Statecharts Modelling and Simulation

Simon Van Mierlo Hans Vangheluwe

PRESENTERS

SIMON VAN MIERLO is a post-doctoral researcher at the University of Antwerp (Belgium). He is a member of the Modelling, Simulation and Design (MSDL) research lab. For his PhD thesis, he developed debugging techniques for modelling and simulation formalisms by explicitly modelling their executor's control flow using Statecharts. He is the main developer and maintainer of SCCD [1], a hybrid formalism that combines Statecharts with class diagrams. His e-mail address is Simon.VanMierlo@uantwerpen.be.

HANS VANGHELUWE is a Professor at the University of Antwerp (Belgium), an Adjunct Professor at McGill University (Canada) and an Adjunct Professor at the National University of Defense Technology (NUDT) in Changsha, China. He heads the Modelling, Simulation and Design (MSDL) research lab. In a variety of projects, often with industrial partners, he develops and applies the model-based theory and techniques of Multi-Paradigm Modelling (MPM). His current interests are in domain-specific modelling and simulation, including the development of graphical user interfaces for multiple platforms. To model such reactive systems, he advocates the use of Statecharts to describe their behaviour. His e-mail address is Hans.Vangheluwe@uantwerpen.be.

ABSTRACT

Statecharts, introduced by Harel [2], is used to specify complex, timed, reactive, autonomous discrete-event systems. It is an extension of Timed Finite State Automata which adds depth, orthogonality, broadcast communication and history. Its visual representation is based on higraphs, which combine graphs and Venn diagrams [3]. This representation is most suited to represent Statecharts models, and many tools offer visual editing and simulation support for the Statecharts formalism. Examples include STATEMATE [4], Rhapsody [5], Yakindu [1], and Stateflow [2].

This tutorial introduces Statecharts modelling, simulation, and testing. As a running example, the behaviour of a simple timed, autonomous, reactive system is modelled: a traffic light. We start from the basic concepts of states and transitions and explain the more advanced concepts of Statecharts by extending the example incrementally. We discuss several semantics options, such as STATEMATE and Rhapsody semantics. We use Yakindu to model the example system.

[1]https://www.itemis.com/en/yakindu/statechart-tools/

[2]https://www.mathworks.com/products/stateflow.html

KEYWORDS

Statecharts, reactive systems, testing, Yakindu

LENGTH

The proposed length of the tutorial is three hours.

LEVEL OF THE TUTORIAL

Introductory. No knowledge of Statecharts or the tool (Yakindu) is necessary.

TARGET AUDIENCE

Modellers with an interest in specifying the behaviour of reactive systems using Statecharts.

NOVELTY

This tutorial has previously been given (in 2017) at the Spring Simulation Multi-Conference (SpringSim)[3]. Due to the success of this tutorial, a second edition of the tutorial is scheduled for the 2018 edition of this conference. Since this conference is mainly targeted at an audience with a simulation background, the tutorial focuses on the introduction of Statecharts as a modelling formalism for specifying reactive systems, and covers the basics of Statecharts. For the audience at the MoDELS conference, we will discuss Statecharts more extensively and in-depth, focusing on the applicability to more complex systems than the running example.

DESCRIPTION OF THE TUTORIAL AND INTENDED OUTLINE

The tutorial starts by explaining the causes of complexity in reactive systems, and explains why Statecharts is an appropriate formalism to model their behaviour. A workflow for specifying, simulating, testing, and deploying Statecharts models is presented, and a running example is used to discuss the features of the Statecharts language and the Yakindu modelling and simulation tool. As a last step, the system is deployed by generating code from the model.

A more in-depth discussion of the steps presented during the tutorial is provided below.

- Explanation of the source of (essential) complexity in engineered systems, which often can be attributed to them having timed, reactive, autonomous behaviour. A number of such systems are presented to demonstrate this complexity. To effectively develop these systems, traditional approaches based on threads and timeouts add accidental complexity. These approaches focus on "how" the system's behaviour is implemented, instead of "what" the system is supposed to do. Instead, a language which

[3]http://scs.org/springsim

has notions of *events*, *timeouts* and *concurrent units* is needed.

- Introduction of the notion of discrete-event abstraction, in which a system's autonomous behaviour can be interrupted by external events coming from the environment, and the system can produce output to that environment.
- Introduction of Statecharts as an appropriate formalism to model a reactive system's behaviour using a discrete-event abstraction.
- Introduction of the running example of the tutorial: a traffic light. This example is basic, but has all essential complexity: it is timed, its lights have to switch autonomously, and it is reactive, since its normal execution can be interrupted by a policeman.
- Throughout the tutorial, a workflow for designing, testing, and deploying systems using Statecharts is presented:

    1) First, a set of requirements are gathered: these are properties the system's behaviour needs to satisfy, and are typically described in text.
    2) Then, an initial design is created as a Statecharts model. The model implements (part of) the system's specification.
    3) The model needs to be verified (i.e., we need to check whether its behavioural properties are satisfied). To do this, the model can be *simulated* (in which the user defines a simulation scenario and checks the outcome of the simulation manually) or it can be *tested* (in which the user defines a number of test cases, which consist of a set of timed inputs that are supplied to the model, and an automatic checker or "oracle" that verifies whether the test succeeds). If a simulation or test results in a *failure* (i.e., one of the system's properties is not satisfied), the system's model needs to be revised.
    4) When the system's design has gone through several simulation and testing phases, and its behaviour is properly verified, code can be generated to deploy it.

- With the workflow in mind, an explanation follows of the basic building blocks of Statecharts: *states* and *transitions* (which are triggered by *events* and optionally have a constraint that needs to be satisfied).
- Progressively introduce new elements of the Statecharts language: hierarchy, orthogonality, and history. For each concept, both the syntax and the semantics of the concept is explained and applied to the example. And, a demonstration of each concept (including simulation to demonstrate semantics) in the Yakindu modelling and simulation tool.
- Demonstration of the testing of Statecharts models. Since a test consists of an input event trace, and an "oracle" which checks the output event trace, a test can be seen as a timed, reactive system as well. The tests in this tutorial, therefore, are modelled with Statecharts as well. They can either be modelled as three communicating Statecharts (the input Statechart, the model under test, and the oracle), or in a single Statechart model in orthogonal components.
- Demonstration of code generation (to Java). For this, the code generation capabilities of the Yakindu tool are used. A generic visualization for the traffic light example is built, with which the generated code from the behavioural model communicates through the interface of the model (input/output events).

Required Infrastructure

A data projector is required.

Sample Slides

A number sample slides of the previous versions of the tutorial are attached.

References

[1] S. Van Mierlo, Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe, "SCCD: SCXML extended with class diagrams," in *3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016*, 2016.

[2] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[3] ——, "On visual formalisms," *Commun. ACM*, vol. 31, no. 5, pp. 514–530, May 1988. [Online]. Available: http://doi.acm.org/10.1145/42411.42414

[4] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 293–333, oct 1996. [Online]. Available: http://doi.acm.org/10.1145/235321.235322

[5] D. Harel and H. Kugler, *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ch. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML), pp. 325–354.