Smyle: A Tool for Synthesizing Distributed Models from Scenarios by Learning*

Benedikt Bollig[1], Joost-Pieter Katoen[2], Carsten Kern[2], and Martin Leucker[3]

[1] LSV, ENS Cachan, CNRS, [2] RWTH Aachen University, [3] TU München

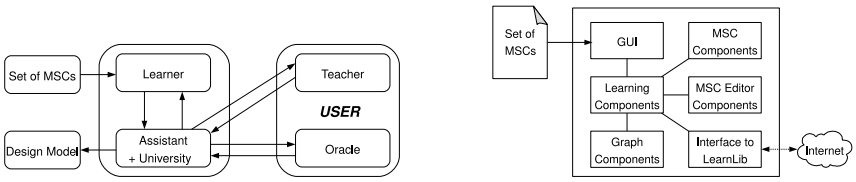
1 Overview

This paper presents *Smyle*, a tool for synthesizing asynchronous and distributed implementation models from sets of scenarios that are given as message sequence charts (MSCs). The latter specify desired or unwanted behavior of the system to be. Provided with such positive and negative example scenarios, *Smyle* employs dedicated learning techniques and *propositional dynamic logic* (PDL) over MSCs to generate a system model that conforms with the given examples.

Synthesizing distributed systems from user-specified scenarios is becoming increasingly en vogue [7]. There exists a wide range of approaches for synthesizing implementation models from a priori given scenarios [6, 11, 8, 5, 14, 15]. The approaches mainly differ in their specification language, the inference procedure, and the final implementation model. Several of them employ MSCs as specification language because they are standardized by the ITU Z.120 [9] and adopted by the UML as sequence diagrams. Other approaches try to utilize more expressive notations like triggered MSCs [14], high-level MSCs [6], or live sequence charts [8]. On the one hand, more expressive power results in richer specifications. On the other hand, however, it is just this great expressiveness that disqualifies them for non-professional or a fortiori unexperienced users, which are overstrained by these formalisms. As requirements specifications over and over demonstrate, human beings strongly prefer to express scenarios in terms of simple pictures including the acting entities and their interaction. Due to this reason, we will restrict to so-called basic MSCs, only, which consist of processes (i.e., vertical axes denoting evolution of time) and messages (i.e., horizontal or slanted arrows between processes signifying asynchronous information exchange).

Many approaches to synthesizing distributed systems typically model synchronous communication, infer labeled transition systems, and use standard automata-theoretic solutions to project the global system onto its local components. As can be shown easily, this results in missing or implied behavior that was not stipulated by the user. In contrast, we regard asynchronous communication behavior and derive a distributed model in terms of a *message passing automaton* (MPA), which models the asynchronous communication in a natural manner deploying FIFO channels. It consist of one local automaton for every process involved in the system. Harel in his recent article [7] states that it is an intrinsically difficult task to "[...] distribute the intuitive played-in behavior [...]" and it is still a *dream* to employ *scenario-based programming*. Nevertheless we try to converge to this vision using the tool *Smyle* presented in this paper.

* This work is partially supported by the DAAD (Procope 2008).



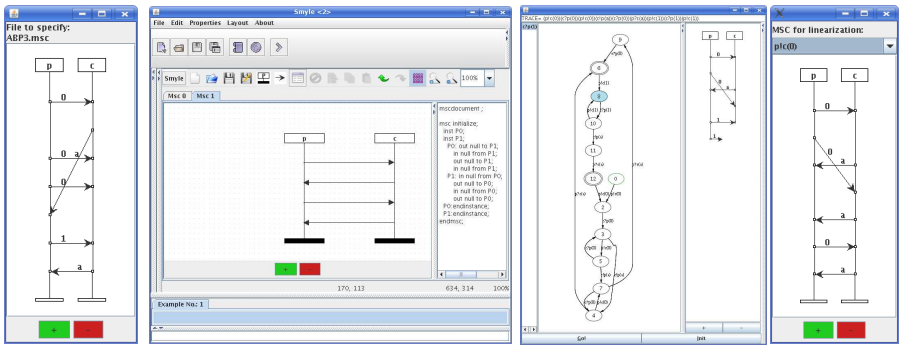
(a) Learning overview (b) Package overview

Fig. 1. *Smyle*'s architecture

2 *Smyle* from a user perspective

Smyle is an acronym for *Synthesizing Models bY Learning from Examples*. Its major objective is to ease the development of concurrent systems. More specifically, the overall goal is to derive communication models of concurrent systems. The synthesis process starts by providing the tool with a set of sample MSCs where each MSC is either positive or negative. Positive MSCs describe system behavior that is *possible* and negative MSCs characterize *unwanted* or *forbidden* behavior. *Smyle* focuses on basic-MSC features like asynchronous message exchange and forbids to deploy the complete MSC standard—which allows for alternation, loops etc.—on purpose: the more expressive a specification language gets the less intuitive and manageable it becomes. Simple pictures however are easy to understand and easy to supply. As mentioned in the previous section, human beings prefer to describe system runs by means of simple examples and pictures. Basic MSCs constitute such a device. More information about the formal basis *Smyle* builds on can be found in [3].

The learning chain: In order to initiate the synthesis process, a so-called *learning setup* is specified where the channel capacities of the final system are fixed a priori by a bound $B \in \mathbb{N}$. The user can choose from two variants: she may either want to *learn* a *universally-B-bounded* system, which means that the outcome will be realizable using finite channel capacity B, thus resulting in a finite-state system, or to infer a possibly infinite-state system by requiring existential bounds on the system's channels. An *existentially-B-bounded* learning setup allows the system developer to include system behavior that may exceed the system's channel bound B but at the same time guarantees that there is at least one execution (i.e., a total ordering of events or a *linearization*) that adheres to this limit for each scenario recognized by the final system. Hence, an appropriate scheduler will always be able to execute the *good* linearizations (i.e., runs not exceeding B) and disregard the ones going beyond the bound. Having chosen a learning setup, a set of MSCs has to be provided as initial input to the tool. *Smyle* will ask the user to classify these MSCs and start the learning procedure (cf. Figure 1(a)). Successively, new MSCs as depicted in Figure 2(a) are presented to the user (acting as Teacher) who in turn has to classify these scenarios as either wanted (positive) or illegal (negative). Whenever *Smyle* has a complete and consistent view of the current internal system, it presents a window (cf. Figure 2(c)) for testing and simulating the derived system. Within this component, the user (now acting as Oracle) may execute actions to see how the system behaves. These actions are monitored and the related scenario is de-



(a) MSC (b) Integrated MSC editor with MSC to classify (c) Simulation component (d) MSC rejected by φ

Fig. 2. *Smyle* GUI

picted as MSC on the right hand side of the frame (cf. Figure 2(c)). If, after an intensive simulation, there is no evidence for wrong or missing behavior, the user will terminate the simulation session and the concurrent system is deduced. If, however, some illegal or missing behavior is detected, then the user can use the corresponding MSC as counterexample, or, respectively, edit the missing scenario to a (positive) counterexample. This singleton set of counterexamples may of course be enriched by additional MSCs, and the learning procedure continues as explained until reaching the next consistent model. An exemplifying video of this learning process can be downloaded from the tool's webpage [1].

The MSC editor: When new MSCs have to be specified in order to start or continue the learning phase, *Smyle* can either load MSC documents containing basic MSCs from the file system or offer to use the integrated MSC editor (cf. Figure 2(b)) for easy specification of basic MSCs. The MSCs can directly be classified and fed back to *Smyle* in order to derive a new MPA. The editor also provides functionality for storing MSCs in many different formats (e.g., LaTeX, fig, etc.). An extended, stand-alone version of the editor covering the ITU Z.120 standard to a large extend will soon be available [1].

Easing the learning process: In order to simplify the user's task of classifying scenarios, *Smyle* contains means for specifying formulas from PDL over MSCs, a simple logic that comes with an efficiently solvable membership problem [4]. Like MSCs, PDL formulas are used to express desired or illegal behavior, but in a broader sense. They are to be seen as general rules which apply for *all* runs of the system (and not only all executions of one scenario). Hence, if a user detects generally wanted or unwanted properties of the presented MSCs she may specify formulas which express these *generics*. *Smyle* is supplied with these formulas and can, from that moment on, accept or reject all MSCs that fulfill or, respectively, violate one of these formulas. This technique reduces the number of user queries substantially. An example formula is $\varphi = \mathbf{A}\big([\, p?c(\mathtt{a}); \mathsf{proc}; p?c(\mathtt{a})\,]\ \mathbf{false}\big)$ which states that there must not be two subsequent occurrences of the same action (i.e., $p?c(\mathtt{a})$) on the same process p. Hence, if

formula φ is fed to *Smyle* as negative generic, all MSCs featuring this behavior, e.g., the one in Figure 2 (d), would be regarded as negative samples without questioning the user.

3 *Smyle*'s implementation details

Smyle is a platform-independent application written in `Java 1.5`. For visualization purposes, e.g., displaying MSCs and the implementation model, it uses the graph-visualization libraries `Grappa`[1]and `JGraph`[2], and employs the MSC2000 parser [12] for parsing the input MSC documents. As depicted in Figure 1(b), *Smyle* consists of six main packages: the graphical user interface (GUI), one package for MSC components, one for learning components, one comprising the MSC editor functionality, one for graph components, and an interface to the learning library *LearnLib* [13]. The functionality of these components is briefly described in the following.

MSC components: This package contains the MSC2000 [12] parser for handling MSC documents according to the ITU Z.120 standard [9]. It provides the classes for representing the internal MSC objects.

Learning components: The tasks of the learning component are manifold. It contains important functionality for efficient partial-order treatment, harbors the simulator which can be applied to the learned model, and comprises the $Learner$, the $Assistant$, and the $University$, which acts as mediator between the components of this package and the other packages as shown in Figure 1(a).

MSC editor components: The MSC editor components feature the implementation of an integrated MSC editor, which is able to load, store, and alter basic MSCs. Moreover, the created MSCs can be exported to LaTeX and the `fig` format and thus can be converted, using available tools, to all other prevalent graphical formats (e.g., eps, pdf, jpeg).

Graph components: The graph components package includes functionality for checking MSC behavior (e.g., the FIFO property) and the consistency of the implementation models.

Interface to LearnLib*:* This package includes an interface to a learning library called *LearnLib* [13], which implements Angluin's algorithm L^* [2]. This interface is by courtesy of the *Fachbereich Informatik, Lehrstuhl 5* (University of Dortmund).

4 Future work

We have applied *Smyle* to a number of examples. We inferred, for example, a model for the ABP, one for a part of the USB 1.1 protocol and one for a leader election protocol for a unidirectional ring. Moreover, *Smyle* has also been considered in [10] where scenarios represented as MSCs are derived from natural language specifications.

[1] Grappa: http://www.research.att.com/~john/Grappa/
[2] JGraph: http://www.jgraph.com/

For future work, we plan to extend the formula component and integrate it into the MSC editor to be able to derive PDL formulas from visually annotated MSCs. Moreover, we intend to apply *Smyle* to larger sized case studies to evaluate its feasibility for real world problems.

The synthesis tool *Smyle*, the MSC editor as well as dedicated theoretic background information and an exemplifying video presenting the learning chain can be freely downloaded for educational and research purposes from:

`http://www.smyle-tool.org/`

References

1. Smyle *webpage*. `http://www.smyle-tool.org/`.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
3. B. Bollig, J.-P. Katoen, K. Carsten, and M. Leucker. Replaying play in and play out: Synthesis of design models from scenarios by learning. In *TACAS*, volume 4424 of *LNCS*, pages 435–450. Springer, 2007.
4. B. Bollig, D. Kuske, and I. Meinecke. Propositional dynamic logic for message-passing systems. In *FSTTCS*, volume 4855 of *LNCS*, pages 303–315. Springer, 2007.
5. C. Damas, B. Lambeau, and P. Dupont. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Softw. Eng.*, 31(12):1056–1073, 2005.
6. B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level mscs: model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006.
7. D. Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, 2008.
8. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
9. ITU-TS Recommendation Z.120: Message Sequence Chart 1999 (MSC99), 1999.
10. L. Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In *15th IEEE RE*, pages 121 – 130, 2007.
11. E. Mäkinen and T. Systä. MAS – An interactive synthesizer to support behavioral modeling in UML. In *ICSE*, pages 15–24. IEEE Computer Society, 2001.
12. H. Neukirchen. *MSC2000 Parser*. `neukirchen@informatik.uni-goettingen.de`.
13. H. Raffelt and B. Steffen. LearnLib: A library for automata learning and experimentation. In *FASE*, volume 3922 of *LNCS*, pages 377–380. Springer, 2006.
14. B. Sengupta and R. Cleaveland. Triggered message sequence charts. *IEEE Trans. Softw. Eng.*, 32(8):587–607, 2006.
15. S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE*, pages 34–43. IEEE Computer Society, 2007.