which OIL models should adhere, and that help prevent such situations. In the process of formalising the semantics of OIL, we have formalised these rules as four validity requirements. These validity requirements are expressed as constraints on the set of paths permitted in the LTS underlying an OIL model. Below we give only an informal explanation of these requirements and the rationale behind them.

Requirement 1. *Safe lookaheadlessness:* Reachable proactive events should not fail.

When the scheduler checks which proactive events are possible, it only checks the transition preconditions and concerns. It does not consider the postconditions as this is computationally expensive, since it would require to apply the updates that correspond to the event and then roll back for every proactive event. By posing this requirement, we prevent the scheduler from possibly producing failing events. We do allow reactive events to fail, as this is considered inappropriate usage of the component by the environment.

Requirement 2. *Finite proactivity:* All reachable sequences of proactive events must be finite.

If a component reaches a state from which an infinite path of proactive events is possible, such as a loop, the scheduler will follow this path and never consider a reactive event. In the scope of a system of components, this would result in a component that never reacts to events from other components, effectively blocking the progress of other components.

Requirement 3. *Confluent proactivity:* In all reachable non-quiescent global states, all possible sequences of proactive events that end up in a quiescent global state must end up in bisimilar quiescent global states.

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 end up in behaviourally different quiescent global states, this non-determinism permeates the whole component, which is considered undesirable.

Requirement 4. *Predictable proactivity:* In all reachable non-quiescent global states, all possible sequences of proactive events that end up in a quiescent global state must consist of the same multi-set of events.

In case these routes of proactive behaviour consist of different events, it would mean that whether an event is produced or not is determined non-deterministically. This is undesired, as this event may be needed for other components to proceed. The scheduler is free to choose the order in which the events are produced however.

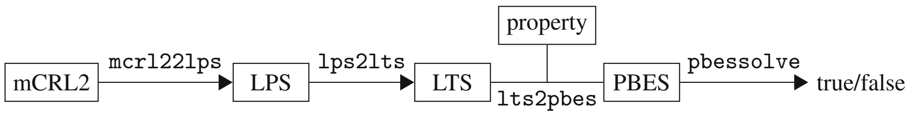

Fig. 3. One of the basic work flows in the mCRL2 toolset for generating an LTS and for checking a mu-calculus property. The edges are labelled with tool names.

3 Model Checking OIL Specifications with mCRL2

To enable the formal verification of OIL specifications we have formalised OIL's LTS semantics in the modelling language mCRL2 [17] by means of a translation. With the mCRL2 toolset [10] one can visualise and simulate an mCRL2 specification and apply model checking techniques to it, such as checking properties and equivalence. Properties can be stated in the modal μ-calculus extended with data [17]. See Fig. 3 for the workflow in the mCRL2 toolset that we use to analyse mCRL2 specifications.

In the following subsections we highlight the key parts of the translation from OIL to mCRL2 (Sect. 3.1), its implementation in Spoofax [36] (Sect. 3.2) and our formalisation of the four validity requirements (Sect. 3.3). We again use the OIL specification of Fig. 1 as running example.

3.1 Translation to mCRL2

To represent the global state we define a type `GS_type`, which is an object that contains all global state variables. To query or change the value of a variable in a global state we define getter and setter functions `GET_v` and `SET_v` for every global state variable `v`. For each area `a`, we define functions `AC_a` and `AU_a` for the area condition and area update respectively. To model that proactive events have priority over reactive events, we define the quiescence condition `QC`, which is true iff the current global state is quiescent. Reactive events are only allowed when the quiescence condition is true.

The process specification consists of two processes: the main recursive process `P` and a process `FAIL` that models event failure by means of a self-loop labelled with an action `failure`. Process `P` is a monolithic process with one parameter of type `GS_type` that represents the current global state and consists of a non-deterministic choice between so-called summands, each representing one event and its transitions. Each summand models what happens when that event occurs and is of the form:

$$PPC_e(s) \wedge CC_e(s) \rightarrow (POST_e(UPD_e(s)) \rightarrow e.\mathtt{P}(UPD_e(s)) \diamond e.\mathtt{FAIL})$$

where s is the global state parameter and e is the event this is the summand of. The operators $b \rightarrow p$ and $b \rightarrow p \diamond q$ are conditional process operators. For either operator, process p is executed iff condition b is true. For the latter operator, process q is executed iff b is false.

The proactive priority condition function $PPC_e(s)$ is true in case e is a proactive event, else it equals `QC`. The function $CC_e(s)$ is the concern condition, which can be easily encoded in mCRL2 using conjunctions and disjunctions over transition preconditions. The rest of the summand is more complex however. With $UPD_e(s)$ we update the current global state to a new one using the transition updates of the transitions of event e. The updates are applied in some fixed order by sequentially rewriting the global state for every update. A complication is that we need to use the original global state s and not the intermediate global state that is being updated for getting values of global state variables to correctly simulate the simultaneous update. Also, to only apply the updates of those transitions that fire, we need the transition preconditions. In mCRL2 we define this sequential update by nesting update functions.

The check whether transition updates are compatible poses another issue. We could check whether assignments to the same variable result in different values before updating the state. However, since we do not know beforehand what transitions of an event will fire, we need to check compatibility for every pair of transitions in the worst case, which can lead to a number of checks quadratic in the number of transitions. Instead, inspired by the C++ code generator from the original OIL tooling, we check compatibility after doing the update. This is done by creating a compatibility check $x == e$ for every assignment $x := e$. Now if two assignments are incompatible when updating the global state, the second assignment effectively overwrites the first, which will make the compatibility check of the first assignment false. These compatibility checks are performed in the function $POST_e(s)$, together with the transition postconditions of the transitions of event e. Similar to $UPD_e(s)$, the transition preconditions are used to only check the postconditions and compatibility checks of transitions that have fired.

Each summand models the following behaviour:

- If $PPC_e(s) \wedge CC_e(s) \wedge POST(UPD_e(s))$ holds, event e is enabled and after execution of e, recurse to `P` with the updated global state $UPD_e(s)$.
- If $PPC_e(s) \wedge CC_e(s) \wedge \neg POST(UPD_e(s))$ holds, event e is enabled and after execution of e, execute process `FAIL`.
- If $\neg(PPC_e(s) \wedge CC_e(s))$ holds, event e is not enabled.

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 typically eliminate such auxiliary variables. Remarkably, it is not able to so on mCRL2 specifications generated by the translation from OIL.

Example. See Fig. 4 for part of the main process `P` of the running example, showing only the summand for the event `turn_on()` with auxiliary variables. Line 4 corresponds to $PPC_e(s)$. On line 5 we define auxiliary variables `f1` and `f2` which represent the transition preconditions of the transitions `turn_on() #1` and `turn_on() #2` respectively. This is done using the `sum`-operator to declare the variables, followed by conditions to fix their values. The concern condition $CC_e(s)$ is checked on line 6. On lines 7–8 we define the auxiliary variable `uv_GS` which represents the updated state $UPD_e(s)$. Note that the update functions here are nested to update the global state sequentially. The variables `f1` and `f2` are supplied to the update functions to only apply the updates of transitions that can fire. The postconditions $POST_e(s)$ are shown on lines 9–10. Note that the compatibility check `GET_tmp(uv_GS) == GET_tmp(v_GS) + 5` is added due to the update `SET_tmp(.., f2, GET_tmp(v_GS) + 5)`. On line 11 the action `turn_on` is done and then the process recurses with the updated global state, or it starts the failure process, depending on whether the postconditions were true.

```
proc
  P(v_GS : GS_type) =
    ...
    QC(v_GS) ->
    sum f1, f2 : Bool.(f1 == AC_off(v_GS) && f2 == AC_heat(v_GS)) ->
    (f1 && f2) ->
    sum uv_GS : GS_type.(uv_GS == AU_heat(SET_tmp(AU_on(v_GS,
                                  f1), f2, GET_tmp(v_GS) + 5), f2)) ->
    ((f1 => AC_on(uv_GS)) &&
     (f2 => (AC_heat(uv_GS) && GET_tmp(uv_GS) == GET_tmp(v_GS) + 5)) ->
    turn_on.P(uv_GS) <> (turn_on.FAIL) +
    ...;
```

Fig. 4. Part of the main process P of the mCRL2 specification generated from the running example of Fig. 1, showing only the summand for the event `turn_on()` with auxiliary variables.

3.2 Implementation of the Translation

The translation from OIL to mCRL2 has been implemented in the Spoofax language workbench [36] using the model transformation language Stratego [8]. It makes use of the already available Spoofax implementations of OIL by Denkers [13] 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. 5 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 [15] on a C++ code generator for OIL in Spoofax, an additional intermediate representation is generated before generating mCRL2, called OILSEM. This intermediate representation is close to the formal definition of the semantics of OIL. On this representation we add compatibility checks to the postconditions of transitions. Lastly, we transform the OILSEM representation to mCRL2.

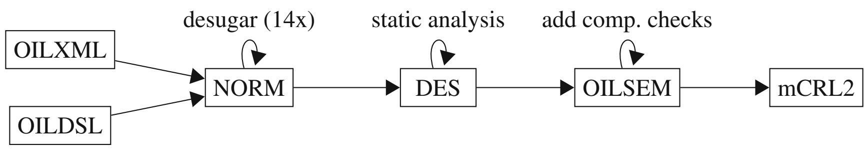

Fig. 5. The transformation pipeline implemented in Spoofax from OIL specification to mCRL2 specification. NORM refers to the normalised AST and DES refers to the desugared AST.

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

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

During the development of the translation to mCRL2 we have relied on the mCRL2 toolset to check for regressions and correctness of the translation. Whenever a new aspect of OIL was added to the translation, an OIL specification illustrating this aspect was translated to mCRL2. Then the corresponding LTS was generated using the mCRL2 toolset to check whether the implementation of the new aspect resulted in expected behaviour. Also, we used equivalence checking to test whether a refactoring in the translation to mCRL2 did not change the behaviour of generated mCRL2 specifications, such as the one that adds auxiliary variables to summands. 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.

3.3 Formal Verification of Validity Requirements

The validity requirements posed in Sect. 2.2 are too complex to be checked on an OIL component specification directly, for instance by means of static analysis. This is mainly because these requirements are about patterns in the global behaviour of the component, which cannot be easily extracted from the structure of the OIL specification. To expose these patterns, we use the translation to mCRL2 and the mCRL2 toolset to generate the underlying LTS. To check the validity requirements on the LTS we define them in terms of the μ-calculus. In this subsection we cover the challenges faced when doing so and provide pattern-like formulae. We use E for the set of all events and E_P for the set of all proactive events in an OIL specification. Wherever possible, we use regular expressions as short-hands.

The first two validity requirements can be easily encoded in the μ-calculus. For safe lookaheadlessness we need to check whether there is any proactive event followed by the `failure` action, which can be encoded by a μ-calculus formula of the form $[E^*][E_P][\texttt{failure}]\mathit{false}$. With finite proactivity we want to enforce finiteness of proactive behaviour, which can be expressed by a least fixpoint operator as follows: $[E^*]\mu X.[E_P]X$. The remaining two Requirements 3 and 4, confluent proactivity and predictable proactivity, are not so easily encoded in the μ-calculus however.

For confluent proactivity we need to be able to check whether states are bisimilar. Since the μ-calculus is based on actions, this is not possible without augmenting the model. To be able to identify bisimilar quiescent states, we first reduce the LTS modulo bisimulation [35] and then add a self-loop to each quiescent state with the action `i(j)`, for some index `j` that is unique for each quiescent state. Then we can check the requirement with a μ-calculus formula of the form:

$$[E^*](\langle E_P\rangle true \Rightarrow \exists_{\mathtt{j:Nat}} : [E_P^*]([E_P]false \Rightarrow \langle \mathtt{i(j)}\rangle true))$$

For predictable proactivity we need to be able check whether sequences consist of the same multi-set of events. Since we use μ-calculus with data, we can build up and store the sequences of proactive events encountered by mapping each proactive event to some value. To check multi-set equality we need to define additional maps and corresponding equations in the data specification of the (generated) mCRL2 specification. Then we can check the requirement with a μ-calculus formula of the form:

$$[E^*](\langle E_P\rangle true \Rightarrow \exists_{w\in E_P^*} : \nu X(w' : E_P^* := \epsilon). \bigwedge_{e\in E_P} [e]X(w'+e) \wedge ([E_P]false \Rightarrow w \approx w'))$$

where ϵ is the empty list and $\approx$ is multi-set equality.

The μ-calculus formulae for the last two requirements quantify over an infinite dataset: the set of natural numbers and the set of all sequences of proactive events are infinite. Therefore, checking these formulae does not terminate without augmenting the LTS further. For each non-quiescent state s, we first follow some sequence of proactive events `w` until we reach some quiescent state with index `j`. Then we add two self-loops to s: one with action `ti(j)` (`t`arget `i`ndex) and one with action `tw(w)` (`t`arget `w`ord). See Fig. 6 for a visualisation of this extension. Note that these transformations do not truly modify the behaviour represented by the model. These actions can then be used in the μ-calculus formulae for Requirements 3 and 4 right after the existential quantifier to give the rewriter a fixed value for the enumeration. It is possible to encode these two requirements in μ-calculus formulae for which such extensions to the LTS are not necessary while guaranteeing termination. However, this exploits knowledge of how the tools currently check these properties, which is undesirable in general.

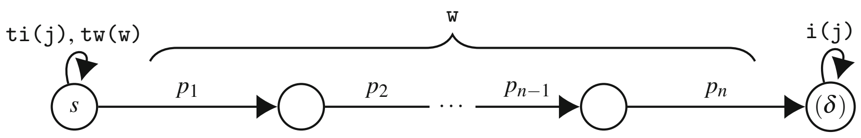

Fig. 6. A visualisation of how the LTS is extended with actions `i`, `ti` and `tw` to help check the mu-calculus formulae for confluent proactivity and predictable proactivity. The actions $p_1, \ldots, p_n$ correspond to proactive events. A (δ) within a state indicates that this state is quiescent.

Transformations have been defined in Spoofax to automatically generate the μ-calculus formulae of all four validity requirements from an OIL specification. These transformations have been defined on OILSEM to reuse the translation to mCRL2 as much as possible. To add the `i(j)`, `ti(j)` and `tw(w)` self-loops, a dedicated graph transformation is applied after the LTS is generated and reduced modulo bisimulation.

4 Experiments

To test the feasibility of our techniques, we have used 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 global state space, we generate the LTS from the generated mCRL2 specification. This LTS is then reduced modulo bisimulation to remove any superfluous behaviour. Afterwards we extend this LTS with additional information as explained in Sect. 3.3 and check the validity requirements on it. See Fig. 3 for the tools used to generate an LTS and to check a property.

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 production environment. With regard to the time needed for the translations, we split the transformation pipeline in two: the translation from OIL specification to analysed desugared AST and from analysed desugared AST to mCRL2 or μ-calculus. This is done because the analysed desugared AST can easily be reused to do multiple translations. For all timings mentioned we have taken the average of at least five runs, except if the time is larger than half an hour, in which case it is only run once.

4.1 The EPC Case

The EPC model is an OIL specification with a total of 10 global state 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 6.9 s. From this analysed model the mCRL2 specification is generated in 3.3 s. The LTS can be generated from the mCRL2 specification in 10 s. 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. The reduced LTS is extended with extra information needed for the last two validity requirements in 0.6 s. See Table 1 for the times needed to generate each requirement and check them on this LTS.

Table 1. The time needed to generate each requirement from the analysed desugared AST and the time needed check the requirement on the reduced LTS for both the EPC and the AGA case. The times shown are in seconds.

	Requirement 1		Requirement 2		Requirement 3		Requirement 4	
	Generate	Check	Generate	Check	Generate	Check	Generate	Check
EPC	0.06	0.7	0.06	0.4	0.1	0.9	57	1.3
AGA	1.9	35	1.8	22	2.1	129	76	125

4.2 The AGA Case

The AGA model is an OIL specification with a total of 55 global state 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 26 s. From this analysed model the mCRL2 specification is generated in 130 s. 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 of global state 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. Note that this effectively abstracts away some event parameters in proactive events, used to send information to the environment. This is no issue, since we are (for now) only concerned with the behaviour of a single component.
- We added assignments to reset global state 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 63 min[2]. The resulting LTS has 113844 states and 177156 transitions. After reduction modulo strong bisim-

[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 6 min from the mCRL2 specification using the options `-bo` for `mcrl22lps` and the option `-rjittyc` for `lps2lts`.

ulation, 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 global state variable has no effect on the behaviour if another global state variable was set to false.

All validity requirements are met on this model. The reduced LTS is extended with extra information needed for the last two validity requirements in 28 s. See Table 1 for the times needed to generate each requirement and check them on this 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 $[E^*]\langle E\rangle true$, 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 $[E^*]\langle E^*.\texttt{start}\rangle true$, 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 fail. Removing these events from E and checking the formula again results in true. These formulae can be checked on the reduced LTS within a few seconds.

5 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. The generation of the fourth validity requirement seems to be an exception to this, but investigation showed that the large generation time is caused by an issue in the pretty printer generated by Spoofax.

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.

Process Structure. We have chosen to describe the semantics of OIL in mCRL2 by using a single monolithic process with a parameter that represents the global state. 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 global state variable can be read or assigned to, the global state would need to be synchronised between all processes frequently. A possible alternative would be to model the global 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 global 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 global state and checking the postconditions, additional communication would be needed.

Automating Preprocessing. As the AGA case clearly shows, the state space of an OIL specification has the potential to explode if it has many global state 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 [33] and live variable analysis [14]. 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`, `lpssumelm` and `lpsstategraph` [28]. 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 global 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.

Enhanced Back-End. The μ-calculus is a good fit for encoding two out of the four validity requirements, namely safe lookaheadlessness and finite proactivity. For confluent proactivity and predictable proactivity however, changes needed to be made to the model. 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 μ-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 [1] or some hybrid logic [22], to encode such properties 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 using other techniques, for instance by encoding these requirements directly in a parameterised Boolean equation system [18] (see Fig. 3), thereby sidestepping the limitations of the μ-calculus, or by building dedicated verifiers.

Another aspect that could be exploited is that specifications such as the AGA model have a number of global state 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 [11]. Some research has already been done regarding model checking software product lines in the context of mCRL2 [4].

6 Conclusion

We have discussed our formalisation of the semantics of OIL, a language for modelling system behaviour. Although OIL was not specifically designed for the efficient application of formal methods, we have been able to define and implement a translation from OIL component specifications to mCRL2 to enable formal verification of such OIL specifications. We have introduced four validity requirements and showed how these can be checked on an OIL specification using mCRL2 and the μ-calculus. Lastly, we have translated two OIL specifications that model systems used in production to mCRL2 and formally verified the four validity requirements on them, thereby positively answering the question whether OIL models can be verified. Scalability could become a concern, as the second case study illustrates. At the same time, the modifications we made to the second case to speed up the analysis show that standard static analyses techniques are likely to help in mitigating such concerns.

Future Work. As software systems typically consist of multiple components, research is needed regarding how to formalise their interaction. We expect this to give rise to more model checking challenges such as checking system wide properties.

OIL can also be used to define protocols, which model the communication behaviour between components. By translating protocol specifications to mCRL2, they can be used to check whether the interaction among components conforms to the protocol. What type of conformance relation would best fit OIL still needs to be investigated.

Due to its mathematical nature, the μ-calculus is not a suitable language for use by software engineers. Since OIL is aimed to be used in an industrial setting, we plan to investigate how engineers can easily define properties without losing precision.

OIL is also used to model existing components (as illustrated by our first case study). To check for regressions between these OIL specifications and the existing implementations, one can use model-based testing; preliminary work in this direction for OIL was carried out by Frenken [15].

Acknowledgements. We thank Canon Production Printing for funding the VOICE-B project, of which this work is part of. We thank Jasper Denkers for his help with understanding Spoofax and its languages and for his remarks on this paper. We thank the reviewers for their helpful comments.

References

1. Axelsson, R., Lange, M., Somla, R.: The complexity of model checking higher-order fixpoint logic. Logical Methods Comput. Sci. **3**(2), 7 (2007)
2. 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: Larsen, K.G., Willemse, T. (eds.) FMICS 2019. LNCS, vol. 11687, pp. 1–21. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27008-7_1
3. ter Beek, M.H., et al.: Adopting formal methods in an industrial setting: the railways case. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 762–772. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_46
4. ter Beek, M.H., de Vink, E.P., Willemse, T.A.C.: Family-based model checking with mCRL2. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 387–405. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_23
5. 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)
6. van Beusekom, R., et al.: Formalising the Dezyne modelling language in mCRL2. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) FMICS/AVoCS -2017. LNCS, vol. 10471, pp. 217–233. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67113-0_14
7. Bouwman, M., Janssen, B., Luttik, B.: Formal modelling and verification of an interlocking using mCRL2. In: Larsen, K.G., Willemse, T. (eds.) FMICS 2019. LNCS, vol. 11687, pp. 22–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27008-7_2
8. 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)
9. Bunte, O., van Gool, L.C.M., Willemse, T.A.C.: Semantics and model checking of OIL component specifications. Technical report, Eindhoven University of Technology (2020)
10. Bunte, O., et al.: The mCRL2 toolset for analysing concurrent systems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 21–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_2

11. Cordy, M., et al.: A decade of featured transition systems. In: ter Beek, M.H., Fantechi, A., Semini, L. (eds.) From Software Engineering to Formal Methods and Tools, and Back. LNCS, vol. 11865, pp. 285–312. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30985-5_18
12. 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)
13. Denkers, J., van Gool, L., Visser, E.: Migrating custom DSL implementations to a language workbench (tool demo). In: SLE, pp. 205–209. ACM (2018)
14. Fernandez, J., Bozga, M., Ghirvu, L.: State space reduction based on live variables analysis. Sci. Comput. Program. **47**(2–3), 203–220 (2003)
15. Frenken, M.: Code generation and model-based testing in context of OIL. Master's thesis, Eindhoven University of Technology (2019)
16. van Gool, L.: Formalising interface specifications. Ph.D. thesis, Eindhoven University of Technology (2006)
17. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
18. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. Theor. Comput. Sci. **343**(3), 332–369 (2005)
19. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M.R., van de Pol, J.: Towards model checking executable UML specifications in mCRL2. Innovations Syst. Softw. Eng. **6**(1–2), 83–90 (2010). https://doi.org/10.1007/s11334-009-0116-1
20. 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)
21. 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)
22. Kernberger, D., Lange, M.: Model checking for hybrid branching-time logics. J. Logic. Algebraic Methods Program. **110**, 100427 (2020)
23. Kim, J.H., Larsen, K.G., Nielsen, B., Mikučionis, M., Olsen, P.: Formal analysis and testing of real-time automotive systems using UPPAAL tools. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 47–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_4
24. Kölbl, M., Leue, S.: Automated functional safety analysis of automated driving systems. In: Howar, F., Barnat, J. (eds.) FMICS 2018. LNCS, vol. 11119, pp. 35–51. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00244-2_3
25. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects Comput. **11**(6), 637–664 (1999). https://doi.org/10.1007/s001659970003
26. Limbrée, C., Cappart, Q., Pecheur, C., Tonetta, S.: Verification of railway interlocking - compositional approach with OCRA. In: Lecomte, T., Pinger, R., Romanovsky, A. (eds.) RSSRail 2016. LNCS, vol. 9707, pp. 134–149. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33951-1_10
27. Mitsch, S., Gario, M., Budnik, C.J., Golm, M., Platzer, A.: Formal verification of train control with air pressure brakes. In: Fantechi, A., Lecomte, T., Romanovsky, A. (eds.) Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification. RSSRail 2017. Lecture Notes in Computer Science, vol. 10598, pp. 173–191. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68499-4_12

28. van de Pol, J., Timmer, M.: State space reduction of linear processes using control flow reconstruction. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 54–68. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04761-9_5
29. Remenska, D., et al.: From UML to process algebra and back: an automated approach to model-checking software design artifacts of concurrent systems. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 244–260. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_17
30. 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)
31. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electron. Notes Theor. Comput. Sci. **55**(3), 357–369 (2001)
32. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 62–77. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_5
33. Silva, J.: A vocabulary of program slicing-based techniques. ACM Comput. Surv. **44**(3), 12:1–12:41 (2012)
34. 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: Howar, F., Barnat, J. (eds.) FMICS 2018. LNCS, vol. 11119, pp. 67–82. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00244-2_5
35. Valmari, A.: Bisimilarity minimization in $O(m \log n)$ time. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 123–142. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02424-5_9
36. Visser, E., et al.: A language designer's workbench: a one-stop-shop for implementation and verification of language designs. In: Onward!, pp. 95–111. ACM (2014)
37. Zhang, S.J., Liu, Y.: An automatic approach to model checking UML state machines. In: SSIRI (Companion), pp. 1–6. IEEE Computer Society (2010)