BEHAVIOURAL MODELS FOR DEVICE CONTROL

L. Andolfato†, M. Comin, S. Feyrin, M. Kiekebusch, J. Knudstrup, F. Pellegrin, D. Popovic,
C. Rosenquist, ESO, Garching bei München, Germany
R. Schmutzer, ESO, Paranal Observatory, Chile

Abstract

ESO is in the process of designing a new instrument control application framework for the ELT project. During this process, we have used the experience in HW control gained from the first and second generation of VLT instruments that have been in operation for almost 20 years. The preliminary outcome of this analysis is a library of Statecharts models illustrating the behaviour of some of the most commonly used devices in telescope and instrument control systems. This paper describes the architectural aspects taken into consideration when designing the models such as HW/SW state representation, common/specialized behaviour, and failure management. An extension to Harel's formalism to facilitate reusability by dynamic creation of orthogonal regions is also proposed. The paper details the behaviour of some devices like shutters, lamps and motors together with the rationale behind the modelling choices. A mapping of the models to a concrete implementation using real HW components is suggested. Although these models have been designed following the principles of our conceptual architecture, they are still generic and platform independent, so they can be easily reused in other projects.

INTRODUCTION

For more than 20 years, the Control Instrument Software group at the European Southern Observatory (ESO), has provided to universities and consortia a software framework to build instruments for the Very Large Telescope (VLT) and Interferometer (VLTI) facilities located at Cerro Paranal in the Atacama desert in Chile.

Part of the framework is dedicated to the monitoring and control of devices such as shutters, lamps, motors, and piezos. For each type of device there are several implementations available on the market. These implementations usually differ in some mechanical or electrical characteristics like accuracy, speed, size, and power consumption, however their logical behaviour is often very similar. The goal of this paper is to promote the creation of libraries of behavioural models for devices commonly used in control systems, so that they can be shared in various projects and organizations. These models could be reused for design documentation, system analysis, simulation, and model transformations.

The rest of the paper is organized in five sections. The first section focuses on the motivation for the adoption of StateCharts XML as the modelling language. The following section describes an extension to the Statecharts formalism to introduce the concept of templates in the domain of state machines. The next two sections are dedicated to the description of the devices' common and specific behavioural models. The last section provides some indication on how to map the models into concrete SW artefacts.

MODELING BEHAVIOUR

The selection of the modelling language has been driven by two main requirements:

1. It shall allow to create models that are independent from specific implementation platforms
2. Syntax and semantic shall be standard and precise.

The motivation for the first requirement is to facilitate the usage of models in projects that have adopted different technologies and tools. The second requirement aims to avoid misinterpretation and to allow automatic model transformation and execution.

We have evaluated three possible modelling languages all based on variations of the Statecharts formalism [1, 2]:

- SysML State Machines
- OPC-UA data model for State Machines
- StateChart XML

SysML language [3] has been standardized by the Object Management Group (OMG) but only a subset of the language, the so called Foundational UML (fUML), has precise semantic [4]. Recently OMG has started a working group to specify the semantic of SysML/UML State Machines: the Precise Semantic for State Machines (PSSM) [5]. Unfortunately, no recommendation has been released yet and we are not aware of any available implementation.

Since the devices we want to model are very often controlled via PLCs, we investigated the possibility of modelling their behaviour using OPC-UA data model which is a de-facto standard for industrial automation and is defined by the OPC Foundation [6]. OPC-UA data model offers a syntax to model a subset of the Statecharts features leaving the definition of the missing parts to the user. The semantic specification is not provided.

StateChart XML (SCXML) is a recommendation released in 2015 by the World Wide Web Consortium (W3C) specifying an event-based state machine language derived from Statecharts [7]. At the moment of writing, SCXML seems to be the only option that provides a precise syntax and semantic definition and that can be easily exchanged thanks to the textual XML representation.

Textual models are easy to edit and compare but they can be more difficult to understand than diagrams. This is especially true for Statecharts since the notation takes advantage of intuitive topological concepts like composition [8]. To overcome this problem, we have defined a mapping between SysML/UML State Machines and SCXML and developed an open source tool, called COMODO, to transform SysML/UML State Machine models, saved in Eclipse

† landolfa@eso.org

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPL02

Modeling Framework XMI format, into SCXML documents [9, 10].

SCXML documents can be executed at run-time by interpreters that conform to the SCXML standard. W3C provides the source code in a Lisp-like language for a possible implementation of the SCXML execution algorithm. It also provides a set of test cases with the expected results that can be used to verify the compliance of the implementation to the standard syntax and semantic.

EXTENDING STATECHARTS

Traditionally Statecharts models were translated into code to be compiled. To take effect, a change in the behaviour defined in the model would require a recompilation of the application implementing the model. With an SCXML interpreter, the model is parsed and loaded in memory. It is therefore possible to apply on-the-fly modification at application start-up (e.g. during the parsing of the SCXML document) or even at run-time (in this case a re-initialization of the application may be required). The ability of changing the model when the application starts-up allows the customization/specialization of generic Statecharts "templates". As an example, consider the case of an instrument being able to control a configurable number N of identical detectors. One possible way of modelling such an instrument is to dedicate one orthogonal region for each detector. With the compiled based Statecharts, this approach cannot be adopted since the number of regions is known only at start-up/configuration time and not at modelling or compilation time.

In SCXML, or in general with Statecharts interpreters, it would be enough to model one single region and to tell the parser at start-up to create N clones of that region. To implement such a feature, the name of the Statecharts elements that need to be cloned, are marked using an identifier within two special characters: '#identifier#'. For example: the event START to start acquiring images can be marked with START_D# and, if cloned twice, would be transformed into two events: "START_D1" and "START_D2". The cloning of Statecharts elements that contains other sub-elements, e.g. composite states or orthogonal regions, requires that every fully contained sub-element is also cloned whether marked or not. An example of sub-elements which are not fully contained are the transitions entering or leaving an orthogonal region that is marked to be cloned.

This extension in the Statecharts notation helps in reducing the modelling effort and increasing model re-usability.

COMMON BEHAVIOURAL PRINCIPLES AND CONVENTIONS

The models of the devices presented in the next section follows some common design principles and conventions which are reported hereafter.

Events Abstraction

The events that trigger the state transitions in our models are an abstraction of the real platform specific HW signals and SW commands, timers, notification, etc. To facilitate the understanding of the model, events are represented by identifiers in capital case with a postfix indicating the type of event: "_CMD" for requests, "_SIG" for HW signals, and "_INT" events generated by the internally by the control application.

Actions and Do-Activities Abstraction

An action is a task that can occur during a transition (transition action) or when entering or leaving a state (entry/exit action). Its duration should be short since it blocks the processing of other incoming events.

A do-activity is a long-lasting task that is started when entering a state and is terminated when the state is exited. The do-activity is executed concurrently with the processing of other events and is usually implemented with a dedicated thread or using co-routines.

In SCXML it is possible to specify the full implementation of the actions directly in the model. We decided to keep the models simple and platform independent by using simple identifiers (in CamelCase style) to specify actions and do-activities (to distinguish between the two, do-activities identifiers have a "Do" prefix) and to leave the development of the corresponding code at the implementation phase. For each action the developer is supposed to implement a free function or a method of a class and for each do-activity the corresponding threading behaviour.

State Semantics

The states used in our models represents both the status of the HW under control and the control SW status. In case of ambiguities, priority is given to the HW. For example, consider an open command successfully applied to a shutter device. After a while the shutter is closed manually by a person. From the SW perspective, the state should be OPEN because of the last command successfully applied while for the HW point of view the state should be CLOSED due to the HW signal. With our convention, the priority is given to the HW and therefore the state is CLOSED. An alternative solution is to model HW and SW states separately using orthogonal regions. This approach presents the drawback of increasing the size and complexity of the model.

Root State

Our models present a composite state that include all other states. This state, called in the examples "ROOT", does not represent any real HW or SW state. It is used to deal with events that should be processed in any state, like a base class in object-oriented design. For example, a STATUS_CMD that returns the current state configuration of the device can be modelled as an internal transition in the ROOT state so that it is accepted at any time. Similarly, it should be possible to terminate the application via the EXIT_CMD regardless of the current state configuration. Figure 1 shows the ROOT state using the graphical SysML/UML Statecharts notation while the corresponding SCXML representation is shown in Figure 2.

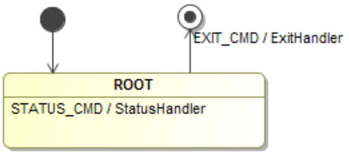
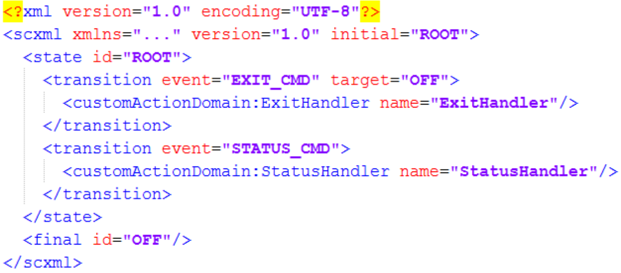


Figure 1: The ROOT state with an internal transition to deal with the STATUS_CMD command.



```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="..." version="1.0" initial="ROOT">
  <state id="ROOT">
    <transition event="EXIT_CMD" target="OFF">
      <customActionDomain:ExitHandler name="ExitHandler"/>
    </transition>
    <transition event="STATUS_CMD">
      <customActionDomain:StatusHandler name="StatusHandler"/>
    </transition>
  </state>
  <final id="OFF"/>
</scxml>
```

Figure 2: The SCXML representation of ROOT state.

Common and Specialized Behaviour

Within the ROOT state two types of behaviour are modelled: the behaviour common to all devices and the one that is specific to the given type of device (shutter, lamp, motor, etc.). The common behaviour is like a protocol that all devices implement to facilitate monitoring and control activities. It is composed of two states: STANDBY, to deal with the device initialization, and OPERATIONAL, to indicate that the device is ready for operation, as shown in Figure 3. STANDBY contains NOTREADY, INITALIZING, READY, and FAILURE sub-states describing the state of the device before, during and immediately after the initialization procedure. The device initialization is triggered by the INIT_CMD and performed in the INITIALIZING state by the DoInit activity. If the procedure succeeds, the state READY is reached otherwise the transition to NOTREADY is taken. The actions InitStart, InitComplete, and InitAbort can be used to implement the start and the successful or unsuccessful completion of the procedure. HW failures and SW errors occurring during the initialization procedures are detected by the DoInit activity and modelled with the ERRINIT_INT event. HW failures within the STANDBY state but outside the scope of the initialization procedure are modelled with the ERRHW_SIG event which brings the system to NOTREADY. The RESET_CMD can be used from any state to go back to NOT_READY and allows to repeat the initialization procedure. The transitions from STANDBY to OPERATIONAL and vice-versa can be performed using the ENABLE_CMD and DISABLE_CMD commands.

The specialized device behaviours, described in the next section, are modelled by adding sub-states and transitions to the OPERATIONAL and, if needed, STANDBY/READY states.

An alternative approach is to use two orthogonal regions: one to model the common behaviour and another one for the specialized behaviour. However, the states of the common behaviour may not be compatible with the specific behaviour states. For example, a motorized device which is in STANDBY/READY should not be allowed to be in also in MOVING state. These incompatibilities could be avoided using guards on the transitions with the undesirable effect of increasing the model complexity.

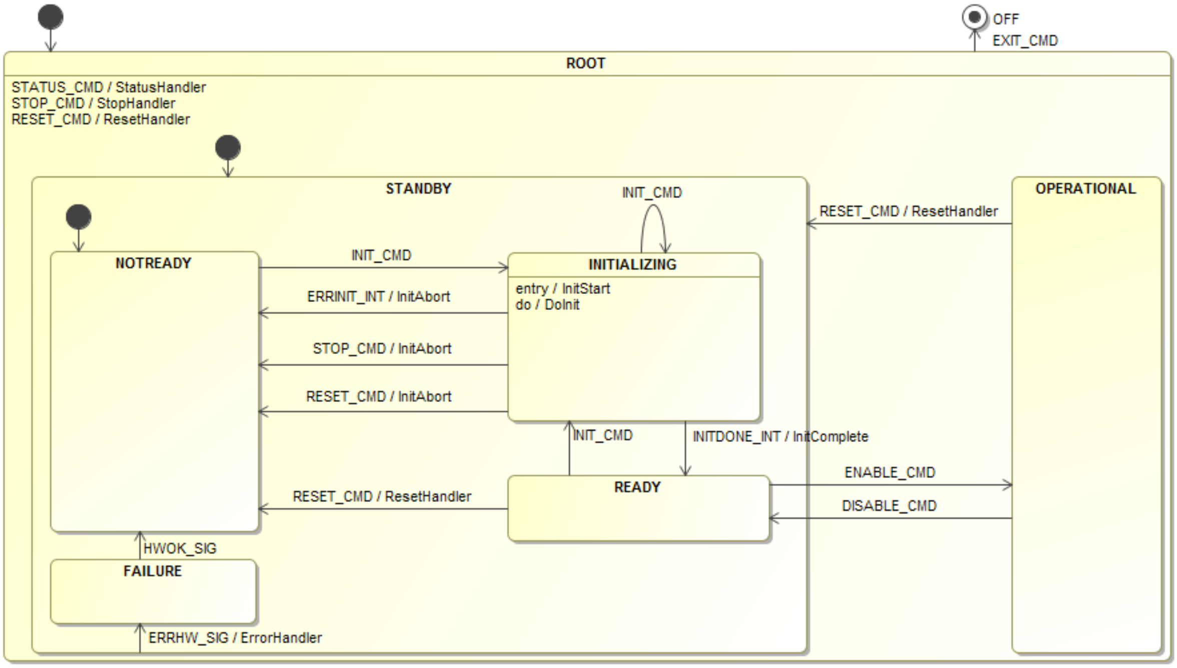


Figure 3: Diagram of the behaviour common to all devices.

Failure Management

We have adopted two different strategies to deal with failures. Failures occurring during system start-up or initialization (i.e. within the STANDBY state) should force a full re-initialization since the goal is to prepare the system for reliable operation (e.g in Figure 3 the signal HWOK_SIG to recover from a HW failure brings the system to STANDBY/NOTREADY regardless of whether the device was already initialized). On the other hand, during operation (i.e. within the OPERATIONAL state), the goal is to try to increase system availability (i.e. system downtime should be minimized) and therefore a failure should not force a full re-initialization of the device. If a full re-initialization is required, it should be invoked explicitly via the RESET_CMD.

LIBRARY OF DEVICE MODELS

We have modelled the behaviour of the following type of devices:

- Digitally controlled shutters
- Lamps with intensity control and digital or analog feedback
- DC and Stepper motors
- Multi-axes analog piezos

Shutter

The specific behaviour of a shutter is shown in Figure 4. Digitally controlled shutters have two physical states at rest: OPEN and CLOSED. The initialization procedure loads and applies the initial device configuration and, if successful, transitions to STANDBY/READY/OPEN or CLOSED, depending on the configuration.

When enabled via the ENABLE_CMD, the device becomes OPERATIONAL. Within the OPERATIONAL composite state, the sub-states OPENING and CLOSING have been introduced to support slow shutters. The DoOpen and DoClose activities are in charge of starting the movement of the device and monitoring its position. Once the final position has been reached the HW (or the do-activities) can trigger the ISOPEN_SIG or ISCLOSED_SIG events to transition to the steady state OPEN or CLOSED.

In case of failures while opening or closing, the ERR_INT event is generated by the do-activities to transition to OPERATIONAL/FAILURE. This includes the case of too slow movement and any error conditions detected by the SW. HW failures, for example loss of communication, are signalled via the ERRHW_SIG event which moves the system in OPERATIONAL/FAILURE state. It is possible to recover by trying to open/close the shutter using the OPEN_CMD/CLOSE_CMD or via the RESET_CMD that would then force a re-initialization of the device. Recovery is automatic when the HW indicates, via the ISCLOSED_SIG/ISOPEN_SIG signals, the actual state of the shutter.

In case of inconsistencies while in a steady state, the HW has priority as it is indicated by the transitions from OPEN to CLOSED on ISCLOSED_SIG event and vice-versa on ISOPEN_SIG event.

Lamp

The specific behaviour of a lamp is shown in Figure 5. In addition to the ON and OFF states, a lamp may need to warm up to reach a stable light source (in intensity and wavelength). For that purpose the WARMING state has been added. Some lamps are sensitive to frequent on/off switching and are protected by a cooling down cycle represented in the model by the COOLING state. The DoSwitchOn/Off activities switch the device on/off and wait for the ISON_SIG/ISOFF_SIG feedback from the device before transitioning to the WARMING/COOLING states. It is possible to configure the device so that the WARMING/COOLING states are bypassed. In this case the DoSwitchOn/DoSwitchOff activities would trigger the ISWARM_INT/ISCOOL_INT event instead of waiting for the ISON_SIG/ISOFF_SIG feedback.

In the WARMING/COOLING states the DoWarmUp and DoCoolDown activities wait for a certain configurable period to allow reaching the correct temperature.

The rest of the model is similar to the shutter device, including the error and failure handling.

Note that it is possible to stop the warming up of the lamp but not the cooling down. This is a safety measure to avoid damaging the device.

Motor

The behaviour of a motor device is shown in Figure 6. The model allows to move, stop, and calibrate the axis. The calibration of the axis is represented by the SETTINGPOS state and is triggered by the SETPOS_CMD. The movement of the motor, can be in position or velocity. It is represented by the MOVING state and is triggered by the MOVE_CMD. In case of moving the motor in position, the DoMove activity detects the completion of the movement and issues the MOVEDONE_INT event to transition to the STANDSTILL state. In case of moving the motor in velocity, the STOP_CMD is needed in order to stop the movement and cause the transition to STANDSTILL. It is also possible to interrupt ongoing motion with another MOVE_CMD and to change on-the-fly the velocity and the target position, or to change from position to velocity or vice versa

When moving, the DoMove activity checks for SW position limits, following errors, etc. and reports them with an ERR_INT event (or dedicated events) which in turn triggers a transition to OPERATIONAL/FAILURE.

Depending on the type of failure (e.g. following error), it might be necessary to reinitialize the device (via the RESET_CMD) or it might be possible to recover by simply issuing a new MOVE command (e.g. in case of hitting a SW limit).

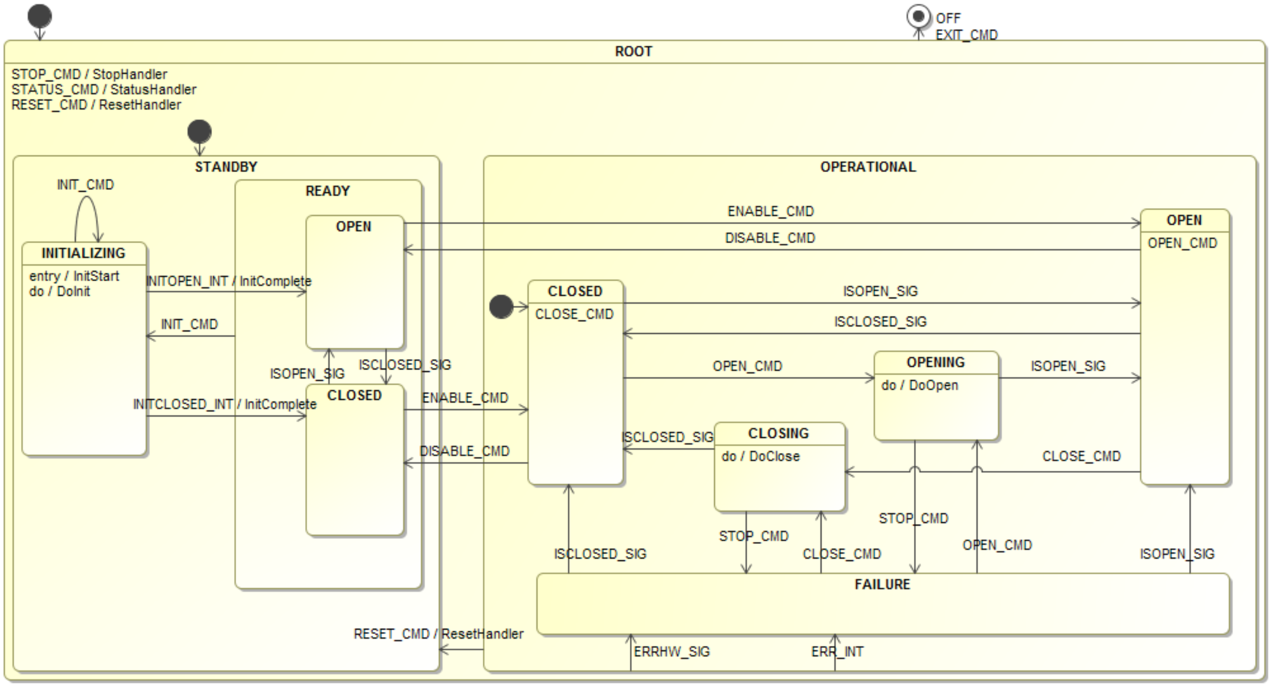


Figure 4: Diagram of the shutter specific behaviour. The STANDBY/NOTREADY and STANDBY/FAILURE states with related transitions are not shown since are identical to Figure 3. Actions are not shown to improve readability.

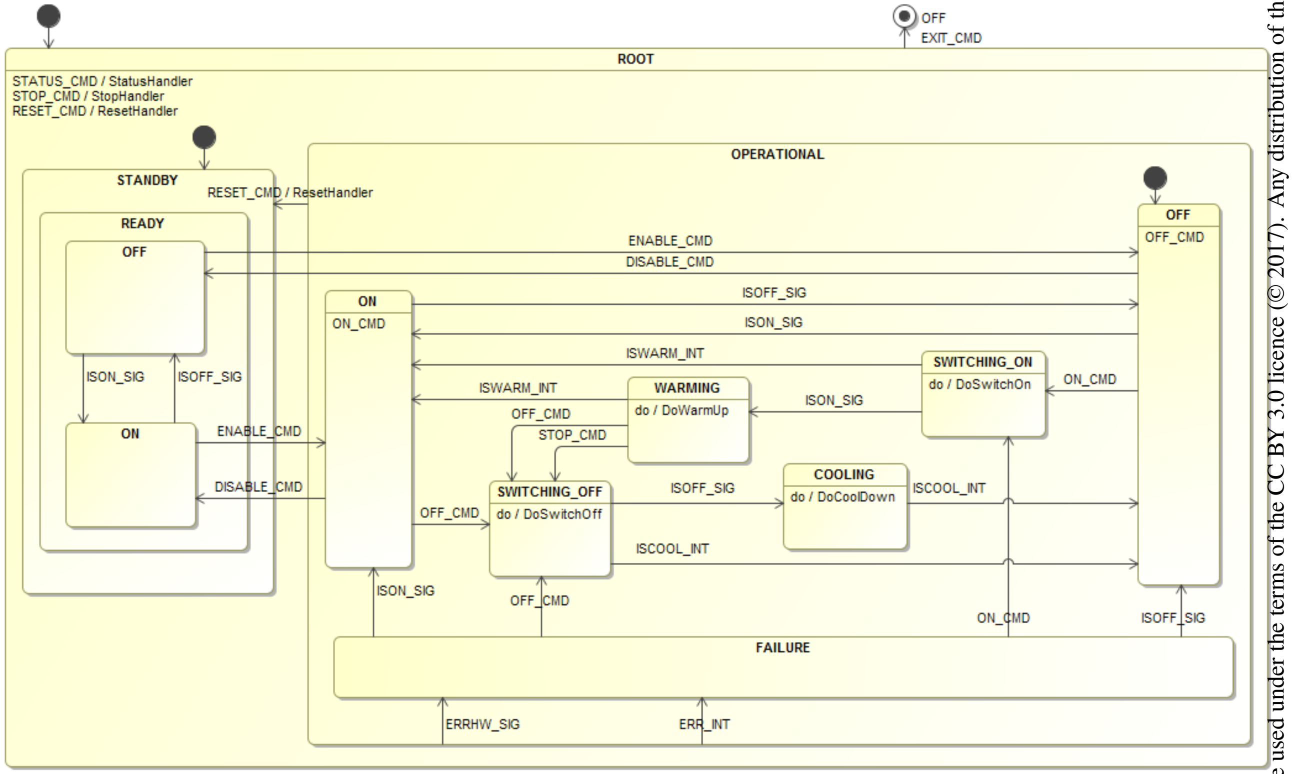


Figure 5: Diagram of the lamp specific behaviour. The STANDBY/NOTREADY, STADBY/FAILURE and STANDBY/INITIALIZING states with related transitions are not shown since they are similar to the shutter.

Motor

The behaviour of a motor device is shown in Figure 6. The model allows to move, stop, and calibrate the axis. The calibration of the axis is represented by the SETTINGPOS state and is triggered by the SETPOS_CMD. The movement of the motor, can be in position or velocity. It is represented by the MOVING state and is triggered by the MOVE_CMD. In case of moving the motor in position, the DoMove activity detects the completion of the movement and issues the MOVEDONE_INT event to transition to the STANDSTILL state. In case of moving the motor in velocity, the STOP_CMD is needed in order to stop the movement and cause the transition to STANDSTILL. It is also possible to interrupt ongoing motion with another MOVE_CMD and to change on-the-fly the velocity and the target position, or to change from position to velocity or vice versa

When moving, the DoMove activity checks for SW position limits, following errors, etc. and report them with an ERR_INT event (or dedicated events) which in turn trigger a transition to OPERATIONAL/FAILURE.

Depending on the type of failure (e.g. following error), it might be necessary to reinitialize the device (via the RESET_CMD) or it might be possible to recover by simply issuing a new MOVE command (e.g. in case of hitting a SW limit).

Piezo

The behaviour of a multi-axes piezo device is shown in Figure 7. The device can be moved to a given target position, represented by the INPOS state, using the SETPOS_CMD or it can follow a trajectory, within the AUTOPOS state, computed and applied by the DoComputePos activity. Since the positioning of a piezo device is almost immediate (usually no feedback is provided), there is no need for transition states like MOVING in the case of the motor device. It is enough to execute the SetPosition action that applies the voltage to the piezos, to execute a SETPOS_CMD request.

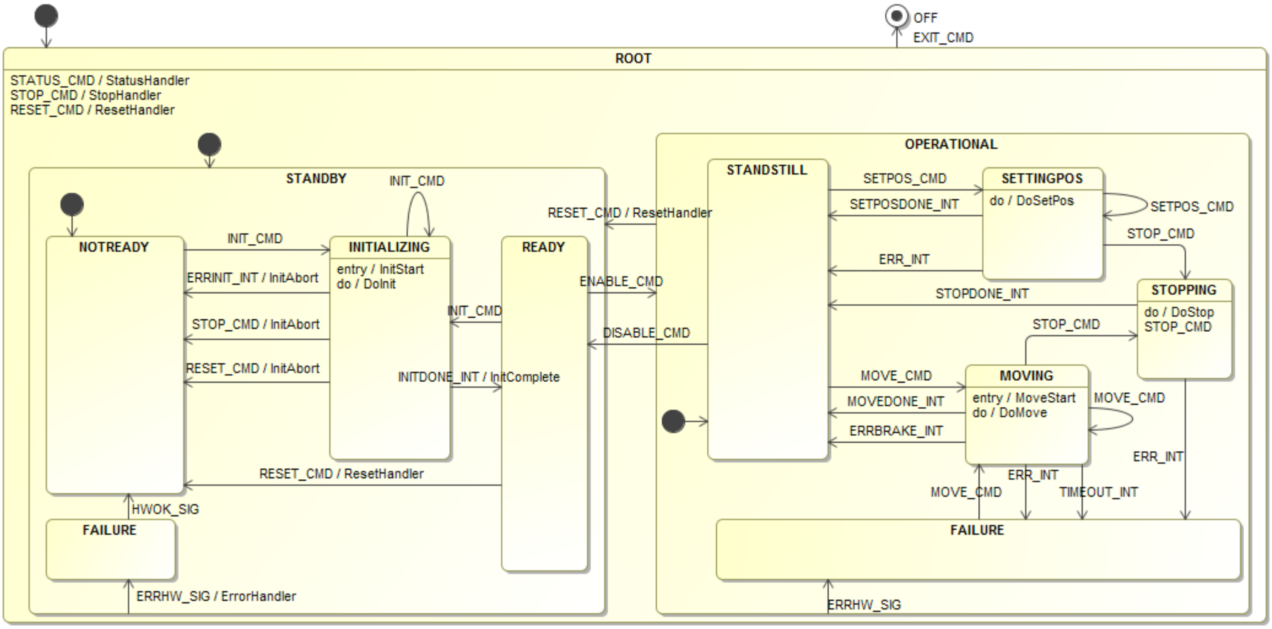


Figure 6: Diagram of the motor device behaviour.

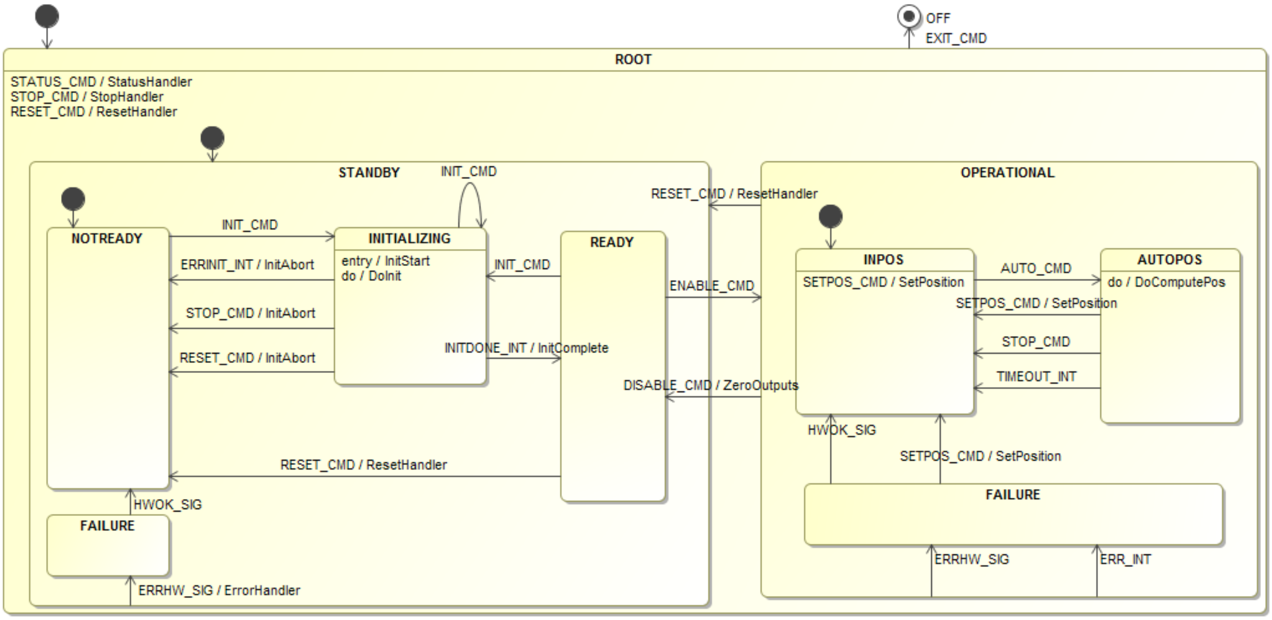


Figure 7: Diagram of the piezo device behaviour.

Any error encountered by the DoComputePos activity or the SetPosition action (e.g. position out of range) is reported via the ERR_INT event (or using dedicated events) and leads to the OPERATIONAL/FAILURE state. Recovery from the OPERATIONAL/FAILURE state can be performed trying a new SETPOS_CMD, or via the DISABLE_CMD / ENABLE_CMD sequence, or using the RESET_CMD and followed by a re-initialization.

The transition from OPERATIONAL to STANDBY/READY on the DISABLE_CMD is safe since the outputs are disabled via the ZeroOutputs action.

MAPPING MODELS TO PLATFORM SPECIFIC ARTEFACTS

The models described in the previous section can be used in several ways. They can be included in the design documentation, executed for simulations using an SCXML interpreter, or transformed into different types of artefacts to allow formal verification and facilitate SW development [11, 12].

We have used the models to develop PLC code for the TwinCAT 3 platform [13] and to build device simulators in Python for the Extremely Large Telescope (ELT) project.

Since there isn't an SCXML interpreter for PLCs yet, we have mapped (manually for the moment) the models into PLC code using a traditional state transition table. Events are mapped to constant integers and the same is done for the states. Actions are mapped to PLC functions while do-activities are transformed into code that is executed at each PLC cycle but only if the state containing the do-activity is active.

The simulators are built by linking together an SCXML interpreter and the implementation of the custom actions and do-activities. As SCXML interpreter we have used a Python library developed at ESO (scxml4py), but there are other libraries available for several programming languages.

The models can also be transformed with the COMODO tool into skeleton applications for one of the ESO existing SW platforms like the VLT SW, Alma Common Software, Java/RabbitMQ, and Java Pathfinder [9].

CONCLUSION

We have presented a library of models describing the behaviour of some devices included in the new instrument control application framework for the ELT project. The models are not bound to any specific implementation technology and therefore they could be reused by other organizations in different projects.

The models described in this paper can be found in the Open Model Based Engineering Environment (OpenMBEE) GitHub repository under the Comodo/Models directory [14]. Open-MBEE is an initiative to facilitate multi-tool and multi-repository integration across engineering, computing, and management disciplines [15].

We plan to extend our model repository by adding the models of tracking devices and supervisory processes with the goal creating a catalogue of behavioural patterns. At the same time, we are going to continue the investigation on how to extend the Statecharts notation to facilitate behavioural inheritance.

ACKNOWLEDGEMENT

We thank Gianluca Chiozzi (ESO) and Robert Karban (NASA/JPL) for their help and guidance.

REFERENCES

[1] D. Harel, “Statecharts: A visual formalism for complex systems”, Journal Science of Computer Programming, vol. 8, issue 3, pp. 231-274 (1987).

[2] D. Harel, “Statecharts in the Making: A Personal Account”, Communications of the ACM, 03/2009, Vol.52, No.03, p.6, (2009).

[3] S. Friedenthal *et al.*, “A Practical Guide to SysML: The Systems Modeling Language”, The OMG Press (2016)

[4] Semantic of A Foundational Subset for Executable UML Model (fUML), http://www.omg.org/spec/FUML

[5] Object Management Group, “Precise Semantics of UML State Machines”, http://www.omg.org/spec/PSSM

[6] OPC Foundation, “OPC Unified Architecture”, Part 5, Release 1.03, p.81-98, Annex B (normative) StateMachines B.1 (2015).

[7] W3C Reccomandation, “State Chart XML (SCXML): State Machine Notation for Control Abstraction”, https://www.w3.org/TR/scxml

[8] D. Harel, “On Visual Formalisms”, Communications of the ACM, 05/1988, Vol.31, No.05, pp.514-530, (1988).

[9] L. Andolfato, G. Chiozzi, N. Migliorini, C. Morales, “A Platform Independent Framework for Statecharts Code Generation”, Proc. ICALEPCS2011, pp. 614-617 (2011).

[10] G. Chiozzi, L. Andolfato, R. Karban, A. Tajeda, “A UML profile for code generation of component based distributed systems”, Proc. ICALEPCS2011, pp. 614-617 (2011).

[11] C. Gibson, R. Karban, L. Andolfato, J. Day, “Formal Validation of Fault Management Design Solutions”, ACM SIGSOFT Software Engineering Notes, Vol. 39 Issue 1, Jan 2014, pp. 1-5 (2014)

[12] L. Andolfato, R. Karban, M. Schiling, H. Sommer, M. Zamparelli, G. Chiozzi, “Experiences in Applying Model Driven Engineering to the Telescope and Instrument Control System Domain”, Proc. MODELS2014, LNCS 8767, Springer, pp. 403-419 (2014).

[13] Beckoff TwinCAT 3, http://www.beckhoff.de/twincat3/

[14] Open-MBEE GitHub repository, https://github.com/Open-MBEE/Comodo/tree/master/Models

[15] Open Model Based Engineering Environment (OpenMBEE), http://www.openmbee.org