Formal Verification of an Industrial UML-like Model using mCRL2

Anna Stramaglia(✉) and Jeroen J. A. Keiren

Eindhoven University of Technology, Eindhoven, The Netherlands
{a.stramaglia,j.j.a.keiren}@tue.nl

Abstract. Low-code development platforms are gaining popularity. Essentially, such platforms allow to shift from coding to graphical modeling, helping to improve quality and reduce development time. The Cordis SUITE is a low-code development platform that adopts the Unified Modeling Language (UML) to design complex machine-control applications. In this paper we introduce Cordis models and their semantics. To enable formal verification, we define an automatic translation of Cordis models to the process algebraic specification language mCRL2. As a proof of concept, we describe requirements of the control software of an industrial cylinder model developed by Cordis, and show how these can be verified using model checking. We show that our verification approach is effective to uncover subtle issues in the industrial model and its implementation.

1 Introduction

Abstract models are commonly used during the design phase of software. For example, class diagrams are used to describe the structure of a software system, and behavioral models describe the possible executions. Model checking can be used to verify that such behavioral models satisfy their requirements. While model checking is a promising technique, its industrial applications are still limited. There are several reasons for this. Among others, it is considered tedious to create a detailed behavioral model prior to implementing the system. Furthermore, model checking tools primarily use low-level, academic languages that require specific expertise not typically acquired by engineers in industry.

Low-code development platforms (LCDPs) [20] are gaining popularity. Such platforms focus on increasing the level of abstraction of software development, shifting from coding to graphical modeling, and generating code from these low-code models. LCDPs allow addressing both issues described above. First of all, the detailed behavioral model is now created during specification of the system. Second, if their semantics are well-understood, the models can be automatically translated to the languages used by state-of-the-art model checkers.

The Cordis SUITE[1] is an LCDP for machine-control applications, based on graphic Model-Driven Software Engineering. Its development environment is the Cordis Modeler, which uses Altova UModel[2] as front-end for drawing the models.

[1] https://www.cordis-suite.com.
[2] https://www.altova.com.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
J. F. Groote and M. Huisman (Eds.): FMICS 2022, LNCS 13487, pp. 86–102, 2022.
https://doi.org/10.1007/978-3-031-15008-1_7

Cordis models are described in a rich language that uses an extension of UML [16] class diagrams and state machine diagrams for describing the static structure and behavior, respectively. Additionally, it includes a large fragment of Structured Text [12]. Source code for Programmable Logic Controllers (PLCs) or the .NET platform can be generated directly from the models. Hence, the resulting implementation is consistent with the corresponding Cordis model. Cordis, the company developing this LCDP, has shown an interest in extending the Cordis SUITE with model checking capabilities.

Our contributions are as follows. We describe the structure and semantics of Cordis models, and automatically translate these to mCRL2 [9] to enable model checking. The use of mCRL2 is motivated by the availability of its tool set [2] with powerful verification tools such as simulation, model checking and the verification of first-order modal μ-calculus formulae [8]. We illustrate the feasibility of modeling and verification of Cordis models using a pneumatic *cylinder*. We specify, informally and formally, two typical requirements of the cylinder and verify whether they are satisfied by the model. One of the requirements is not satisfied by the cylinder model. We analyze its counterexample and identify a subtle issue in the model. The issue is reproducible in the implementation. A fix of the issue, now distributed by Cordis, is described and verified.

Related Work. A large amount of work has been done in the application of formal verification to industrial domains. Most of this work focuses on specific domains, such as railway infrastructure management [1,10,21] and medical applications [13,17,22]. Closer to our research are works on modeling and verification of control software, such as CERN's FSM language [11], which uses a strict hierarchical architecture of finite state machines for a specific machine control application, and OIL, developed and used by Canon Production Printing, which has a strong focus on separation of concerns [3].

Modeling languages such as SysML and UML can be used to model systems from any domain. The verification of state machine diagrams in these languages has been studied extensively, see, e.g., [1,6,14,15,19,23,24]. UML state machine diagrams are, e.g., transformed to Petri nets [15]. Others transform various UML behavioral diagrams into a single transition system for the model checker NuSMV [5]. The work closest to ours focuses on the verification of SysML state machines in the railway industrial domain [1]. Like in our work, state machines are assigned a formal semantics, and translated to mCRL2 for formal verification. Their semantics and execution model focuses on distributed execution of state machines communicating via queues, whereas our work focuses on a strictly sequential execution with communication via shared variables.

Outline. In Sect. 2 we introduce Cordis models. The cylinder model is described in Sect. 3. In Sect. 4 we describe the mCRL2 specification of Cordis models, the requirements of the cylinder model, and the results of its verification. Discussion and conclusions are presented in Sect. 5 and Sect. 6, respectively.

2 Cordis Models

The Cordis SUITE is a collection of tools for developing, testing and deploying system control software, with a focus on machine control. We consider three of these tools. The *Cordis Modeler* is an LCDP for creating machine-control applications. It uses an extension of the Unified Modeling Language (UML) such that every Cordis model describes the structure and behavior of a *machine* using class diagrams and state machine diagrams, respectively. Additionally, it can check for design errors, and generate source code for Programmable Logic Controllers (PLCs) or the .NET platform. The *Cordis Machine Control Server (MCS)* loads model information from the modeler, and connects to the PLC in order to exchange state information and data with the running system. The *Cordis Machine Control Dashboard (MCD)* is a Human-Machine Interface used to show live system data and live state machine diagrams when the PLC is running, providing real-time and historical information about the execution.

2.1 Class Diagrams

In this paper we illustrate the syntax and semantics of Cordis models, and their verification, using the concrete example of a pneumatic cylinder. The *static structure* of a Cordis model is described by its class diagram. The class diagram of the pneumatic cylinder is described in Example 1. Pneumatic cylinders are commonly used in factory automation systems for clamping, ejecting and lifting, and in industrial processes for materials handling and packaging. A pneumatic cylinder consists of a cylinder barrel with a piston that moves back and forth by means of compressed air controlled by electrically controlled valves.

Example 1. The cylinder we consider moves the piston between the *zero position* (completely retracted) and the *end position* (extended). Its class diagram consists of a single class, shown in Fig. 1. Classes can be tagged with stereotypes <<Machine>> and <<MachinePart>>, respectively, denoting the machine controlled by the system and a component of it. For the sake of simplicity, we consider the cylinder in isolation, but it typically is a machine part in a larger machine. A class

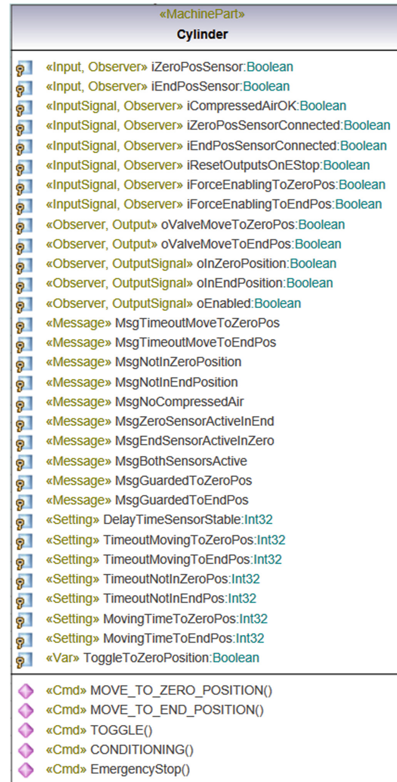

Fig. 1. Cylinder class

has *properties*, variables stored in the class, and *operations*, both tagged with Cordis-specific stereotypes describing their role in the system.

Stereotypes `<<Input>>` and `<<Output>>` describe variables used to interface with the environment, typically the hardware. Inputs `iZeroPosSensor` and `iEndPosSensor` detect whether the cylinder is at its zero or end position, respectively. The outputs `oValveMoveToZeroPos` and `oValveMoveToEndPos` are used to actuate the valves. Stereotypes `<<InputSignal>>` and `<<OutputSignal >>` are used to define shared variables to communicate between objects within the model, input signals are read by the cylinder and output signals are written by the cylinder. Stereotypes such as `<<Observer>>`, `<<Var>>`, `<<Setting>>` and `<<Message>>` are less important for the verification and ignored in this paper, see [25] for a more detailed explanation. Class operations, with stereotype `<<Cmd>>`, are commands issued (asynchronously) to the class, by the environment or another component. Commands `TOGGLE`, `MOVE_TO_ZERO_POSITION`, `MOVE_TO_END_POSITION`, and `EmergencyStop` are self-explanatory. Command `CONDITIONING` can be used to force (re)initialization of the cylinder.

2.2 State Machine Diagrams

Structure. The behavior of an object is defined using a hierarchy of state machine diagrams. Cordis state machine diagrams are similar to those defined in standard UML [16], with some Cordis specific details.

At the highest level, a state machine diagram consists of *top-level state machines*. A (top-level) state machine consists of a hierarchy of states and pseudo-states, whose types are mostly taken from standard UML, connected by transitions. A state can be a reference to a *subdiagram* or to a *substatemachine*. The key difference between these is whether they are executed as part of the diagram that references it (subdiagram) or separately (substatemachine). When a transition to a substatemachine is taken, control is transferred to the substatemachine.

Transitions have a *source* and *target* state, and can optionally be labeled by *guards* and *actions*. For a transition without guard, the guard is assumed to be *true* if the source of the transition is an initial state or an exit node, or the target of the transition is a choice node. If the source is a choice node, the empty guard is treated as *else*. Otherwise, the empty guard is assumed to be *false*.

Example 2. Consider the top-level state machine `Main` of the cylinder from our running example, shown in Fig. 2. From the initial state of `Main`, denoted by •, substatemachine `Disabled` is reached. This substatemachine has a number of states used to model different ways out of it (see Figs. 4 and 5). For instance, if `Disabled` determines that the cylinder is in its zero position it reaches state `CondInZero`, hence, guard `[State(Disabled.Conditioning.CondInZero)]` evaluates to *true* and state machine `Main` takes the transition to `In_Zero_Pos`.

Cordis models can contain *prestates* and *poststates* to model behavior that must be executed every time an object is allowed to execute a step, regardless of its current state. Pre- and poststates can either appear inside a state machine,

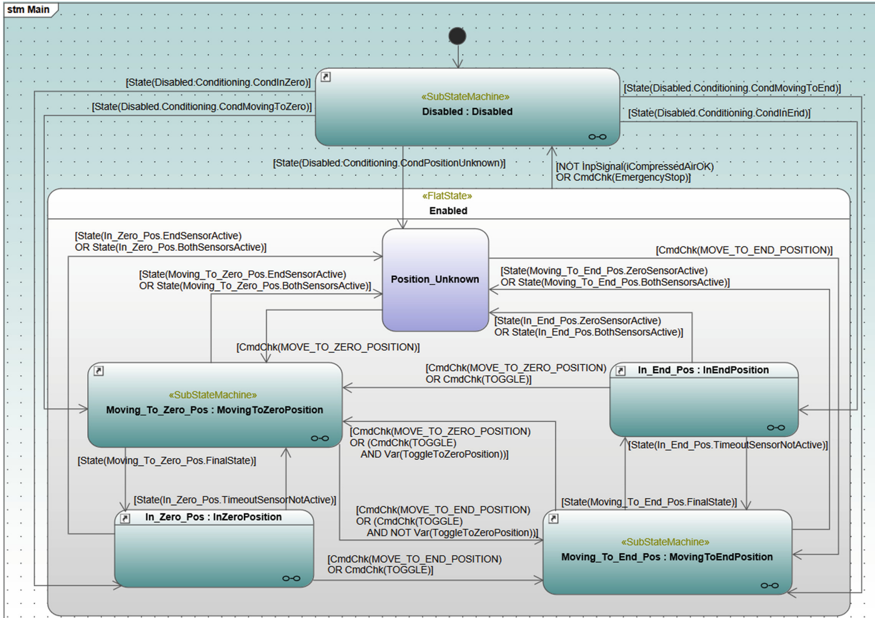

Fig. 2. State machine `Main`.

or at the top-level of the state machine diagram. The behavior of multiple pre- and poststates is always combined into a single pre- or poststate by taking the sequential composition of all pre- and poststates in a predefined order.

The poststate of the cylinder model updates output signals `oInEndPosition`, `oInZeroPosition` and `oEnabled` to reflect the cylinder's current position.

Semantics. Cordis models are executed using a cyclic execution model. In each cycle all objects execute in a predetermined order defined in the class diagram.

The order of execution within an object is depicted as an activity diagram in Fig. 3a. First the *inputs* are read. This essentially caches the current values of the inputs, input signals and the currently active command in local variables. To facilitate reasoning about the behavior of subsystems in isolation during verification, we also consider input signals and commands as free variables. Second, the (combined) prestate of the object is executed. If a new command was sent to the object, the *guard condition* is evaluated to determine whether the command can be accepted. If the guard condition is true, the *command action* is executed, otherwise the *reject action* is executed; these are defined in Structured Text by the user. The *command ready condition* determines whether, at the end of the current cycle, the command has been fully processed and can be removed from the interface. If this condition is false, the command will remain on the interface; if it is not overwritten by a new command, in the next cycle only the command ready condition is reevaluated. Since a single command can be evaluated per

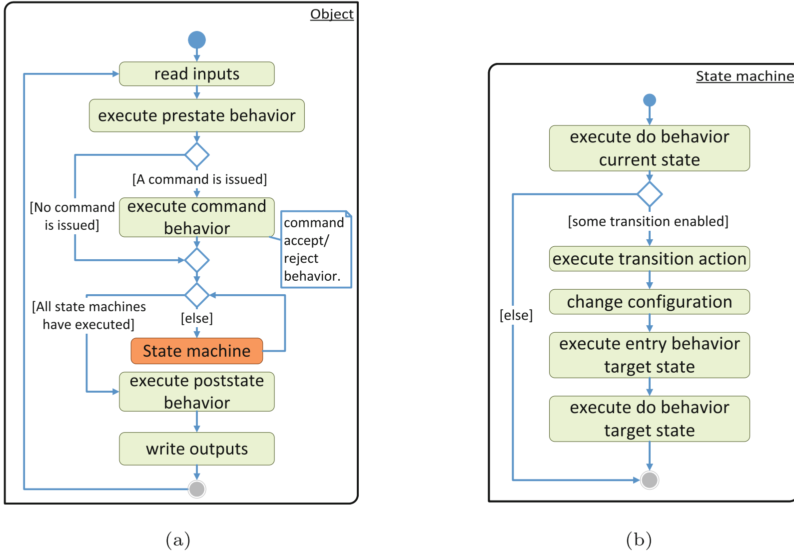

Fig. 3. Order of execution of (a) one object, and (b) a state machine within the object.

cycle, if a new command is issued it overwrites the previous one. After the command has executed, all state machines in the object execute in a predetermined order, in turns. Finally, the poststate is executed and the outputs are written.

Figure 3b depicts the execution of a single state machine. First, the do behavior of the current active state is executed. Second, if a transition is enabled in the current configuration, one such transition is executed. If multiple transitions are enabled, one is selected as follows: if the source state of one enabled transition contains the source state of another enabled transition, the transition from the outermost state is executed; transitions to other states take priority over self-loop transitions; otherwise the first transition from a predetermined order is executed.[3] If a transition was executed, the current state is changed and the behavior of the transition and the entry behavior of the target state are executed. If no transition is executed, the current state is unchanged.

3 Cylinder

We now describe the behavior of the cylinder model introduced in Sect. 2.

State machine `Main`, in Example 2, refers to substatemachines `Disabled`, `MovingToZeroPosition`, `MovingToEndPosition`, and it contains subdiagrams `InZeroPosition` and `InEndPosition`. We next elaborate on the details relevant for the verification, described in Sect. 4.2. For a full model description, see [25].

[3] Currently, the implementation chooses the order of creation.

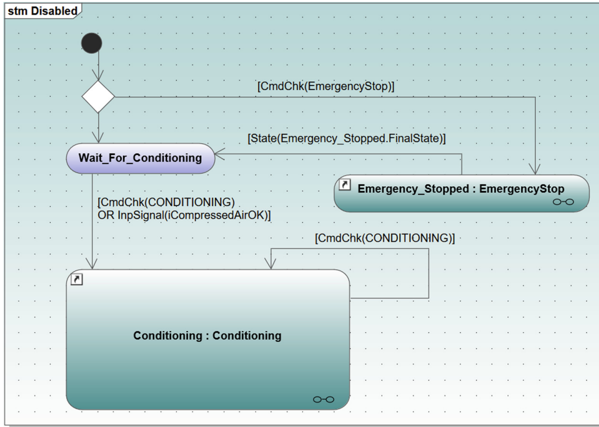

Fig. 4. Substatemachine `Disabled`

Disabled. From the initial state of `Disabled`, shown in Fig. 4, if command `EmergencyStop` was accepted, the system moves to subdiagram `Emergency Stop`, otherwise the system moves directly to `Wait_For_Conditioning`. Subdiagram `Conditioning` of `Disabled`, shown in Fig. 5, determines, based on the current values of the input signals, inputs and outputs, which state in `Main` reflects the current situation of the cylinder using a cascade of choice nodes. The cascade of choice nodes can be interpreted as an **if** ... **else if** ... **else** ... conditional. The states without outgoing transitions are used from state machine `Main` to determine the appropriate exit from `Disabled`.

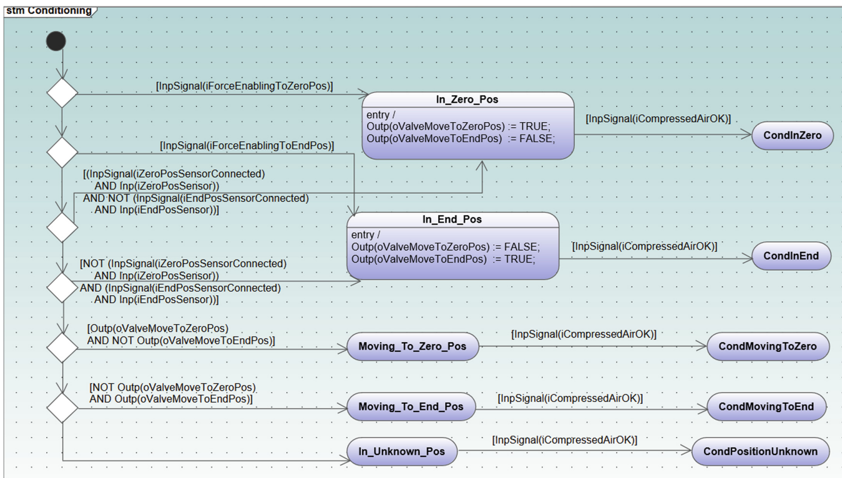

Fig. 5. State machine diagram `Conditioning`

MovingToZero- MovingToEndPosition and InZero- InEndPosition. The two substatemachines `MovingToEndPosition` and `MovingToZeroPosition` are symmetric, and describe the behavior of the cylinder when it is moving to the end position or to the zero position, respectively. Subdiagrams `InEndPosition` and `InZeroPosition` are also symmetric, and describe the behavior of the cylinder when it is completely extended or retracted, respectively.

4 Model Checking Cordis Models Using mCRL2

We enable formal verification of Cordis models through an automatic translation of the Cordis semantics to the modeling language mCRL2 [9]. The language is based on process algebra with data. Its associated tool set [2] can be used for modeling, validating and verifying systems. Although mCRL2 allows specifying communicating, parallel processes, the formalization of the semantics of Cordis models we present in this work only uses sequential processes. In the following subsections we describe our translation to mCRL2, see Sect. 4.1, and the formalization of a number of properties, see Sect. 4.2. We again use the model of a pneumatic cylinder (see Example 1) as a running example.

4.1 Translation to mCRL2

The mCRL2 specification of Cordis models consists of a sequence of several processes that model the behavior of the system. Essentially, each execution step shown in Fig. 3 is represented by a process in the mCRL2 specification.

The basic building block in a process is an action, such as `a`, `b`, that can be parameterized with data, e.g., `a(0)`, `b(`*`false`*`)`. When `p` and `q` are processes, the sequential composition `p.q` denotes the process in which first `p` is executed and upon termination, `q` is executed. The alternative composition, or choice, `p + q` denotes that either `p` or `q` is executed. Recursive processes can be defined by writing process equations of the form `P = q`, where `P` is the name of the process, and `q` is a process expression in which named processes are referred to.

In the mCRL2 specification of Cordis models, all processes are parameterized with the current configuration of the system, i.e., the current states of all state machines, and the current values of all class properties and operations. In what follows we sometimes omit (part of) the parameters, and write . . . instead. Each state machine in the model is identified uniquely by a nonnegative index.

Example 3. For the cylinder example discussed in this paper, the process describing the top-level of the system is as follows.

```
P_main(state_machine: Nat, s1: List(State), ..., cmd2: Command, cmd2_ready: Bool,
       cmd2_accepted: Bool, behaviors: List(Int), ...,
       M2'ToggleToZeroPosition: Bool, M2'iZeroPosSensor: Bool,
       M2'iEndPosSensor: Bool, M2'oValveMoveToZeroPos: Bool,
       M2'oValveMoveToEndPos: Bool, M2'iCompressedAirOK: Bool,
       M2'iZeroPosSensorConnected: Bool, M2'iEndPosSensorConnected: Bool,
       M2'iResetOutputsOnEStop: Bool, M2'iForceEnablingToZeroPos: Bool,
       M2'iForceEnablingToEndPos: Bool, M2'oInZeroPosition: Bool,
       M2'oInEndPosition: Bool, M2'oEnabled: Bool, ...) = P_set_inputs();
```

The parameter `state_machine` tracks the state machine that is currently executing. The current configuration of the system is tracked by, for every top-level state machine, a list of states `s1`, containing the currently active states. States are defined using **`sort`** `State =` **`struct`** `State_(state:`*`Nat`*`, entry:` *`List`*`(behavior), cont:`*`List`*`(behavior))` that is, unique identifier, `state`, and entry and continuous behavior, `entry` and `cont`. For every machinepart, indexed by an integer `i`, the command currently on the interface (along with some additional information) is kept in `cmdi`. Also class properties are stored as parameters. The machinepart `Cylinder` has index 2 and, e.g., input `iEndPosSensor` is stored as `M2'iEndPosSensor`, where `M2` refers to machinepart 2.

We next focus on the most relevant parts of the mCRL2 specification following the Cordis semantics execution. See [25] for a more detailed description.

At the beginning of each cycle, the values of the inputs are received by the system. As the inputs are not controlled by the system, we model these by receiving arbitrary values of the domain of the inputs.

Example 4. For the cylinder model this is formalized as follows.

```
P_set_inputs(..., M2'iZeroPosSensor: Bool, M2'iEndPosSensor: Bool, ...) =
    sum M2'iZeroPosSensor', M2'iEndPosSensor': Bool
    . inputs(M2'iZeroPosSensor', M2'iEndPosSensor')
 . P_set_free_input_signals(M2'iZeroPosSensor = M2'iZeroPosSensor',
                            M2'iEndPosSensor = M2'iEndPosSensor');
```

In this equation `P_set_inputs` is a parameterized process which represents the reception of the `<<Input>>` parameters. The *sum* denotes a generalized alternative composition that generates the choice between all four combinations of the input parameters. Subsequently, `P_set_free_input_signals` is called, where the new values of the inputs are assigned to the process parameters.

The process `P_set_free_input_signals` is similar to `P_set_inputs`, it allows setting arbitrary values to the input signals. This process in turn calls `P_set_free_commands`, which cycles through all machineparts to model commands that are issued by the environment. Issuing commands is modeled by a non-deterministic choice over all commands of the machinepart. Commands are indexed by an integer `i`. If no new command is issued this is indicated by action `no_freecmd`. If command `i` is issued this is indicated by action `freecmd(i)`.

Once all external inputs to the system have been established, the cyclic execution of machineparts is performed. In the case of the cylinder, only machinepart 2 needs to execute. First, the prestate is executed (which is empty in case of the cylinder). Subsequently, the command on the interface is executed.

Example 5. In the cylinder model, command `MOVE_TO_ZERO_POSITION` has index 6, and it is executed using the following code.

```
P_command_6(..., s1: List(State), ..., cmd2: Command,
          cmd2_ready: Bool, cmd2_accepted: Bool, ...) =
  (isCommand2_MOVE_TO_ZERO_POSITION(cmd2) && !cmd2_accepted)
    -> command(6, true)
      . P_command_6_exec(behaviors = accept(cmd2), cmd2_accepted = true,
                cmd2_ready = S79 in s1 || S103 in s1 || S89 in s1 || S93 in s1)
 + (isCommand2_MOVE_TO_ZERO_POSITION(cmd2) && cmd2_accepted)
    -> chk_ready . P_statemachines_M2(cmd2_ready = ...);
```

When a command is issued, `cmd2_accepted` is currently false. If the command guard evaluates to true, the second argument of the action `command` is *`true`*; otherwise it is set to *`false`*. If the command is accepted, the command accept behavior is listed for execution, indicated by `behaviors = accept(cmd2)`; if the command is rejected, `reject(cmd2)` is assigned instead. If the command was accepted in a previous cycle, `cmd2_accepted` is true, and action `chk_ready` is reported. In both cases, the command ready condition is evaluated in the assignment to `cmd2_ready`. Here it is true if the state machine is currently in one of four states. If the command was just accepted, the corresponding behavior is executed in `P_command_6_exec`, otherwise no transition behavior is executed.

Subsequently, the state machines execute. There is a separate process for each top-level state machine. In the cylinder, the corresponding process for `Main` is `P_statemachines_S1`. This cycles through all state machines in order, and allows each state machine to take a transition. Transitions are defined by **`sort`** `Transition =` **`struct`** `Transition_(source:`*`List`*`(State), dest:` *`List`*`(State), behavior:`*`List`*`(behavior))`, that is, its source states, its target states, and the behavior to execute if it is taken. The state machine process offers a non-deterministic choice over all transitions in the state machines.

Example 6. We give an example of one transition in the process of the cylinder. The other transitions are similar.

```
P_transitions_S1(state_machine: Nat, s1: List(State),...) =
  ...
 + (state_machine == 1 && (head(source(t100)) in s1)
     && (!M2'iCompressedAirOK || isCommand2_EmergencyStop(cmd2) && cmd2_accepted))
   -> trans(100)
    .P_execute_behaviors_S1(behaviors = behavior(t100) ++ entry(head(dest(t100))),
                 s1 = dest(t100) ++ remove_prefix(s1, rhead(source(t100))), ...)
 + ...
```

In this excerpt, `t100` refers to the transition with source state `Main.Enabled` and target state `Main.Disabled.InitialState` in Fig. 2, guarded by `[NOT InpSignal(iCompressedAirOK) OR CmdChk(EmergencyStop)]`.

The summand consists of a guard which says that state machine `Main` is executing, i.e., `state_machine == 1`, and source state `Main.Enabled` is part of the current configuration, i.e., `(head(source(t100))` *`in`* `s1)`. Furthermore, it checks if command `EmergencyStop` was accepted using `isCommand2_EmergencyStop(cmd2) && cmd2_accepted`.[4] In case the condition is satisfied, the action `trans(100)` is executed and `P_execute_behaviors_S1` is called in order to execute the behaviors labelling the transition (if any), `behavior(t100)`, as well as the entry behavior of the target state, `entry(head(dest(t100)))`. The next state that is reached in the state machine is `dest(t100) ++ remove_prefix (s1, rhead(source(t100)))`, where `dest(t100)` is the configuration reached after taking `t100`, and `remove_prefix(s1, rhead(source(t100)))` removes all the states that are left by taking `t100` from the configuration. Following the priority rules, transitions that have lower priority include the negation of the guards of all transitions with higher priority in their condition.

[4] Note that in mCRL2, `&&` (conjunction) binds stronger than `||` (disjunction).

After all state machines have executed one transition and the corresponding behavior, the poststate is executed. A pre- or poststate consists of Structured Text code that is translated to a sequence of mCRL2 processes. The poststate execution amounts to executing the corresponding processes.

For the poststate of the cylinder, this is modeled as follows.

```
P_poststate_M2(..., behaviors: List(Int), ...) =
  (behaviors == []) -> post_done.P_remove_command_M2()
 + (behaviors != [] && head(behaviors) == 3)
  -> post(3).P_3(behaviors = tail(behaviors));
```

In `P_3` the process parameters are updated, reflecting the poststate assignments.

After this, `P_poststate_M2` is reentered and transition `post_done` is taken, and if a command was on the interface and the command ready condition was true, it is removed from the interface and the process parameters for it are reset to their default value. Execution subsequently repeats from the beginning.

4.2 Formal Verification of Requirements

One of the primary goals of formalizing Cordis models using mCRL2 is to enable the formal verification of requirements. In this section, we first describe the requirements. Subsequently we discuss their formalization.

Requirements. In total, we have formulated 12 requirements for the cylinder, and formalized and verified them. Due to space limitations, in this section we describe one safety requirement and one liveness requirement. For details of the remaining requirements the reader is referred to [25].

The requirements we consider are the following two:

1. Invariantly, if one of the output signals `oInEndPosition` or `oInZeroPosition` is *true*, also output signal `oEnabled` is *true*.
2. Whenever output signal `oEnabled` is *false* and input signal `iCompressedAirOK` is *true*, inevitably output signal `oEnabled` becomes *true* unless command `CONDITIONING` is accepted.

Formalization of Requirements Using the Modal μ-calculus. We describe requirements using the first order modal μ-calculus [8]. This is a very expressive temporal logic that extends the μ-calculus with data.

In general, the requirements we are interested in refer to the interfaces of the machine parts, that is, their inputs, input signals, commands, output signals, and outputs. The first three are set explicitly in the translation. The output signals and outputs are only available implicitly, thus, in order to expose their values, we have extended the translation with self-loops. For this, we use actions such as `state_M2'oInEndPosition(true)`, where `state` indicates this is a stateloop, `M2` refers to the machinepart, `oInEndPosition` is the name of the output, and *true* is its current value. Similarly, we expose the current state of the system.

This is used to formalize the first requirement as follows.

```
[true*](<state_M2'oInEndPosition(true)||state_M2'oInZeroPosition(true)>true
        => <state_M2'oEnabled(true)>true)
```

This formula should be read as follows. First, [*true**] represents all sequences consisting of zero or more actions. After each such sequence the remainder of the formula should hold. For the remainder, note that formula <a>*true* holds in every state with an outgoing a transition. If we write the action formula a || b inside a modality, this matches the set of actions containing a, b; essentially, || here denotes the union of the sets of action represented by a and b, which are the singleton sets containing a and b, respectively. Hence, <state_M2'oInEndPosition (*true*)||state_M2'oInZeroPosition(*true*)>*true* holds in every state that has an outgoing transition labeled state_M2'oInEndPosition(*true*) or state_M2'oInZeroPosition(*true*). In each such state, the formula requires that also <state_M2'oEnabled(*true*)>*true* holds, i.e., the state has an outgoing transition labeled state_M2'oEnabled(*true*). We refer to [9] for a more extensive introduction to the first order μ-calculus.

The second requirement is formalized as follows.

```
nu X(enabled: Bool = false, compressedAirOk: Bool = false) .
  (forall e: Bool . <state_M2'oEnabled(e)>true =>
    ((forall c: Bool . [exists a2, a3, a4, a5, a6: Bool .
      free_input_signals(c, a2, a3, a4, a5, a6)]X(e,c)) &&
      [!exists a1, a2, a3, a4, a5, a6: Bool .
        free_input_signals(a1, a2, a3, a4, a5, a6)]X(e,compressedAirOk))) &&
  (forall e: Bool . [state_M2'oEnabled(e)]false =>
    ((forall c: Bool . [exists a2, a3, a4, a5, a6: Bool .
      free_input_signals(c, a2, a3, a4, a5, a6)]X(enabled,c)) &&
      [!exists a1, a2, a3, a4, a5, a6: Bool .
        free_input_signals(a1, a2, a3, a4, a5, a6)]X(enabled,compressedAirOk))) &&
  (val(!enabled && compressedAirOk) =>
    mu X . [!((exists a2, a3, a4, a5, a6: Bool .
                free_input_signals(false, a2, a3, a4, a5, a6)) ||
                command(9, true) ||
            (exists b: Bool . state_M2'oValveMoveToZeroPos(b) ||
                              state_M2'oValveMoveToEndPos(b) ||
                              state_M2'oInZeroPosition(b) ||
                              state_M2'oInEndPosition(b) ||
                              state_M2'oEnabled(b)) ||
            (exists i: Nat, l: List(Nat) . states(i, l))
    )]X || <state_M2'oEnabled(true)>true)
```

This formula uses a greatest fixed point (*nu*) and a least fixed point (*mu*). The greatest fixed point is parameterized by two Boolean variables, enabled and compressedAirOk, that keep track of whether oEnabled or iCompressedAir Ok have become *true*, respectively. In order to keep track of these values, we distinguish two cases. If a transition state_M2'oEnabled(e) is enabled, denoted by *forall* e: *Bool*. <state_M2'oEnabled(e)>*true*, we check if an action free_input_signals is enabled. If so, we determine the value assigned to iCompressedAirOk using *forall* c: *Bool* . [*exists* a2, a3, a4, a5, a6: *Bool* .free_input_signals(c, a2, a3, a4, a5, a6)]X(e,c)). We use *exists* inside the modality to represent generalised union, and match any value for the rest of the input signals. We update enabled to the value observed by the self-loop, and compressedAirOk to the value set in free_input_signals. If free_input_signals is not enabled, only compressedAirOk is updated. The case where state_M2'oEnabled(e) is not enabled is handled in a similar way.

Now, if `enabled` is *false*, and `compressedAirOk` is *true*, the least fixed point subformula needs to hold. To interpret this formula, we first note that formula `mu Y . [!a]Y || <b>true` captures that inevitably a state is reached where a `b` transition is enabled, unless an `a` transition happens. So, in principle, the following formula denotes that, as long as `iCompressedAirOk` does not become *false*, represented by the first argument to `free_input_signals`, and command `CONDITIONING` is not accepted, represented by `command(9, true)`, then we inevitably end up in a state where `oEnabled` is *true*.

```
mu Y . [!((exists a2, a3, a4, a5, a6: Bool .
           free_input_signals(false, a2, a3, a4, a5, a6)) ||
           command(9, true))]Y ||
       <state_M2'oEnabled(true)>true)
```

However, as we extended the model with self-loops, by taking such self-loops we trivially end up in an infinite sequence on which no state where `oEnabled` holds is reached. We therefore need to exclude paths through these self-loops.[5]

4.3 Results

We have verified the two properties from Sect. 4.2, as well as 10 additional requirements. For our experiments we have used Cordis Modeler version 3.14.1630. 7156 and mCRL2 tool set Release 202106.0. The cylinder model described and studied in this paper is relatively simple, its state space after reduction has 3049 states and 18736 transitions. This is reflected by the verification time: each of the properties can be verified in less than 5 s. Property 2 is *false*, and all of the other requirements are satisfied by the model. In case a property does not hold, the mCRL2 tool set offers a subset of the labeled transition system that underlies the cylinder specification as a counterexample that contains sufficient information to prove that the property is violated [26]. In the next section, we discuss the counterexample to property 2 in detail.

5 Discussion

The counterexample of requirement 2 has 39 states and 39 transitions. It is a transitions sequence that leads to a cycle on which `iCompressedAirOK` remains *true* and command `CONDITIONING` is never accepted, but `oEnabled` remains *false*, shown in Fig. 6. The counterexample follows the execution model of Sect. 2.2. In each cycle the state machines are executed in the predetermined order: `Main`, `MovingToZeroPosition`, `MovingToEndPosition` and `Disabled`.

For the sake of readability, in Fig. 6, the actions which are not essential to describe the trace are labeled with τ; a sequence of τ transitions is denoted by a dotted arrow labeled τ. We denote *true* and *false* as `tt` and `ff`, respectively.

[5] We here rely on the fact that the additional information is only exposed through self-loop transitions. This avoids the need for introducing an additional greatest fixed point.

The execution starts from the state with an unlabeled arrow pointing to it; in the first cycle, state `Main.Disabled.Wait_For_Conditioning` is reached via *trans*(*207*). In the second cycle, `iCompressedAirOK` is set to *true*, command `CONDITIONING` is accepted, indicated by *command*(*9*, `tt`), and, with *trans*(*174*) state `Main.Disabled.Conditioning.InitialState` is reached.

The third cycle starts in the loop, moving in counterclockwise direction. In this cycle (and subsequent) `iCompressedAirOK` remains *true*, no new command is issued, but action *chk_ready* expresses that command `CONDITIONING` is still on the interface. It follows that the self-transition in Fig. 4, *trans*(*175*), from state `Main.Disabled.Conditioning` to state `Main.Disabled.Conditioning .InitialState`, is taken. Subsequent cycles behave identically and state `Main .Disabled.Conditioning.InitialState` is infinitely often re-entered.

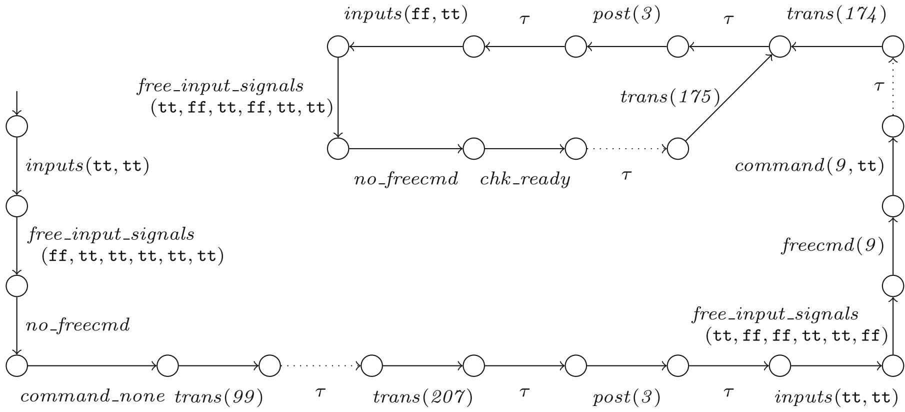

Fig. 6. Counterexample found verifying property 2

Reproducing the Counterexample. By loading the executing PLC code in the debugger provided by the PLC vendor and stepping through it, we can reproduce the exact behavior of the counterexample. This increases the confidence in the correctness of the translation to mCRL2, and proves that the counterexample contains ample information to be efficiently reproduced in the running system.

Root Cause Analysis and Solution. The system is able to loop in the self-transition from and to subdiagram `Conditioning`, in Fig. 4, because of two reasons: (1) in Cordis models, the outermost enabled transition gets priority over more deeply nested transitions, this is inherited from the semantics; and (2) the command continuously remains active on the interface. We focus our analysis on the latter. Command `CONDITIONING` has guard condition `State(Main.Disabled)`, command ready condition `NOT State(Main.Disabled)`, and no accept and reject actions. Thus, command `CONDITIONING` is accepted if the system currently is in

state machine `Disabled`, and the command is ready if the system leaves `Disabled`. In the counterexample, when command `CONDITIONING` is accepted, we are in state machine `Disabled`, the command ready condition is *false*, and the self-loop on `Conditioning` has higher priority than the transitions in Fig. 5.

Based on the analysis, we observe that command `CONDITIONING` behaves like a trigger that always remains high. The solution to avoid this behavior is to modify the command ready condition from `NOT State(Main.Disabled)` to *true*. This way, the command will only act as one single trigger to enter state machine conditioning: when issued and accepted, command `CONDITIONING` stays active on the interface for exactly one cycle. The change does not affect the relevant behavior of the cylinder model. Changing the cylinder model accordingly, and re-verifying the requirements shows that also requirement 2 holds.

6 Concluding Remarks

In this paper we have discussed the semantics of Cordis models, an extension of standard UML used for modeling complex machine-control applications, in order to enable the verification of these models. Even though Cordis models are not primarily designed for formal verification, we were able to characterize and implement an automatic translation to mCRL2. As a proof of concept we have verified the behavior of an industrial cylinder model against a number of requirements. Furthermore, we have shown that the verification process is effective to find bugs, and that these can be reproduced in the actual system.

There are some aspects to the formalization and verification process that we do not explicitly report in this paper. In particular, using earlier versions of the Cordis modeler and the mCRL2 translation, we have uncovered inconsistencies between the implementation of the models and the mCRL2 translation. Addressing those has resulted in modifications to both the semantics in the Cordis modeler and the mCRL2 translation. In order to understand and debug such issues, both having clear counterexamples in mCRL2, and the ability to step through the PLC code using a debugger have proven indispensable.

Future Work. Cordis models of complete industrial systems usually consist of multiple interacting objects. To this end, the translator from Cordis models to mCRL2 has been extended to deal with multiple component systems. We are currently expanding our work to deal with such complex models. In particular, we are investigating the use of symbolic model checking techniques [4] to deal with large state spaces. Furthermore, compositional model checking [18] could help in the verification of large models by incrementally generating state spaces of subsystems, reducing them, and combining them into larger subsystems, prior to verification. We are investigating improvements to static analysis tools to optimize mCRL2 models [7], and static analysis techniques such as dead variable analysis for Cordis models, reducing state space sizes. Finally, in our collaboration with Cordis, we are integrating model checking into the Cordis SUITE.

Acknowledgements. This work was supported partially by the MACHINAIDE project (ITEA3, No. 18030) and through EU regional development funding in the context of the OP-Zuid program (No. 02541). We thank Wieger Wesselink and Yousra Hafidi for contributions to the development of the mCRL2 translation, and Cordis Automation B.V. for their feedback on earlier versions of this paper.

References

1. Bouwman, M., Luttik, B., van der Wal, D.: A formalisation of SysML state machines in mCRL2. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 42–59. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_3
2. 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
3. Bunte, O., Gool, L.C.M., Willemse, T.A.C.: Formal verification of OIL component specifications using mCRL2. In: ter Beek, M.H., Ničković, D. (eds.) FMICS 2020. LNCS, vol. 12327, pp. 231–251. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_10
4. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A
5. Cimatti, A., et al.: NuSMV 2: an OpenSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
6. Dubrovin, J., Junttila, T.: Symbolic model checking of hierarchical UML state machines. In: 2008 8th International Conference on Application of Concurrency to System Design, pp. 108–117. ISSN: 1550–4808 (2008). https://doi.org/10.1109/ACSD.2008.4574602
7. Groote, J.F., Lisser, B.: Computer assisted manipulation of algebraic process specifications. ACM SIGPLAN Notices **37**(12), 98–107 (2002). https://doi.org/10.1145/636517.636531
8. Groote, J.F., Mateescu, R.: Verification of temporal properties of processes in a setting with data. In: Haeberer, A.M. (ed.) AMAST 1999. LNCS, vol. 1548, pp. 74–90. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49253-4_8
9. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014). https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems
10. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M.R., van de Pol, J.: Towards model checking executable UML specifications in mCRL2. Innov. Syst. Softw. Eng. **6**(1–2), 83–90 (2010). https://doi.org/10.1007/s11334-009-0116-1
11. Hwong, Y.L., Keiren, J.J.A., Kusters, V.J.J., Leemans, S., 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). https://doi.org/10.1016/j.scico.2012.11.009
12. John, K.H., Tiegelkamp, M.: The programming languages of IEC 61131–3. In: John, K.H., Tiegelkamp, M. (eds.) IEC 61131–3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids, pp. 99–205. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12015-2_4

13. Keiren, J.J.A., Klabbers, M.D.: Modelling and verifying IEEE Std. 11073–20601 session setup using mCRL2. Electron. Commun. EASST **53** (2013). https://doi.org/10.14279/tuj.eceasst.53.793
14. Liu, S., et al.: 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. Lyazidi, A., Mouline, S.: Formal verification of UML state machine diagrams using petri nets. In: Atig, M.F., Schwarzmann, A.A. (eds.) NETYS 2019. LNCS, vol. 11704, pp. 67–74. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31277-0_5
16. Object Management Group: OMG Unified Modelling Language (UML). Technical report Version 2.5.1 (2017). https://www.omg.org/spec/UML/2.5.1/PDF
17. Pore, A., et al.: Safe reinforcement learning using formal verification for tissue retraction in autonomous robotic-assisted surgery. In: 2021 IEEE/RSJ IROS, pp. 4025–4031 (2021). https://doi.org/10.1109/IROS51168.2021.9636175. ISSN: 2153-0866
18. de Putter, S., Wijs, A.: Compositional model checking is lively. In: Proença, J., Lumpe, M. (eds.) FACS 2017. LNCS, vol. 10487, pp. 117–136. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68034-7_7
19. Rodríguez, R.J., Fredlund, L.Å., Herranz, Á., Mariño, J.: Execution and verification of UML state machines with erlang. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 284–289. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_22
20. Sahay, A., Indamutsa, A., Ruscio, D.D., Pierantonio, A.: Supporting the understanding and comparison of low-code development platforms. In: 2020 46th Euromicro Conference on SEAA, pp. 171–178 (2020). https://doi.org/10.1109/SEAA51224.2020.00036
21. Salunkhe, S., Berglehner, R., Rasheeq, A.: Automatic transformation of SysML model to event-B model for railway CCS application. In: Raschke, A., Méry, D. (eds.) Rigorous State-Based Methods. LNCS, vol. 12709, pp. 143–149. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77543-8_14
22. Santone, A., et al.: Radiomic features for prostate cancer grade detection through formal verification. La radiologia medica **126**(5), 688–697 (2021). https://doi.org/10.1007/s11547-020-01314-8
23. Santos, L.B.R., Júnior, V.A.S., Vijaykumar, N.L.: Transformation of UML behavioral diagrams to support software model checking. In: FESCA 2014. EPTCS, vol. 147, pp. 133–142 (2014). https://doi.org/10.4204/EPTCS.147.10, arXiv: 1404.0855
24. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. ENTCS **55**(3), 357–369 (2001). https://doi.org/10.1016/S1571-0661(04)00262-2
25. Stramaglia, A., Keiren, J.J.A.: Formal verification of an industrial UML-like model using mCRL2 (extended version) (2022). arXiv: 2205.08146
26. Wesselink, W., Willemse, T.A.C.: Evidence extraction from parameterised Boolean equation systems. In: Benzmüller, C., Otten, J. (eds.) proceedings of ARQNL 2018 affiliated with IJCAR 2018, Oxford, UK, 18 July 2018. CEUR, vol. 2095, pp. 86–100. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2095/paper6.pdf