Execution and Verification of UML State Machines with Erlang* (Tool Paper)

Ricardo J. Rodríguez, Lars-Åke Fredlund, Ángel Herranz and Julio Mariño

Universidad Politécnica de Madrid, Spain
{rjrodriguez, lfredlund, aherranz, jmarino}@fi.upm.es

Abstract. Validation of a system design enables to discover specification errors before it is implemented (or tested), thus hopefully reducing the development cost and time. The Unified Modelling Language (UML) is becoming widely accepted for the early specification and analysis of requirements for safety-critical systems, although a better balance between UML's undisputed flexibility, and a precise unambiguous semantics, is needed. In this paper we introduce UMerL, a tool that is capable of executing and formally verifying UML diagrams (namely, UML state machine, class and object diagrams) by means of a translation of its behavioural information into Erlang. The use of the tool is illustrated with an example in embedded software design.

1 Introduction

A better integration with the development process is crucial for the success of UML as a language for the specification and design of safety-critical software. This requires tools capable of validating complex requirements, and linking them to operational code through an unambiguous semantics. With these goals in mind we have developed UMerL, a tool that executes and verifies UML designs consisting of UML2 *state-machine*, *class* and *object* diagrams, using the concurrent language Erlang [1]. UMerL is available (with source code) at `https://bitbucket.org/fredlund1/umerl`.

UMerL differs from other analysis tools [2–4] in that an executable prototype is first produced, and then the validation of the UML model is done by performing various analyses on that prototype. Using Erlang has been crucial for implementing this approach. First, Erlang's concurrency features simplified the task of coding the state machine interpreter, which has to run several state machines with little runtime overhead. Also, the availability of advanced analysis tools for Erlang like Quviq QuickCheck [5] or the McErlang model checker [6] facilitates model validation without having to introduce a new set of tools.

Outline. Sec. 2 describes UMerL. Sec. 3 relates the application of UMerL to a real industrial case in embedded software design. We focus on the execution of

* The research has received funding from the ARTEMIS JU: grant agreement 295373 (nSafeCer), from the Spanish MINECO: STRONGSOFT (TIN2012-39391-C04-02), and from the Comunidad Autónoma de Madrid: PROMETIDOS (P2009/TIC-1465).

state-machine diagrams in Erlang and the verification of behavioural properties by program analysis on the Erlang code. Finally, Sec. 4 concludes the paper.

2 The UMerL tool

UMerL is an interpreter of UML state machines implemented in Erlang that executes a system modelled as a collection of UML state machines with acceptable performance. Its design also allows us to verify using model checking techniques whether a UML system meets some correctness properties. UMerL executes the UML state machines inside an object following a UML-friendly semantics: the meaning assigned to the constructs is consistent with respect to the UML (informal) semantics. An excerpt of the supported semantics and its interpretation by UMerL is described in Sec. 2.1.

System Description. In UMerL, a system description consists of a UML class diagram, a set of UML state-machine diagrams each one associated to a class and a UML object diagram. Each object has a private data store defined by the properties indicated in the class diagram. Several UML state-machine instances are running in each object, one for each UML state-machine associated to its class. To describe the system a domain specific language embedded in Erlang is used. The environment of the system can communicate with an object (and its associated state machines) by simply sending normal Erlang messages to the Erlang process associated with the object.

Verification Workflow. Figure 1 depicts the verification workflow of UMerL. A verification scenario consists of a system description and an environment model. The environment model sends messages and signals to the objects. We use Quviq QuickCheck [5] to randomly generate sequences of sensible messages. UMerL provides two functionalities: the system can be executed (with user interaction), providing early feedback regarding behaviour; or the verification scenario can be (model) checked against a set of correctness properties specified in Linear Temporal Logic (LTL) [7]. Properties are defined by the user using the McErlang tool [6], which provides a counterexample (in terms of message traces) when an LTL property is not satisfied by the system.

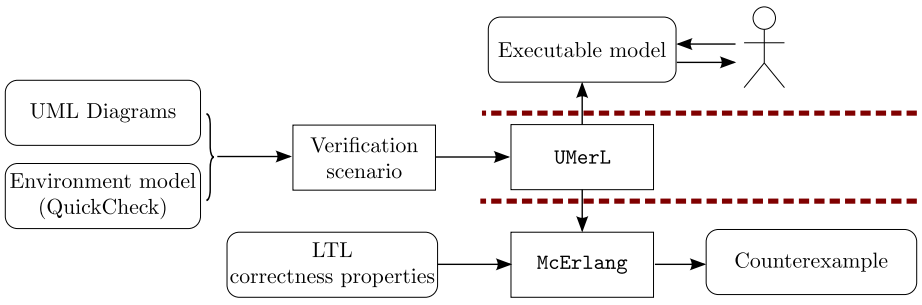


Fig. 1. Verification workflow of UMerL.

Architecture. UMerL maps each object to a single Erlang process: a generic interpreter that executes transitions of the UML state machines in that object, i.e., an individual UML state machine is not mapped to an Erlang process. Every step of the generic interpreter consists in choosing, non-deterministically, one of the enabled transitions and executing it (see Section 2.1).

A message sent to an object is received by its associated process, and is broadcast to every UML state machine running in the object. Conceptually, each state machine has its own mailbox for storing incoming messages until they are processed. Mailboxes are ordered in our implementation, i.e., if a message m_1 arrives before a message m_2, then m_1 will precede m_2 in the mailbox.

2.1 Semantics

Transition execution. A transition of a UML state machine can be executed when it is enabled: the mailbox contains a message that matches the trigger and the guard (a condition expressed over the contents of the message and the object data store) evaluates to true. The execution of a transition consists of three steps: (1) processing the first eligible message in the mailbox, (2) executing the activity that updates the private data store and sends messages to other objects, and (3) entering the target state. The execution of a transition appears to occur instantaneously since the execution of transitions follows a linearizable (atomic) semantics (atomicity does not necessarily mean mutually exclusive).

Do activities. A `do` activity is managed by an independent Erlang process, which is terminated when the execution of a transition leads outside the state.

Entry and exit activities. The semantics of an `entry` is implemented by adding it to the activity of every transition entering the state; an `exit` is implemented by adding it to the activity of every transition leaving the state.

Processing of messages. The most interesting aspect in our implementation is the processing of messages during the execution of a transition, and the deferral of messages. A message in the mailbox is eligible when it matches the trigger of a transition, and the corresponding guard evaluates to true. Our implementation chooses the oldest eligible message providing the additional guarantee (compared to the standard semantics for UML state machines) that messages from the same object are treated sequentially.

Deferral of messages. According to the UML semantics, when a message arrives in a state, there is no transition with a matching trigger or guard, and the deferral condition does not mention the trigger, the message should be discarded. This semantics still leaves room for interpretation in the implementation. As UMerL provides an ordering guarantee for messages (see above), it is possible to talk about the arrival order of messages. It is for instance clear that all non-deferrable messages that arrived before a message which causes a transition to be taken, are to be discarded. The doubt is which deferral annotation (the one at the source state, or the one at the target state) should affect the messages that arrived later than the message which caused the transition.

An *eager* semantics would discard all the messages not deferred by the source state while a *lazy* semantics would discard all the messages not deferred by the

target state. The implications are crucial. For instance, in a two step communication protocol where one machine sends two messages m_1 and m_2 in two consecutive transitions, and the other machine receives both messages, message m_2 could be discarded if the second machine does not defer it in the initial state (m_1 and m_2 could both be received when the machine is in that state).

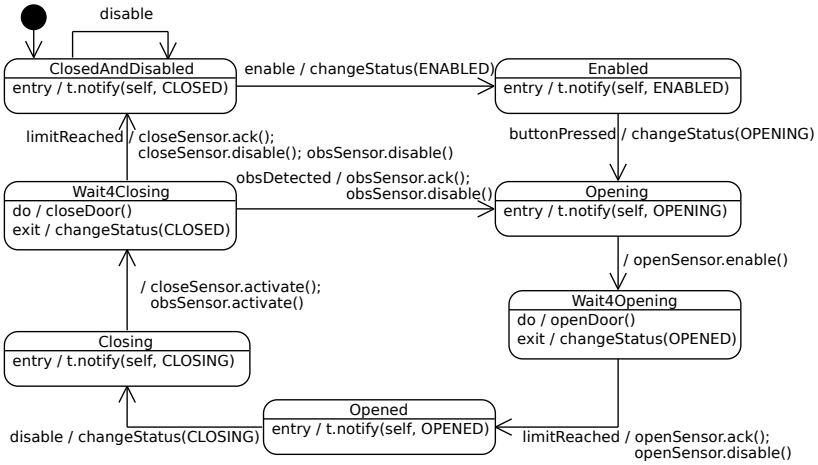
In our experience a designer of a distributed asynchronous system regularly makes mistakes causing messages to be silently discarded. We have decided to implement several semantic options to permit a designer to experiment with different interpretations of the discarding rule: (i) enable as the default that all messages are deferrable in states which has no explicit deferral condition. The use of such an option is, we argue, preferable when modelling distributed systems using state machines as it leads to fewer errors committed and less syntactic clutter (avoiding the need to repeat defer annotations in all states); and (ii) the choice of an eager or lazy deferral semantics, the default being lazy.

3 Safety Assessment of an Embedded Software Design

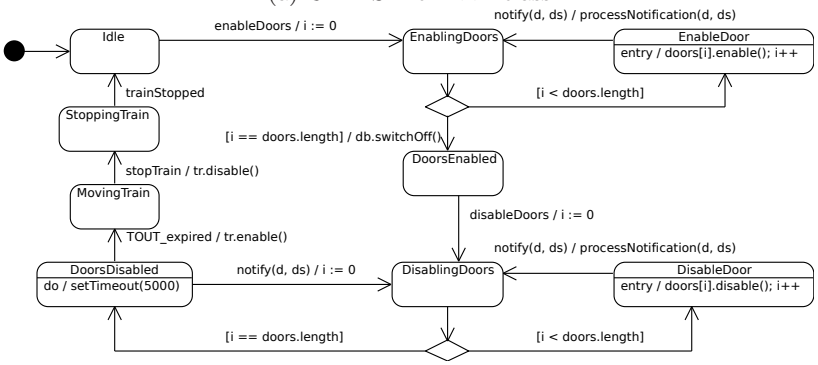
As case study, we consider a system for managing door operations in a train, provided by an industrial partner in a collaborative project. The system is composed of three major parts represented by UML classes: a Train Control Management System (TCMS), a traction system, and several doors. The traction system moves the train, or stops it. The doors allow passengers to enter or exit the coaches. Finally, the TCMS is an embedded device in charge of supervising both the traction system and the doors, to ensure safe operation.

Figure 2(a) and (b) show the UML-State Machine diagrams (UML-SMs) of the `Door` and `TCMS` classes, respectively. A `Door` object starts closed and disabled, and can be opened once it has been enabled (which is performed by the TCMS after receiving a `enableDoors` message), if a passenger presses the door button (triggering the sending of `buttonPressed` message to a door). Once a door is again disabled, it can be closed when no obstacle is detected. The `TCMS` starts in state *Idle*, which represents a state where the train is stopped. A message `enableDoors` will be eventually sent (by the train driver) and received by the TCMS. Then, the TCMS sends to each `Door` an `enable` message, and waits for an acknowledgement message (`notify`). The TCMS reports that all doors have been enabled by switching an informative LED off, and waits until the message `disableDoors` is received. After receiving it, the `TCMS` sends a `disable` message to each `Door`, and it waits a safety time interval of 5 seconds, before enabling the `Traction`, and moving to the *MovingTrain* state. This safety time is designed to permit to verify that all doors have been correctly closed. The transition from *MovingTrain* to *StoppingTrain* is triggered by the `stopTrain` message, and the TCMS acts by sending the `disable` message to `Traction`. The signal `trainStopped` is received once the train has been completely stopped, and so TCMS moves to the *Idle* state again.

Safety Assessment. A correctness property that must be assured in the system is that *No door should be open when the train is moving.* This is a safety property,



(a) UML-SM of `Door` class



(b) UML-SM of `TCMS` class

Fig. 2. UML-State Machine diagrams of (a) `Door` and (b) `TCMS` classes.

i.e., stating that *something bad never happens*. Note that the state predicate "a door is open" is true when the value of attribute `status` in a `Door` object is equal to `OPENED`. The predicate "the train is moving" is true when the value of attribute `speed` (in the `Traction` class, not shown here) is greater than zero. As a verification example, given an environment model in which an `enableDoors` message is first received by TCMS, and then a `buttonPressed` message is sent to a door, the McErlang model checker can verify that the two predicates above are never true in the same system state.

The above property can also be reformulated as *all doors must be closed when the train is moving*, i.e., the status attribute of a door must be `CLOSED`. Given that the `status` attribute of a door can be either `OPEN`, `CLOSED`, `ENABLED`, `OPENING` or `CLOSING`, this is a more restrictive (and safe) formulation. If we use the same environment model, except that we assume that a `disableDoors` message also arrives at the TCMS, the McErlang tool quickly finds a counterexample to the

second property. The counterexample indicates that the failure is that a door may not have had sufficient time to process earlier messages sent to it.

In brief, we have verified the above safety property for a coach with two doors in 0.35 seconds using McErlang (running under an Intel i7-2640M CPU with 8GB memory), with a resulting state space of 1,133 states, under the assumption that all messages are deferrable. The model of a coach with three (and four doors) has 7,323 (53,743 states), and its checking time is 1.74 seconds (14.88).

4 Conclusion

The experimental results obtained with our tool are quite promising. Although the tool works with only three kinds of UML diagrams and a quite restrictive syntax for state-machine diagrams (which has been crucial for defining the underlying semantics), the language is expressive enough to model embedded systems of a moderate complexity, e.g., the train doors example used here.

Several other tools exist which perform model checking on UML state machines. USMMC [2] is remarkable for being *self-contained*, it does not rely on a foreign formalism and checker, thus avoiding some inconveniences of translation-based tools like HUGO [4] or UMerL. However, UMerL does have a number of advantages too. First, an executable prototype, constructed from the UML model, provides early validation that can reveal mistakes even before attempting a detailed verification. Moreover, UMerL users can take advantage of existing analysis and testing tools for Erlang such as McErlang and QuickCheck.

To improve the usability of the tool a translation from the XMI notation is being implemented, and verification counterexamples will be presented as UML Sequence Diagrams. We also aim at specifying LTL properties at the UML level using a UML-friendly syntax, such as Object Constraint Language (OCL). Moreover, we aim at supporting hierarchical structures and pseudo-states among other UML features missing from the current prototype.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice-Hall (1996)
2. Liu, S., Liu, Y., Sun, J., Zheng, M., Wadhwa, B., Dong, J.S.: USMMC: A Self-contained Model Checker for UML State Machines. In: Proc. of the 2013 Meeting on Foundations of Software Engineering, New York, NY, USA, ACM (2013) 623–626
3. Lilius, J., Paltor, I.P.: Formalising UML State Machines for Model Checking. In: The Unified Modeling Language. Volume 1723 of LNCS. Springer (1999) 430–444
4. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive verification of UML state machines. In: Formal Methods and Software Engineering. Volume 3308 of LNCS. Springer (2004) 434–448
5. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing Telecoms Software with Quviq QuickCheck. In: ACM SIGPLAN Int. Erlang Workshop, ACM (2006)
6. Fredlund, L.Å., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: 12th ACM SIGPLAN ICFP, ACM (2007)
7. Pnueli, A.: The Temporal Logic of Programs. In: FOCS. (1977) 46–57