A Formalisation of SysML State Machines in mCRL2

Mark Bouwman[1(✉)], Bas Luttik[1], and Djurre van der Wal[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
m.s.bouwman@tue.nl

[2] University of Twente, Enschede, The Netherlands

Abstract. This paper reports on a formalisation of the semi-formal modelling language SysML in the formal language mCRL2, in order to unlock formal verification and model-based testing using the mCRL2 toolset for SysML models. The formalisation focuses on a fragment of SysML used in the railway standardisation project EULYNX. It comprises the semantics of state machines, communication between objects via ports, and an action language called ASAL. It turns out that the generic execution model of SysML state machines can be elegantly specified using the rich data and process languages of mCRL2. This is a big step towards an automated translation as the generic model can be configured with a formal description of a specific set of state machines in a straightforward manner.

1 Introduction

The importance of correct specifications is evident for safety-critical systems such as those in the railway domain. At the same time, due to the increasing use of digital technology in those systems, specifications are getting more and more complex and harder to get completely correct. To cope with the complexity, railway engineers are gradually adopting a model-based system engineering approach for the development of their systems. EULYNX[1], an initiative of a consortium of thirteen European railway infrastructure managers, uses SysML to specify a standard for interfaces between the various components of a signalling system (signal, point, level crossing, interlocking, etc.).

The use of SysML for system requirements specification is a big step forward for the railway domain as it is significantly more precise than natural language. SysML has a fairly intuitive graphical syntax, which allows railway engineers to understand and use it without extensive training. Still, SysML is *semi-formal*: it has a well-defined syntax, but its semantics is informal and not firmly grounded in mathematics. As a consequence, system behaviour specified by a SysML model is not directly amenable to the more thorough kind of analysis that genuine *formal* methods offer.

[1] See https://eulynx.eu.

K. Peters and T. A. C. Willemse (Eds.): FORTE 2021, LNCS 12719, pp. 42–59, 2021.
https://doi.org/10.1007/978-3-030-78089-0_3

The aim of the *FormaSig*[2] project, a collaboration of the Dutch and German railway infrastructure managers, Eindhoven University of Technology and the University of Twente, is to formalise the aforementioned EULYNX standard to the extent that delivered components conforming to the standard provably satisfy a collection of safety properties. The idea is to associate with each EULYNX SysML model a formal mCRL2 model [5,6]. Then mCRL2's model checker can be used to establish that the model satisfies the required safety properties, and automated model-based test technology can be used to reliably test compliance to the model of actual implementations (see Fig. 1). In a first case study, we have demonstrated the viability of this idea. We took the EULYNX SysML model of the Point interface, associated an mCRL2 model with it, used the mCRL2 model checker to analyse its correctness and used the model-based test tool JTorX [2] to check conformance of a SysML simulator of Point [4].

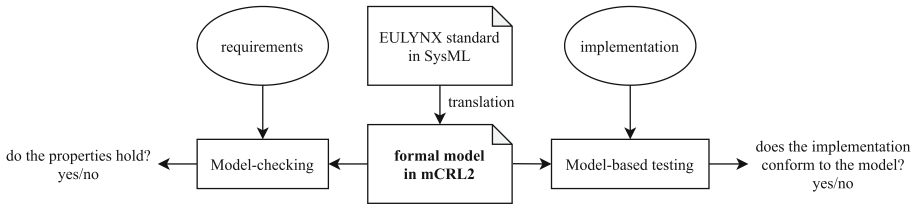


Fig. 1. FormaSig: using a formal mCRL2 model to establish that implementations conforming to the EULYNX standard satisfy properties.

The EULYNX standard is under development, and it is likely that also in the future it will be subject to changes. Hence, it is impractical to rely on manual translations from the EULYNX SysML models to mCRL2. To facilitate that model-checking and model-based test techniques will become an integral aspect of maintaining the standard, it is imperative that the translation from EULYNX SysML to mCRL2 is automated. Another benefit of having a automated translation is that, as its correctness can be rigorously examined, the likelihood of introducing mistakes in the formalised model is reduced.

How EULYNX SysML models are meant to be interpreted is specified in the EULYNX modelling standard. So far FormaSig delivers *an* interpretation of EULYNX SysML. Going forward, FormaSig also aims to increase precision of the modelling standard and become *the* official interpretation.

Implementing an automated translation from SysML to mCRL2 is, however, a nontrivial undertaking, most notably hampered by the lack of a complete and comprehensive formal semantics for SysML and the complexity of the informally described SysML execution model. Furthermore, also due to the lack of a fixed formal semantics, there are many dialects of SysML. A particular variation point is the action language, the language used to specify guards and the effects of

[2] Formal Methods in Railway Signaling Infrastructure Standardization Processes.

transitions. In EULYNX SysML all communication is performed via ports, which are referenced as variables in the action language. The action language itself is ASAL, which is tied to the PTC Windchill tool[3].

The main contribution of this paper is to present a formalisation of the informal semantics of EULYNX SysML state machines directly in mCRL2. Our formalisation consists of three parts. The first part is a generic, comprehensive formalisation of the operational semantics of UML state machines, which form the basis of EULYNX SysML state machines. This part involves formalising the notion of state hierarchy and transition selection. The second part adds an interpretation of the particular communication mechanism via ports that is used in EULYNX SysML. The third part defines an execution model for the ASAL action language. In this paper, we generalise to a class of action languages that reference ports as variables. The resulting mCRL2 specification can straightforwardly be turned into an actual formal model interpreting a particular EULYNX SysML interface by populating the relevant data types with some static details from the SysML model and generating a suitable number of instantiations of predefined processes. For the latter, we have implemented a tool that is discussed in a companion paper [22]. The resulting mCRL2 specification can be model-checked and used for model-based testing and serve as the formal model central to FormaSig idea (see Fig. 1).

The semantics of UML/SysML state machines has been formalised in preceding academic work. A number of papers describe a translation from UML state machines to PROMELA (the input language of the SPIN model checker) [11–13,18,21]. Our formalisation of transition selection draws inspiration from [13]. In [14] a structural operational semantics is presented along with a custom verification tool USM^{2}C. The AVATAR [19] tool offers a SysML-style environment with particular focus on verifying security properties; it offers translation to UPPAAL and ProVerif. Other translations and formalisations include a translation from xUML class diagrams and state machines to mCRL2 [7,8], a translation from SysML BDDs and state machines to NuSMV [23], a formalisation of UML state machines using structured graph transformations [10] and a formalisation of UML state machines in Object-Z [9]. In [20] a translation is given from sequence diagrams to mCRL2. Our approach to formalisation differs from earlier work by specifying the generic semantics in the target formal language which can be instantiated with a specific configuration. Moreover, our formalisation includes a communication mechanism using ports.

The OMG organisation, which manages the UML and SysML standards, has also released "Precise Semantics of UML State Machines (PSSM)" [17], which gives an informal but very precise semantics. Our formalisation differs in at least one way from PSSM. We do not create completion events in order to prevent cluttering the event queue. Instead, transitions relying on a completion event have completion of the source state as an extra guard. PSSM also provides an extensive compliance test suite. In the future we would like to make a version

[3] https://www.ptc.com/en/products/windchill/integrity/.

of our model that adheres to PSSM and measure the effect of completion events on the state space.

This rest of the paper is organised as follows. In Sect. 2 we present the visual syntax of state machines and a summary of the run-to-completion semantics; this section can be skipped by readers familiar with state machines. In Sect. 3 we give a quick introduction to mCRL2. From Sect. 4 to Sect. 6 we go into the details of the formalisation. We conclude the paper in Sect. 7.

2 An Informal Introduction to UML State Machines

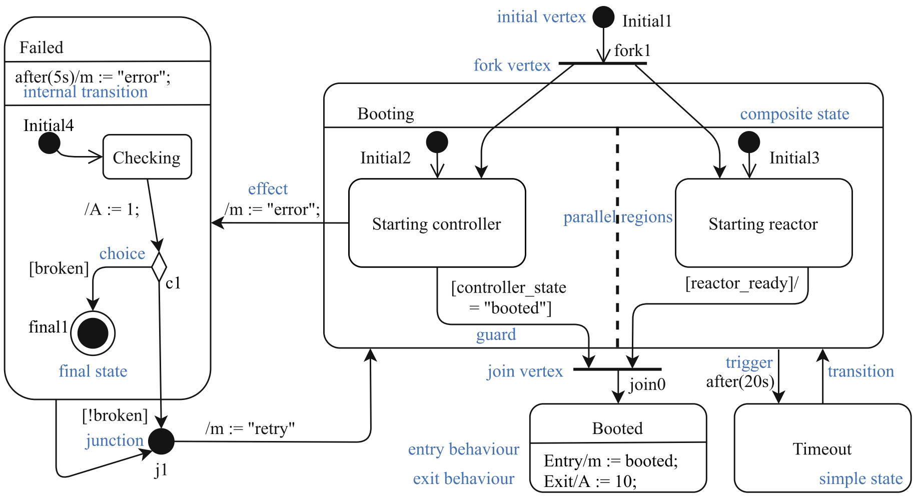


Fig. 2. Example showing all state machine constructs supported in EULYNX SysML.

Figure 2 shows an example of a state machine, with names of the various constructions added in blue. In this section, we briefly discuss the informal semantics of each construction as in the UML standard [16].

The basic constituents of state machines are *states* and *transitions*. Initial states, choice states, final states, junctions, forks and joins are also called *pseudostates*. The UML state machine formalism derives its expressiveness from these the possibility to have states and transitions nested within states, and even have transitions cross state border. Transitions may have a *trigger*, a *guard* and an *effect*. The trigger of a transition (which is optional) is an *event*; it can be a change event (notation when(x)) or a timeout event (notation after(x)).

The modeller can define behaviour that is executed upon entering or exiting a simple or composite state. *Exit behaviour* is executed before the effect of a transition, *entry behaviour* is executed after the effect of a transition. Simple and composite states can also have *internal transitions*, which do not change state (see, e.g., the state Failed in Fig. 2).

Junctions and *choice* vertices allow more concise specification of transitions that induce the same behaviour. The choice vertex c1 in Fig. 2 combines two transitions from Checking which share the common behaviour A:=1. Junctions serve a similar purpose (see junction j1 in Fig. 2) with the difference that for junctions the guards of outgoing transitions need to be checked before taking a transition to the junction, whereas for choice vertices the guards are checked when arriving at the vertex.

A state can contain other states, in which case it is called a *composite state* and the states it encloses are called *substates*. A composite state can have a *final state* (see, e.g., the state Failed in Fig. 2). Transitions from the border of a composite state can be fired regardless of the current substate, except when the transition does not have a trigger, in which case the current substate of the composite state must be a final state. A composite state may also have multiple *parallel regions*. Each region has an *initial state* and can perform local transitions independently of other regions. A transition ending at the border of a composite state with parallel regions will let each region start from its initial state. A fork indicates that a transition ends on specific states in multiple regions. Conversely, a join can begin from specific states in parallel regions.

Due to the presence of composite states, a state machine is not just in a single state but in a collection of states, a *state configuration*. A state configuration is *stable* when it does not contain pseudostates. Transitions are specified on states; a state machine may combine several transitions (as is the case with joins, forks and junctions) to perform a bigger step from one state configuration to another, which we will call a *step*. Events that occur are stored in an *event pool* until they are dispatched. A step is enabled when the specified trigger (if any) is in the event pool and all guards of transitions involved in the step evaluate to true. State machines have *run-to-completion* semantics: a state machine selects a step to execute and will completely finish executing the behaviour of the step and entry and exit behaviour before it considers performing a new step. Parallel regions may start a step simultaneously when both steps have the same trigger; in that case the state machine performs a *multi-step*.

3 Introduction to mCRL2

The mCRL2 toolset is designed to model and analyse concurrent and distributed systems. The mCRL2 language is an ACP-style process algebra and contains a rich data language based on abstract data types. The semantic interpretation of an mCRL2 model is a Labelled Transition System (LTS). By translating from SysML to mCRL2 we indirectly associate an LTS to the SysML model. The toolset contains tools for the verification of parametrised modal μ-calculus formulas, bisimulation reduction, counterexample generation, simulation and visualisation. To aid the reader in understanding the mCRL2 snippets in following sections, we will cover some basics using an example unrelated to the contents of this paper. For more information on mCRL2 we refer to https://mcrl2.org and [6].

The mCRL2 language has some primitive data types built in, such as integers, natural numbers and booleans, including common operations on them. Users can also define their own data types and operations. The code below shows an example. The *sort* `Place` has one *constructor*, `Coordinates`, with *projection functions* `X` and `Y`. Equations are treated as rewrite rules; terms that match the left hand side will be rewritten to the right hand side.

```
sort Place = struct Coordinates(X:Nat, Y: Nat);
map computeManhattanDistance: Place#Place -> Nat;
var p1, p2:Place;
eqn computeManhattanDistance(p1,p2) = abs(X(p1)-X(p2))+abs(Y(p1)-Y(p2));
```

The process definition below specifies the behaviour of the `Point` process; it can perform three actions: `move`, `invite` and `respond`. The sum operator represents a non-deterministic choice over all values of the quantified data domain. Summations over infinite data domains can be restricted by adding a condition. In the example below a condition is used to restrict a point process to `move` to any place on a 10 by 10 grid.

```
act move: Nat; invite, respond, meet: Place;
proc Point(p:Place) = sum new:Place. (X(new) < 10 && Y(new) < 10)
    -> move(computeManhattanDistance(p,new)).Point(new)
  + invite(p).Point(p) + sum new:Place. respond(new).Point(new);
```

The initial process expression specifies the initial state of the labelled transition associated with the specification. The example below specifies a parallel composition of two `Point` processes wrapped in a communication and an allow operator. Both `Point` processes can perform a `move` *action*, which is allowed by the allow operator. The `invite` and `respond` actions are not allowed and hence blocked. However, the two processes can synchronize on a *multi-action* `invite|respond`, which is transformed to a `meet` action by the communication operator, which is allowed by the allow operator. The labelled transition system (sometimes referred to as the state space) associated with this specification will have exactly 10.000 states, representing every combination of coordinates.

```
init allow({move,meet}, comm({invite|respond -> meet},
  Point(Coordinates(1,1))||Point(Coordinates(2,3))));
```

4 The Operational Semantics of State Machines

In Sect. 2 we already gave a rough sketch of the execution semantics of state machines. In this section we treat the semantics of UML state machines and its formalisation in mCRL2, including the role of the action language and some mCRL2 snippets that are illustrative of the formalisation. The model itself is available on GitHub [3]. In Sect. 5 we extend the UML semantics with EULYNX SysML specific communication over ports. In Sect. 6 we detail how to complete the model with a configuration and touch on the subject of verification.

4.1 Strategy to Formalisation

Our goal is to generically describe the semantics of state machines in mCRL2 achieving a high degree of modularity. There are several choices to be made (e.g., with respect to the granularity of interleaving, run-to-completion semantics, syntax and semantics of the action language) and we want to set up our specification in such a way that parts of it can be easily modified or replaced. A particular concern is that the specific details of a concrete state machine to be translated are separated from the generic semantics.

Due to our modular setup it is rather straightforward to configure the generic model with a specific set of communicating state machines. The user needs to do two things: 1) define the semantics of the action language and 2) encode the structure of the state machines in an mCRL2 data type and pass it as a parameter to the generic state machine process.

4.2 Abstract Action Language

The UML standard [16] is not prescriptive of the action language used to specify guards and the effect of transitions. In this paper we abstract from any particular action language.

Let `Instruction` be a sort containing all action language expressions, which we will also refer to as a *behaviour*. Let `VarName` be a sort containing all variable names. It is assumed that variables range over elements of a sort `Value`.

In order to formalise the action language semantics it may be necessary to include additional data structures, e.g., a program stack or a valuation of local variables. To encapsulate such additional data structures we introduce the notion of *execution frame*, represented by the mCRL2 sort `ExcFrame`, which will be assumed to consist of all data necessary to execute programs of the action language. We do not assume that execution of behaviour is atomic, we allow that two components interleave their execution of behaviour when they are both taking a transition. We abstract from the granularity of interleaving and simply allow an execution frame e to make a step to an execution frame e', where e' may still have behaviour waiting to be executed.

To define the semantics of the action language the user needs to add equations for the following mappings. We assume a subset of action language expressions represent predicates, which can be evaluated using `checkPredicate`.

```
sort VarValuePair = struct VarValuePair(getVariable:VarName, getValue:Value);
  Instructions = List(Instruction);
map initializeExcFrame: Instructions#(VarName -> Value) -> ExcFrame;
  executeExcFrameCode: ExcFrame -> ExcFrame;
  checkPredicate: Instructions#(VarName -> Value) -> Bool;
  isFinished: ExcFrame -> Bool;
  getValuation: ExcFrame -> VarName -> Value;
  getVariableUpdates: ExcFrame -> List(VarValuePair);
  resetVariableUpdates: ExcFrame -> ExcFrame;
```

The mapping `getVariableUpdates` is assumed to retrieve all updates to variables that occurred during the execution of the execution frame. This field is needed for deriving change events, described in Sect. 4.5.

4.3 Representing State Machines in mCRL2

We assume that `StateName` and `CompName` have been declared as mCRL2 enumeration sorts, enumerating, respectively, all state names and all state machine identifiers occurring in the SysML model under consideration. These two sorts are part of the configuration in the model as they need to be instantiated.

We proceed by introducing the sort `StateInfo`, which is an example of mCRL2's facility to define structured sorts. By means of a structured sort, data can be concisely aggregated. An element of the sort `StateInfo` is either a triple with *constructor* `SimpleState` or with constructor `CompositeState`, both with *projection functions* `parent`, `entryAction` and `exitAction`, or it stores a single data element together with a constructor (`JoinVertex`, `JunctionVertex`, etc.).

```
StateInfo = struct SimpleState(parent: StateName, entryAction: Instructions,
    exitAction: Instructions) | CompositeState(parent: StateName,
    entryAction: Instructions, exitAction: Instructions)
  | JoinVertex(parent: StateName) | JunctionVertex(parent: StateName)
  | ForkVertex(parent: StateName) | InitialState(parent: StateName)
  | FinalState(parent: StateName) | ChoiceVertex(parent: StateName);
```

The parent of a state is stored to represent the hierarchy of states induced by composite states. A state's parent is the first enclosing composite state. We assume that the sort `StateName` has a special element `root`; states that are not enclosed in a composite state have `root` as their parent.

Our framework supports change events and timeout events, see the definition of the sort `Event` below. The event type `none` is used as placeholder for transitions without a trigger. Time is currently not modelled explicitly in our framework, even though mCRL2 does support it. Explicit timing would result in a significantly larger state space, while it is not relevant for the properties that need to be verified in the context of EULYNX. Instead, transitions with a timeout event as trigger can fire non-deterministically. The generation of change events is discussed in Sect. 4.5.

```
Event = struct none | ChangeEvent(getTriggerExpr:Instructions) | TimeoutEvent
```

The sort `Transition` (given below) is used to specify the transitions of a state machine. The Boolean `internal` is used to differentiate between selfloops and internal transitions, the latter do not induce entry and/or exit behaviour.

```
Transition = struct Transition(source:StateName, trigger:Event,
  guard:Instructions, effect:Instructions, target:StateName, internal:Bool);
```

We also define the sort `StateMachine`, which aggregates all the information we need of a state machine.

```
StateMachine = struct StateMachine(
  transitions:List(Transition),initialState:StateName,states:List(StateName),
  stateInfo: StateName -> StateInfo, initialValuation: VarName -> Value);
```

The `initialState` designates the initial state in the root of the state machine (i.e. the initial state that is not contained in a composite state). The projection functions `states` and `stateInfo` retrieve which states are present in the state machine and the associated `StateInfo`, respectively. Note that functions can be partial in mCRL2, the function `stateInfo` only needs to be defined for the state names that occur in that state machine.

Due to the hierarchy of states a state machine is 'in' a collection of states, a state configuration. A state configuration can be represented as a tree structure where the top node is not enclosed in a composite state. Parallel regions introduce nodes with multiple children. The mCRL2 excerpt below gives the definition of state configurations in the model.

```
StateConfig =
  struct StateConfig(rootState:StateName,substates:List(StateConfig));
```

An example configuration of the state machine depicted in Fig. 2 is

```
StateConfig(Booting,[StateConfig(Initial2,[]),StateConfig(Initial3,[])]).
```

4.4 Step Selection and Execution

As explained in Sect. 2, state machines make a step from one state configuration to another. Such a step could consist of multiple transitions, as is the case with junctions, joins and forks. We could in theory perform step selection by performing a reachability search across the transitions. We anticipate that this will make step selection computationally expensive. Instead, we opt to preprocess `Transitions` into `Steps`. The definition of `Step` is given below, as well as the mapping that derives `Steps` from `Transitions`. The `effect` of the step is a `ComposedBehaviour`. It allows us to create a partially ordered set of behaviours, which is needed for defining steps in the context of parallel regions (see Fig. 3).

```
sort Step = struct Step(source: StateConfig, trigger: Event,
    guard: List(Instructions),effect: ComposedBehaviour, target: StateConfig,
    internal: Bool, arrowEnd: StateName);
  ComposedBehaviour = List(InstructionOrPar);
  InstructionOrPar = struct Instruction(getInstruction:Instruction)
    | ParBehaviours(parBehaviours:List(ComposedBehaviour));
map transitionsToSteps: StateMachine -> List(Step);
```

The mCRL2 code specifying the transformation from `Transitions` to `Steps` consists of over 200 lines. Avoiding too much detail we illustrate what transformations are done. The first transformation is to create a `Step` object for every `Transition` by adding the ancestors to the source and target state.

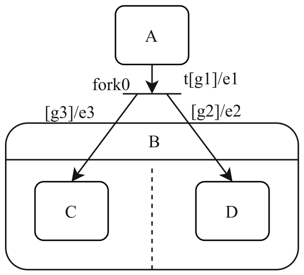
We deal with forks by combining the outgoing transitions. The transition to the fork is changed by adding the guards of the outgoing transitions. The effects of the outgoing transitions are put in parallel (See Fig. 3). Note that in the case of a fork the step does not have a single `arrowEnd`; we assume that `StateName` has a special element `multiple` which will be used in the case of forks.

Similarly, we deal with joins by combining incoming steps and their guards. For steps ending on a composite state we add the initial state in the target. For steps from composite states there are two options: if the step has a change event as trigger then we do not add a substate to the source (step is enabled regardless of the substate); if the step does not have a trigger we require that all the parallel regions of the composite state are in a final state.

We remove junctions by introducing a step for each path over the junction.

Given a state configuration and a set of steps we can reason about which steps are enabled for firing. We will go over the restrictions for firing steps that are checked in different data equations.

The most basic requirement for selecting a step is that the source of the step must match the current state configuration. This is checked by



```
Step(
  StateConfig(A,[]),
  t,
  [g1,g2,g3],
  [e1],
  StateConfig(fork0,[]),
  false,
  fork0)
```

```
Step(
  StateConfig(fork0,[]),
  none,
  [],
  [ParBehaviours(e2,e3)],
  StateConfig(B,
    [StateConfig(C,[]),
    StateConfig(D,[])]),
  false,
  multiple)
```

Fig. 3. Example steps to and from a fork.

`filterPossible`, defined below. The helper function `getAllStatesConfig` returns the set of all states that are in a state configuration. The helper function `containsPseudoState` checks whether a state configuration contains a pseudostate. Due to the run-to-completion semantics we only select a new step when we have reached a stable state configuration (i.e. a state configuration without pseudostates). For this reason we add the condition that if the current state configuration contains a pseudostate then we will only consider transitions from the pseudostate.

```
map filterPossible: List(Step)#StateConfig#StateMachine -> List(Step);
  matchState:StateConfig#StateConfig -> Bool;
var sc, sc1, sc2: StateConfig; step: Step; steps: List(Step);
  sm:StateMachine;
eqn filterPossible([], sc, sm) = [];
  filterPossible(step |> steps, sc, sm) = filterPossible(steps,sc,sm)
    ++ if(matchState(sc,source(step))
        && (containsPseudoState(sc,sm)=>containsPseudoState(source(step),sm)),
     [step], []);
  matchState(sc1,sc2) = (getAllStatesConfig(sc2)-getAllStatesConfig(sc1))=={};
```

Another requirement is that the guard evaluates to true and the trigger matches the current event that is being processed. These two checks are performed by `filterEnabled`.

```
filterEnabled: List(Step)#Event -> List(Step);
```

Another rule is that steps for which the source is lower (i.e. more deeply nested) in the state hierarchy have a higher priority than steps for which the source is higher in the state hierarchy. The mapping `filterPriority` selects the steps with the highest priority among the input. Note that there may be multiple steps on the same priority level.

```
filterPriority: List(Step) -> List(Step);
```

As mentioned earlier, a state machine can also perform a multi-step if multiple steps with the same trigger event are enabled in parallel regions. To be more precise: the state machine selects a multi-step consisting of the maximal set of non-conflicting steps, where non-conflicting means that no two steps in the set exit the same state. The mapping `multiStepPossibilities` computes all such multi-steps given a set of steps.

```
multiStepPossibilities: List(Step) -> List(List(Step));
```

Due to the way we have constructed `Steps` the target field of a transition is not always a complete state configuration. We leave out parallel regions in defining transitions when they do not actively contribute. If we were to include all the parallel regions in the `source` and `target` of `Steps` we would have to compute all variations. To construct the new state configuration `computeNextState` takes the target of a transition and adds the parallel regions of the current state configuration that are unaffected (i.e. not exited).

```
computeNextState: StateConfig#Step -> StateConfig;
```

`computeNextState` recurses through the tree structure of the target state configuration. At each level it copies over unaffected regions. It is unaffected when the region was not present in the source of the step (it was not an active participant of the step) and it is not exited by the step.

The behaviour of performing a step, i.e. an instance of `Instructions`, is the behaviour of the step itself combined with possible exit and entry behaviour. For internal transitions no state is entered or exited. The snippet below shows the definition of `determineBehaviourStep`.

```
map getEntryBehaviour: StateMachine#StateConfig#Step -> ComposedBehaviour;
  getExitBehaviour: StateMachine#StateConfig#Step -> ComposedBehaviour;
  determineBehaviourStep: StateMachine#Step#StateConfig -> ComposedBehaviour;
var sm: StateMachine; cur: StateConfig; st: Step;
eqn (!internal(st)) -> determineBehaviourStep(sm,st,cur) =
    getExitBehaviour(sm,cur,st) ++ effect(st) ++ getEntryBehaviour(sm,cur,st);
  internal(st) -> determineBehaviourStep(sm,st,cur) = effect(st);
```

Both `getEntryBehaviour` and `getExitBehaviour` compute the new state configuration after firing the transition and which states are entered/exited; subsequently they determine the order in which behaviour needs to be executed and construct a `ComposedBehaviour`. The order of composing entry behaviours is outside-in (top level states first) and the order of composing exit behaviours is inside-out (nested states first). To determine the order both functions recurse through the new state configuration. Behaviour of states that are entered/exited that are on the same level (parallel regions) is put in parallel.

We use a mapping `computeExecutionOptions` to compute all the options for what behaviour from a `ComposedBehaviour` can be executed next. If the head of the composed behaviour is a sequential composition of instructions it will return one execution option with all instructions up to the end of the composed behaviour or up to a parallel composition (whatever comes first). If the head of the composed behaviour is a parallel composition then we get multiple options corresponding to each parallel branch.

```
sort ExecutionOption = struct ExecutionOption(getCodeToExecute:Instructions,
     getRemainingBehavior:ComposedBehavior);
map computeExecutionOptions: ComposedBehavior -> List(ExecutionOption);
```

4.5 Change Events

A change event is generated when the content of a `when(`x`)` trigger *becomes* true. When a variable is updated we need to check which change events need to be generated. For this purpose we introduce the sort `Monitor`. A monitor stores an action language expression and the last evaluation. When we update a variable we can check which change events are generated using `deriveChangeEvents`. The mapping `updateMonitors` updates the valuation stored in the monitors.

```
sort Monitor = struct Monitor(getExpression:Instructions, getValuation:Bool);
map deriveChangeEvents: List(Monitor)#(VarName -> Value) -> List(Event);
  updateMonitors: List(Monitor)#(VarName -> Value) -> List(Monitor);
var vars: VarName -> Value; mon: Monitor; mons: List(Monitor);
eqn deriveChangeEvents(mon |> mons,vars) =
  if(checkPredicate(getExpression(mon),vars) && !getValuation(mon),
    [ChangeEvent(getExpression(mon))], []) ++ deriveChangeEvents(mons,vars);
  deriveChangeEvents([],vars) = [];
```

4.6 StateMachine Process

The state machine process uses the data operations that we described in earlier sections and uses them to specify the observable *actions* of a single state machine, which will be visible in the LTS associated to the mCRL2 model. For now we will present a slightly simplified version, which we will extend when we incorporate SysML specific communication in Sect. 5. Below we present the parameters of the process and the declaration of the observable actions (which includes the parameters of those actions).

```
act discardEvent: Event; selectMultiStep: Event#List(Step);
  executeStep: Step; executeBehaviour;
proc StateMachine(ID:CompName, SM:StateMachine, sc:StateConfig,
    eq:List(Event), steps:Set(Transition), behav:ComposedBehaviour,
    mon:List(Monitor), vars:VarName -> Value, exc:ExcFrame) = ...
```

The UML standard does not define in what order events are processed. We have opted to process events in FIFO order, hence `event_queue` is a list of events. The `StateMachine` process consists of one big alternative composition where each summand performs one action and then recurses (with updated parameters).

The observable actions are chosen to reflect decisions in the run-to-completion cycle. When both `steps` and `behav` are empty a new multi-step should be considered. If no step is enabled by the head of the event queue the process can perform a `discardEvent` action and remove it from the event queue. Alternatively, we can select a multi-step with a `selectMultiStep` action. We can now perform a `executeStep` action to start executing one of the selected steps, which updates `sc` and puts the composed behaviour of the step in `behav`. The process selects one of the execution options calculated by `computeExecutionOptions` and initializes an `ExcFrame` which is stored in `exc`. The process calls `executeExcFrameCode` and performs an `executeBehaviour` action until the execution frame is finished. Every time code is executed (and thus possibly variables are updated), it is checked whether change events can be derived. When the execution frame is finished we compute a new execution option. When the execution of `behav` is finished we select a next step from `steps`. When there are no more steps to execute the process is ready to select a new multi-step.

As an example, consider the summand that performs the `executeStep`. Note that mCRL2 allows for an abbreviated, assignment-like syntax in which only the to be updated parameters need to be mentioned in a recursive call; all other parameters of the process remain the same.

```
+ (#behavior_to_execute == 0) ->
  sum next_step:Step. (next_step in steps) -> executeStep(next_step)
    .StateMachine(steps = steps - {next_step},
      behav = determineBehaviourStep(SM,next_step,sc),
      sc = computeNextState(sc,next_step))
```

Depending on the kind of analysis that will be performed on the resulting LTS we might want different observable actions. If we would want to verify something regarding the state configuration we might want to add a self loop signalling the current state configuration. Alternatively, we might want to hide some actions by renaming them to τ, indicating that they are unobservable.

5 SysML Specific Communication

Specific to EULYNX SysML is that there are ports over which communication takes place. Internal Block Diagrams (IBDs) describe the interfaces of components by specifying the ports of components and their connections.

This paper focuses on the semantics of a set of communicating state machines. For the semantics of IBDs we refer the reader to [22]. Here we assume that we have the following communication structure. Each component has a set of ports, which are subdivided in input and output ports. An output port can be connected to multiple input ports. Both input and output ports need not be connected at all, in which case they interact with the environment. One more assumption on the action language is that ports are treated as variables: changing the variable associated to an output port leads to a communication, which updates the variable associated to the input port of the receiver.

The sort `Component` extends state machines with extra information. The sort `Channel` models the connections between ports. Both sorts are defined below. The sort `CompName` defines a finite enumeration of identifiers for components.

```
CompPortPair = struct CompPortPair(getComp: CompName, getPort: VarName);
Component = struct Component(SM: StateMachine, in_ports: List(VarName),
  out_ports:List(VarName));
Channel = struct Channel(sender: CompPortPair, receivers: List(CompPortPair));
```

To take into account communication between state machines, we modify the `StateMachine` process of Sect. 4.6 by replacing the state machine parameter with a `comp` parameter, adding a parameter `oq` and adding two extra actions:

```
act sendComp,receiveComp: CompPortPair#Value;
proc StateMachine(...,comp:Component, oq:List(VarValuePair)) = ...
```

When executing an execution frame we check whether there are updates to output ports and store those updates in output queue `oq`. When `oq` is not empty it can perform a `sendComp` action, communicating the update. At any point in time the process can receive messages via a `receiveComp` action. The summand related to receiving messages is given below.

```
sum v:Value,p:VarName. receiveComp(CompPortPair(ID,p),v)
  .StateMachine(vars = vars[p -> v],
    eq = eq ++ deriveChangeEvents(mon, vars[p->v]),
    mon = updateMonitors(mon,vars[p -> v]))
```

We want to ensure that when a value is sent on an output port, it is received by all (and only) connected input ports. This is enforced by the `Messaging` process and the allow and communication operators in the initialization process (both given below). When the number of components in a configuration is n then the allow operator and `Messaging` process should be extended with the ability to perform a `send` with up to $n-1$ `receive` actions.

```
proc Messaging(channels: List(Channel)) =
  sum ch:Channel, v:Value. (ch in channels) ->
  ((#receivers(ch) == 1) -> receiveI(sender(ch),v)|sendI(receivers(ch).0,v)
    + (#receivers(ch) == 2) -> receiveI(sender(ch),v)|sendI(receivers(ch).0,v)
      |sendI(receivers(ch).1,v)
  ).Messaging();
init allow({selectMultiStep, discardEvent, executeStep, executeBehaviour,
    send|receive, send|receive|receive},
    comm({sendComp|receiveI -> send, sendI|receiveComp -> receive},
        MessagingIntermediary||Environment
        ||StateMachine(...)||StateMachine(...) ...));
```

The central idea is that individual components need not know how ports are connected. Instead, the `Messaging` provides a 'meeting place' with which the sender and receivers synchronize. As an example, suppose some component $C1$ sends some value v on port $P1$ that should be received by two receivers $C2$ and $C3$ on ports $P2$ and $P3$, respectively. The `Messaging` process and the `StateMachine` process of the sender and the two receivers can perform the *multi-action*

```
sendComp(C1,P1,v)|receiveI(C1,P1,v)|sendI(C2,P2,v)
|sendI(C3,P3,v)|receiveComp(C2,P2,v)|receiveComp(C3,P3,v).
```

This is transformed by the communication operator to `send(C1,Port1,v) |receive(C2,Port2,v)|receive(C3,Port3,v)`.

Ports that are not connected to any other port are exposed to the environment, i.e. adjacent systems not included in the model. Input ports exposed to the environment can expect inputs at any moment in time. We model the environment with the `Environment` process, which can always send messages to ports in `envInputs` and receive messages from ports in `envOutputs`. Note that a connection between the environment and an exposed port must also be passed to the `Messaging` process.

```
Environment(envInputs:List(CompPortPair), envOutputs:List(CompPortPair)) =
  sum inp:CompPortPair, v:Value. (inp in envInputs)
    -> sendComp(CompPortPair(Environment,getPort(inp)),v).Environment()
  + sum out:CompPortPair, v:Value. (out in envOutputs)
    -> receiveComp(CompPortPair(Environment,getPort(out)),v).Environment();
```

6 Creating a Configuration and Model Checking

In the previous sections, we have discussed the generic parts of the mCRL2 model; in this section, we describe how to configure the model with a specific configuration and touch on the subject of model checking.

First, the enumerations `StateName`, `CompName` and `VarName` need to be instantiated. The action language needs to be defined: the sorts `Value` and `Instruction` need to be defined. Also the semantics of the action language need to be defined by extending the sort `ExcFrame` and giving defining equations for the mappings listed in Sect. 4.2. Finally, the initial process expression needs to be given, in accordance with the structure described in Sect. 5. The `Environment` and `Messaging` processes must be given appropriate parameters. For every state machine a process expression `StateMachineInit(`c`,`x`)` needs to be added, where c is a `CompName` and x is a `Component` object.

The model available on GitHub [3] contains an example configuration. This configuration contains just one component named `C1` with the state machine of Fig. 2. The initial valuation of `C1` gives `controller_state` the string "booted" as initial value and sets `reactor_ready` to true. There is one channel: component `C1` has an output port `m`, which is open to the environment.

We can use the mCRL2 model checker to verify properties expressed in the expressive parametrised first-order modal μ-calculus. For instance, we can verify that we always eventually reach the state Booted. We need to capture this property in a μ-calculus formula using the action labels of the model. Note that when `C1` enters the state Booted, it sends a message on port `m` to the environment, so the desired property is expressed by the following formula:

```
mu X. [true]X || <send(CompPortPair(C1,m),Value_String(STR_booted))
|receive(CompPortPair(Environment,m),Value_String(STR_booted))>true.
```

Using the mCRL2 toolset we can check the formula, which does not hold for the model. The toolset produces a counterexample file containing the part of state space that (dis)proves the formula. In this case we get a lasso shaped counterexample with a loop between the states Booting and Failed. The labels on the transitions in the trace are the same as in the mCRL2 model (`selectMultiStep`, `executeStep`, `executeBehaviour` and `send|receive`).

7 Discussion and Conclusion

One of the main benefits of our generic formalisation of the semantics of SysML in mCRL2 is that that it facilitates a straightforward automated translation. To have an automated translation from SysML to mCRL2 we only need to implement a tool that extracts the configuration data from a SysML model and prints the mCRL2 code as described in Sect. 6. Such a tool has recently been built and is discussed in a companion paper (see [22]).

Another benefit of directly formalizing in mCRL2 (compared to formalising in plain mathematics) is that the mCRL2 toolkit acts as an IDE. The parser and type checker of the editor root out the most obvious mistakes. Moreover, the model can be simulated when provided with a configuration of a simple set of state machines. This provides an additional way of verifying whether the semantics is as intended. There is still room for improvement of the mCRL2 toolset though: subtle mistakes in data equations can be hard to debug. Debugging techniques such as breakpoints and being able to step through term rewriting would be beneficial in this regard.

The statespace induced by a SysML model is potentially infinite as event queues can grow without bound. This happens when incoming messages trigger change events faster than the receiving component can process the events.

Since mCRL2 has an explicit-state model checker verification is no longer possible when the state space is infinite, though symbolic tools are in development [15]. The state space can be restricted by bounding the event queue, disallowing reception of messages until some events are processed. The downside of this approach is that the model 'loses' behaviour that could be analysed during model-checking. We leave it for future work to implement more sophisticated bounded event pools such as, e.g., the *controlled buffers* used in [1].

We reckon that significant optimization can be done to reduce the state space. One such optimization possibility was discovered in a case study of the EULYNX Point interface [4]. In our model all updates to variables are stored in the `vars` parameter of the `StateMachine` process. This is not always necessary; only when a variable is read in an action language expression do we really need to store the value. In particular, the variables associated to output ports are rarely referenced by the state machine. We could add a `referencedVariables` field to state machines and adjust the semantics to only remember the value of variables that are actually referenced. This would reduce the state space whilst preserving the behaviour modulo strong bisimilarity.

The UML standard does not give guidelines about the degree of interleaving in the execution of action language expressions. This ambiguity affects both the interleaving between state machines and the interleaving between parallel behaviours in a step. We would like to be able to generate mCRL2 models with varying interleaving models. The finest mode could break behaviour execution down to single instructions (such as looking up the value of a variable) and would allow the most detailed analysis. The coarsest mode could implement a run-to-completion semantics for parallel behaviour, reducing the state space. This variation can be realised by modifying the `ExecuteExcFrameCode` mapping.

Evidence provided by the mCRL2 toolkit (dis)proving properties is presented as an LTS with labels from the mCRL2 model. In the future we would like to improve usability by converting these evidence LTSs to UML sequence diagrams. This may not always be possible (or beneficial) as the evidence LTS may contain the entire state space. We reckon that some common evidence structures such as simple traces and lassos are well suited for conversion to sequence diagrams.

Concluding, we have shown how we have formalised the semantics of (SysML) state machines directly in mCRL2. The generic mCRL2 model is flexible and could be adjusted for a wide range of action languages. The step to an automated translation using our model is small and has been achieved in FormaSig.

Acknowledgement. FormaSig and, by extension, this work are fully funded by ProRail and DB Netz AG. The vision presented in this article does not necessarily reflect the strategy of DB Netz AG or ProRail, but reflects the personal views of the authors.

We also thank the anonymous reviewers for their constructive suggestions, which led to improvements of the paper.

References

1. Abdelhalim, I., Schneider, S., Treharne, H.: An integrated framework for checking the behaviour of fUML models using CSP. Int. J. Softw. Tools Technol. Transf. **15**(4), 375–396 (2013). https://doi.org/10.1007/s10009-012-0243-0
2. Belinfante, A.: JTorX: Exploring Model-Based Testing. Ph.D. thesis, University of Twente, Enschede, Netherlands (2014). http://purl.utwente.nl/publications/91781
3. Bouwman, M.: mCRL2 model capturing the generic semantics of EULYNX SysML. https://github.com/markuzzz/SysML-to-mCRL2
4. Bouwman, M., van der Wal, D., Luttik, B., Stoelinga, M., Rensink, A.: What is the point: formal analysis and test generation or a railway standard. In: Baraldi, P., Di Maio, F., Zio, E. (eds.) Proceedings of ESREL2020-PSAM15, pp. 921–928. Research Publishing, Singapore (2020). https://doi.org/10.3850/978-981-14-8593-0_4410-cd
5. Bunte, O., et al.: The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019). Lecture Notes in Computer Science, vol. 11428, pp. 21–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_2
6. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press (2014)
7. 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). https://doi.org/10.1007/s11334-009-0116-1
8. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M.R., van de Pol, J., dos Santos, O.M.: Automated verification of executable UML models. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. Lecture Notes in Computer Science, vol. 6957, pp. 225–250. Springer, Cham (2010). https://doi.org/10.1007/978-3-642-25271-6_12
9. Kim, S., Carrington, D.A.: A formal model of the UML metamodel: the UML state machine and its integrity constraints. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002: Formal Specification and Development in Z and B. ZB 2002. Lecture Notes in Computer Science, vol. 2272, pp. 497–516. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45648-1_26
10. Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 241–256. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45441-1_19
11. 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). https://doi.org/10.1007/s001659970003
12. Lilius, J., Paltor, I.: vUML: a tool for verifying UML models. In: The 14th IEEE International Conference on Automated Software Engineering, ASE 1999, Cocoa Beach, Florida, USA, 12–15 October 1999, pp. 255–258. IEEE Computer Society (1999). https://doi.org/10.1109/ASE.1999.802301
13. Lilius, J., Paltor, I.P.: The semantics of UML state machines (1999)
14. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J.S.: A formal semantics for complete UML state machines with communications. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 331–346. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38613-8_23

15. Neele, T., Willemse, T.A.C., Groote, J.F.: Solving parameterised boolean equation systems with infinite data through quotienting. In: Bae, K., Ölveczky, P.C. (eds.) FACS 2018. LNCS, vol. 11222, pp. 216–236. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02146-7_11
16. Object Managament Group: OMG Unified Modeling Language, version 2.5.1 (2017). https://www.omg.org/spec/UML/
17. Object Managament Group: Precise Semantics of UML State Machines (PSSM), version 1.0 (2019). https://www.omg.org/spec/PSSM/
18. Lilius, J., Paltor, I.P.: Formalising UML state machines for model checking. In: France, R., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 430–444. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-46852-8_31
19. Pedroza, G., Apvrille, L., Knorreck, D.: AVATAR: A SysML environment for the formal verification of safety and security properties. In: 11th Annual International Conference on New Technologies of Distributed Systems, NOTERE 2011, Paris, France, 9–13 May 2011, pp. 1–10. IEEE (2011). https://doi.org/10.1109/NOTERE.2011.5957992
20. 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.) NASA Formal Methods. NFM 2013. LNCS, vol. 7871, pp. 244–260. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_17
21. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electron. Notes Theor. Comput. Sci. **55**(3), 357–369 (2001). https://doi.org/10.1016/S1571-0661(04)00262-2
22. van der Wal, D., Bouwman, M., Stoelinga, M., Rensink, A.: On capturing the EULYNX railway standard with an internal DSL in Java. In: preparation for submission (2021)
23. Wang, H., Zhong, D., Zhao, T., Ren, F.: Integrating model checking with SysML in complex system safety analysis. IEEE Access **7**, 16561–16571 (2019). https://doi.org/10.1109/ACCESS.2019.2892745