Supporting the evolution of behavioural software models – A Research Vision

Tom Mens
Software Engineering Lab, COMPLEXYS Research Institute
University of Mons, Belgium
tom.mens@umons.ac.be

Abstract

This position paper starts by presenting the important research challenges related to model-driven software evolution. We revisit those challenges in the context of executable models of behavioural design (such as statecharts), and outline the research goals we aim to address in this domain.

1 Research Challenges

Model-driven software engineering (MDE) [SV06, BCW12] relies on the systematic use of models as primary artifacts throughout the software development lifecycle. Software models aim to reduce the accidental complexity introduced by technical details related to the chosen hardware and software platform. Unless if software models are considered as one-shot throw-away artefacts that are used only for designing the first version of a system, it becomes incumbent to support their evolution over time, together with the systems they describe.

The challenges raised by this need to evolve software models have been identified repeatedly in the past. For example, a 2005 working group report identified **supporting model evolution** as one of the main challenges in software evolution research. A 2008 workshop focused on specific challenges in MDE [VMV08], and the need to support evolution in some form emerged repeatedly. Let us summarise the evolution-related challenges that have been identified in both reports:

- The need to *support model quality*. This includes defining, measuring, preserving, controlling, and improving model quality, and also encompasses approaches like *model refactoring* and *model testing*.
- The need to *support co-evolution* in all its incarnations. This includes *traceability* between requirements and design models, *synchronisation* between design models and code, *consistency management* between different types of models, and coupled evolution between models and metamodels.
- The need to cope with *dynamic model evolution*, by providing techniques to cope with the evolution of executable models *at runtime*.
- Providing *formal support* for evolution. While *formal verification* techniques allow to verify interesting properties over models, these techniques should be made more *lightweight* to facilitate their take-up by software designers with little mathematical background. In addition, these techniques need to be made more *incremental*, taking into account the fact that models evolve over time, and avoiding the need to reverify the full model even if only small changes have been made to it.

Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE), Bergen, Norway, 11-13 July 2016

- Supporting *heterogeneous, multi-formalism or multi-paradigm modelling*, in which systems are modelled using different formalisms and modelling languages, always choosing the most suitable formalism at the most appropriate level of abstraction.
- Supporting *fuzzy modelling*, taking into account the fact that certain models may be imprecise, incomplete, partially inconsistent or ambiguous.
- *Integrating change as a first-class concept* in modelling languages, model-driven software development environments and processes.
- Providing better support for *change management* in MDE. This includes better "model-aware" mechanisms for versioning, differencing and merging, taking into account the specificities (syntax and semantics) of the modelling language.
- Provide generic support for all of the above, so that it can be integrated easily in *domain-specific modelling*.
- Addressing the *scalability* problem, to cope with very large models, or collections of interacting models.

In our research, we aim to address many of these challenges, focusing on a specific kind of software models that require particular attention, namely *executable models of software behaviour*. Such models have the advantage of being able to represent and simulate a system's behaviour, and to generate executable code for it.

2 Statecharts

Our research focus will initially go to *statechart*-based design models, but could be extended to other types of behavioural models in a second phase. We have chosen to focus on *statecharts* because of their relative complexity and the lack of advanced change support for these types of models.

Statecharts were introduced nearly three decades ago by David Harel [Har88, HG97] as a formal and visual executable modelling language. Being now part of the UML standard, they are frequently used in industry for modeling the executable behaviour of complex reactive event-based systems (e.g., real-time systems and embedded systems), relying on commercial tools such as *IBM Rational StateMate* [HN96], *IBM Rational Rhapsody* [HK04], *Mathworks Stateflow*, *Yakindu Statechart Tools*, and many more.

UML deliberately leaves the statechart semantics underspecified for certain aspects, enforcing tool developers to make their own choice (especially w.r.t. how to ensure deterministic behaviour). Many semantic variations can therefore be found in commercial statechart tools, and detailed comparisons have been reported in literature [vdB94, Esh09, EDAN10]. Such semantic differences may lead to misunderstandings or conceptual errors for statechart designers.

We have implemented SISMIC, a research prototype for interpreting and reasoning about executable statecharts. It offers a discrete, step-by-step, and fully observable simulation engine, supporting the majority of the UML 2 statechart concepts and semantics. The tool is provided as an open source library[1] in Python 3 that can be installed through the Python Package Index[2]. We aim to use this tool as a research platform for exploring the evolution-related research goals that will be presented in Section 3

3 Research goals

In the remainder of this paper, for the sake of generality, we will use the term *behavioural model* instead of *statechart*. Our medium and long-term research goal, subject to our ability to obtain funding for this research, is to provide formal support and tool support that allows us to advance the state-of-the-art along the following directions:

Advanced model testing: To increase their reliability, design errors should be detected in behavioural models as early as possible. Examples of support include test-driven [Bec03], behaviour-driven [WH12] and contract-driven development [Mey07], and advanced testing techniques [Ber07] such as mutation testing [JH11] and concolic testing [Sen07]. We propose to explore and combine these complementary techniques at the level of behavioural models. We will provide mechanisms based on test-driven development, in which unit tests can be expressed by means of behavioural models themselves. We will apply the idea of behaviour-driven

[1] `github.com/AlexandreDecan/sismic`
[2] `https://pypi.python.org/pypi/sismic`

development at the level of behavioural models. We will also use design by contract at the level of behavioural models, since this technique has been shown to detect certain types of programming errors and bugs more effectively [PFN+14]. We propose to rely on constraint solvers and machine learning to generate new contracts automatically, by detecting invariants during statechart execution based on the ideas of [EPG+07]. We also aim to generate new tests from contract specifications, and to automate detection and resolution of inconsistencies or removing redundancies in contract specifications.

Lightweight formal verification: Formal verification complements model testing since it allows to find conceptual design errors early (and more reliably than with conventional testing techniques). In order to be usable by and useful to software designers with little mathematical background, we propose to use *lightweight* techniques [JW96] for verifying properties and ensuring constraints over behavioural models, using both static and dynamic verification approaches. For example, the use of specification patterns [DAC99] or domain-specific property languages [MDL+14] allows to specify formal properties in a language that is easier to understand or closer to the domain expert. Because the statechart formalism is undecidable in general, one needs to restrict to decidable fragments over which useful properties can be expressed with an appropriate automata-based formalism. We will explore which type of formalism can express which type of property in the most expressive and efficient way. We will also seek solutions to the well-known state space explosion problem that goes hand in hand with the use of model checking techniques.

Model quality assessment: Analysing and improving model quality is crucial in any MDE setting. Individual solutions have been proposed for measuring model quality, detecting model smells and improving quality through behaviour preserving model refactorings. These solutions need to be made more integrated, more scalable and more generic. We will also explore the idea of *refactoring by example*, based on genetic algorithms and heuristic search [GEBK14], at the level of behavioural models.

Semantic variation: Given the presence of different semantic variants of statecharts, designers should be informed about, and supported to cope with, the consequences of these differences. We propose to achieve this by providing generic tool support to make their designs more robust to semantic variations.

Scalable modelling: Complex designs require the need to deal with multiple interacting behavioural models. We therefore need to provide proper composition and interaction mechanisms allowing to enable such scalable designs. We propose to achieve this by drawing inspiration from component-based software engineering research. The proposed mechanisms will require the integration of appropriate solutions for the aforementioned challenges of quality assessment, automated testing and formal verification.

Design space exploration: To select the most appropriate design model for a given problem, techniques are needed to explore and compare these alternatives (e.g., in terms of desirable properties or quality characteristics) and to select the most appropriate design alternative. We propose to explore techniques based on mutation analysis and genetic algorithms to "evolve" existing designs to ones with a better fitness for purpose.

Variability analysis of model families. Software product line engineering [WL99] aims to support designing families of software products as opposed to individual software products, thereby maximising reuse, increasing product quality and reducing development effort. This is achieved by expressing and leveraging the commonalities and variabilities between the different members of the product family. We propose to use these techniques to express, reason about, and facilitate the evolution of families of behavioural models.

Semantic model evolution: We propose a wide range of techniques to cope with the evolution of behavioural models. An operation-based model representation [**?**] will be provided and combined with an appropriate model versioning approach to facilitate reasoning over the model evolution history. We will provide support for semantic model differencing [MRR11] to detect the semantic impact of changes to behavioural models, and model merging to integrate parallel changes. Finally, we will study support for incremental model verification, to reverify only those model properties that are potentially impacted by a model change.

References

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan and Claypool, 2012.

[Bec03] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103. IEEE Computer Society, 2007.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Int'l Conf. Software Engineering*, pages 411–420. ACM , 1999.

[EDAN10] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering*, 15(2):235–265, 2010.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[Esh09] Rik Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65 – 99, 2009.

[GEBK14] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. Model refactoring using examples: a search-based approach. *J. Software: Evolution and Process*, 26(7):692–713, 2014.

[Har88] David Harel. On visual formalisms. *Comm. ACM*, 31(5):514–530, 1988.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[HK04] David Harel and Hillel Kugler. *Integration of Software Specification Techniques for Applications in Engineering*, chapter The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML), pages 325–354. Springer, 2004.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[JH11] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Soft. Eng.*, 37(5):649–678, Sept 2011.

[JW96] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A framework for generating domain-specific property languages. In *Int'l Conf. Software Language Engineering (SLE)*, volume 8706 of *Lect. Notes in Computer Science*, pages 1–20. Springer, 2014.

[Mey07] Bertrand Meyer. Contract-driven development. In *Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *Lect. Notes in Computer Science*, page 11. Springer, 2007.

[MRR11] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. *Models in Software Engineering: Workshops and Symposia at MODELS 2010 - Reports and Revised Selected Papers*, chapter A Manifesto for Semantic Model Differencing, pages 194–203. Springer, 2011.

[PFN+14] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Soft. Eng.*, 40(5):427–449, 2014.

[Sen07] Koushik Sen. Concolic testing. In *Int'l Conf. Automated Software Engineering*, pages 571–572. ACM, 2007.

[SV06] Thomas Stahl and Markus Völter. *Model Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[vdB94] Michael von der Beeck. Int'l symp. formal techniques in real-time and fault-tolerant systems. In Hans Langmaack, Willem-Paul Roever, and Jan Vytopil, editors, *A comparison of Statecharts variants*, pages 128–148. Springer, 1994.

[VMV08] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in model-driven software engineering. In Michel Chaudron, editor, *Workshops and Symposia at MoDELS 2008*, volume 5421 of *Lect. Notes in Computer Science*. Springer, 2008.

[WH12] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[WL99] D. M. Weiss and R. Lai, editors. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.