Computing and Informatics, Vol. 29, 2010, 45–72

SMA – THE SMYLE MODELING APPROACH

Benedikt Bollig

Laboratoire Spécification et Vérification
CNRS UMR 8643
École Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 CACHAN Cedex, France
e-mail: bollig@lsv.ens-cachan.fr

Joost-Pieter Katoen, Carsten Kern

Lehrstuhl für Informatik 2
RWTH Aachen University Ahornstrasse 55
52074 Aachen, Germany
e-mail: {katoen, kern}@cs.rwth-aachen.de

Martin Leucker

Lehrstuhl für Informatik 4
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
e-mail: leucker@in.tum.de

Revised manuscript received 16 October 2009

Abstract. This paper introduces the model-based software development lifecycle model *SMA* – the *Smyle Modeling Approach* – which is centered around *Smyle*. *Smyle* is a dedicated learning procedure to support engineers to interactively obtain design models from requirements, characterized as either being desired (positive) or unwanted (negative) system behavior. Within *SMA*, the learning approach is complemented by so-called *scenario patterns* where the engineer can specify *clearly*

desired or unwanted behavior. This way, user interaction is reduced to the interesting scenarios limiting the design effort considerably. In *SMA*, the learning phase is further complemented by an effective analysis phase that allows for detecting design flaws at an early design stage. Using learning techniques allows us to gradually develop and refine requirements, naturally supporting evolving requirements, and allows for a rather inexpensive redesign in case anomalous system behavior is detected during analysis, testing, or maintenance. This paper describes the approach and reports on first practical experiences.

Keywords: Requirements elicitation, design model, learning, software engineering lifecycle, Message Sequence Charts, UML

Mathematics Subject Classification 2000: 68N30, 68Q85, 68Q32

1 INTRODUCTION

To put it bluntly, software engineering – under the assumption that a requirements acquisition has taken place – amounts to bridging the gap between requirements, typically stated in natural language, and a piece of software. To ease this step, in model-driven design (like MDA), architecture-independent *design models* are introduced as intermediary between requirement specifications and concrete implementations. These design models typically describe the control flow, basic modules or components, and their interaction. Design models are then refined towards executable code typically using several design steps where system details are incorporated progressively. Correctness of these design steps may be checked using e.g., model checking or deductive techniques.

Problem statement. Typically, an abundant number of requirements are formulated manually, using natural language or semi-formal notations – with the typical implication that requirements are ambiguous and contradictory. Moreover, requirements typically change over time, meaning at all stages of the development process, let be due to changing user requirements or to anomalous system behavior detected at a later design stage. As one consequence, also the design model may not reflect the requirements correctly. Standard software engineering lifecycle models are, unfortunately, not designed to support evolving requirements.

Contribution. This paper presents the *Smyle Modeling Approach* (SMA) as *novel* software engineering lifecycle model, which is based on a new approach towards requirement specification and high-level design. It is tailored to the development of communicating distributed systems whose behavior can be specified using sequence diagrams exemplifying either desired or undesired system runs. A widespread notation for sequence diagrams is that of message sequence charts (MSCs). They have

been adopted by the UML, are standardized by the International Telecommunication Union (ITU) [26], and are part of several requirements elicitation techniques such as CREWS [30].

At the heart of SMA a dedicated *learning technique* supports the engineer to interactively obtain implementation-independent design models from MSCs exemplifying the system's behavior. These techniques are implemented in the *Smyle* tool (Synthesizing Models bY Learning from Examples, cf. [8]). The incremental learning approach allows to gradually develop, refine, and complete requirements, and supports evolving requirements in a natural manner, rather than requiring a full-fledged set of requirements up front. Importantly, *Smyle* does not only rely on given system behaviors but progressively asks the engineer to classify certain corner cases as either desired or undesired behavior, whenever the so-far provided examples do not allow to determine a (minimal) system model in a unique manner.

As abstract design models, *Smyle* synthesizes distributed finite-state automata (referred to as communicating finite-state machines [12], or CFMs for short). This model is implementation-independent and describes the local control flow as finite automata which communicate via unbounded order-preserving channels. Thus, the behavior of these models can directly be represented as sets of MSCs.

SMA is a software engineering lifecycle model centered around *Smyle*. The learning approach is complemented by so-called *scenario patterns* where the engineer can specify *clearly* desired or unwanted behavior graphically or via a dedicated formula editor. This way, user interaction is reduced to the interesting scenarios limiting the design effort considerably. Once an initial high-level design has been obtained by learning, *SMA* asks for intensive simulation of the obtained model and for checking elementary correctness properties of the CFM, for example by means of model checking or dedicated analysis algorithms [9]. This allows for an early detection of design flaws. In such a case, i.e., some observed behavior should be ruled out or some expected behavior cannot be realized by the current model, the learning phase can be continued with the corresponding scenarios yielding an adapted design model now reflecting the expected behavior for the given scenarios.

A satisfactory high-level design may subsequently be refined or translated into, e.g., Stateflow diagrams [21] from which executable code is automatically generated using tools such as Matlab/Simulink. Alternatively, code skeletons may be generated that perform the desired communication expressed by the design model, but which may be enriched by concrete computations of values, memory management etc.

The final stage of *SMA* is a model-based testing phase [13] in which it is checked whether the software conforms to the high-level design description. The MSCs used for formalizing requirements now serve additionally as abstract test cases. Supplementary test cases are generated in an automated way. This systematic on-the-fly test procedure is supported by tools such as TorX and TGV [5] that can easily be plugged in into our design cycle. Again, any test failure that results from an invalid design model can be described by MSCs which may be fed back to the learning phase.

Related work. To our best knowledge there is no related work on defining lifecycle models based on learning techniques. However, several approaches for synthesizing models based on scenarios are known [36, 16, 23, 11]. One of the first attempts to exploit learning for interactively synthesizing models was proposed in [29] where for each process in the system an automaton is inferred using Angluin's learning technique [3]. The drawback of this approach is that putting the resulting automata in parallel yields a system that may have previously undesired behavior and also may easily deadlock.

In Damas et al. [15], positive and negative scenarios are used for learning a global system model (LTS) via grammar induction. Similar as for [29], the resulting design model does not necessarily conform to the given examples and requires that unwanted "[...] implied scenarios should be detected and excluded" [15], manually.

In [37, 36], Uchitel et al. recommend the use of high-level MSCs (HMSCs) as input for model synthesis. For larger size projects, however, constructing HMSCs may become very involved and error-prone: HMSCs pretend to model global system behavior whereas the processes contained in the nodes can only act according to local information. In [36], a logic called FLTL is employed to assure system properties while synthesizing a model – a modal transition system, which can differentiate between possible and required transitions. For each scenario, an automaton is derived which is composed in parallel with the others.

In general all three approaches exhibit the drawback of implementing synchronous (or at least not fully asynchronous) communication behavior and extracting models which only represent approximations to the system to be.

A very interesting prospect is described in [22] where Harel presents his ideas and dreams about *scenario-based programming* and proposes to use learning techniques for system synthesis. In his vision "[the] programmer teaches and guides the computer to get to *know* about the system's intended behavior [...]", just as it is our intention. This paper describes the *SMA* approach, compares it to widely adopted software development lifecycles as the waterfall model [33, 34], V-model [34], Böhm's spiral model [7, 34], and reports on first practical experiences, including an industrial case study from the automotive domain yielding insights on advantages and disadvantages of the approach.

Outline. In Section 2 the ingredients for our learning approach are described and complemented by a theoretical result on its feasibility. Section 3 describes *SMA* in detail and compares it to traditional and modern software engineering lifecycle models. In Section 4 we apply *SMA* gradually to a simple example, followed by insights on an industrial case study in Section 5.

2 INGREDIENTS OF THE *SMA*

We now recall some the basic notions of message sequence charts (MSCs), corresponding automata (communicating finite-state machines), and recall the gist of *Smyle* [8]. Moreover, we present a logic for specifying sets of MSCs.

2.1 Message Sequence Charts

Message Sequence Charts (MSCs) are an ITU standardized notation [26] for describing message exchange between concurrent processes. An MSC depicts a single partially ordered execution sequence of a system. It defines a collection of processes, which are drawn as vertical lines and interpreted as top-down time axes. Labeled vertical arrows represent message exchanges. An example MSC over three processes is depicted in Figure 1 a).

In its mathematical essence, an MSC can be understood as a graph whose nodes represent communication actions. For example, the graph in Figure 1 b) represents the MSC of Figure 1 a). A node or *event* represents the communication action indicated by its label, where, e.g., $1!2(a)$ stands for sending a message a from 1 to 2, whereas $2?1(a)$ is the complementary action of receiving a from 1 at process 2. The edges reflect causal dependencies between events. An edge can be of two types: it is either a *process edge* (proc), describing progress of one particular process, or a *message edge* (msg), relating a send with its corresponding receive event. Technically, this graph can be represented as a partial order of communication events.

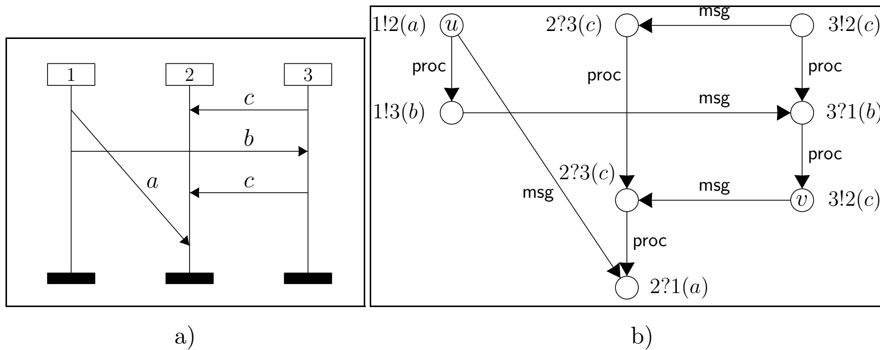

Fig. 1. a) An MSC and b) its graph

In this work we abstract from several features provided by the standard. Many of them (e.g., local actions, co-regions, etc.) can be easily included into our framework, but are omitted here for ease of presentation. Some of them, however, are excluded on purpose: loops and alternatives are not allowed as *single* executions are to be specified by MSCs. Note that, in correspondence to the ITU standard but in contrast to most works on learning MSCs, we consider the communication of an MSC to be *asynchronous* meaning that sending and receiving of a message may be delayed.

A (finite or infinite) set of MSCs, which we call an *MSC language*, may represent a system in the sense that it contains all possible scenarios that the system may exhibit. MSC languages can be characterized and represented in many ways. Here, the notion of a *regular* MSC language is of particular interest, as it comprises languages that are *learnable* in a sense made more precise below. Regularity of MSC languages is based on the concept of linearizations: A *linearization* of an MSC M containing

the events E_M is a total ordering of E_M that does not contradict the transitive closure of the edge relation. Any linearization can be represented as a word over the set of communication actions. Two sample linearizations of the MSC from Figure 1 are $l_1 = 1!2(a)3!2(c)2?3(c)1!3(b)3?1(b)3!2(c)2?3(c)2?1(a)$, $l_2 = 3!2(c)\ 2?3(c)\ 1!2(a)\ 1!3(b)\ 3?1(b)\ 3!2(c)\ 2?3(c)\ 2?1(a)$. Let $Lin(M)$ denote the set of linearizations of M and, for some set $\mathtt{M}$ of MSCs, let $Lin(\mathtt{M})$ denote the set $\bigcup_{M \in \mathtt{M}} Lin(M)$. An MSC language $\mathtt{M}$ is now regular iff $Lin(\mathtt{M})$ is regular.

2.2 Communicating Finite-State Machines

The reason of considering *regular* MSC languages in *SMA* is twofold: First, regular languages are supported by *SMA*'s learning algorithms that are briefly sketched in the next section. Second, regular MSC languages learned within *SMA* can be naturally and effectively implemented in terms of *communicating finite-state machines* (CFMs) [12]. CFMs constitute an appropriate automaton model for distributed systems where processes are represented as finite-state automata that can send messages to one another through reliable FIFO channels. We omit a formal definition of CFMs and instead refer to the example depicted in Figure 2 illustrating the *Alternating Bit Protocol* [28, 35]. There, we deal with a producer process (p) and a consumer process (c) that exchange messages from $\{0, 1, a\}$. Transitions are accordingly labeled with communication actions such as $p!c(0)$, $p?c(a)$, etc. (as we deal with two processes only, we omit the sender and receiver, just writing !0, ?a, and so on). For a concise description of this protocol, see Section 4.

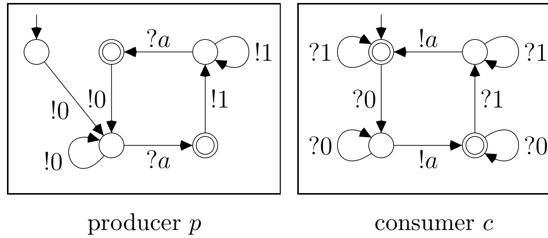

Fig. 2. Example CFM

A CFM accepts a set of MSCs in a natural manner. For example, the language of the CFM from Figure 2 contains the MSCs depicted in Figure 4. Using CFMs, we account for the *asynchronous* communication behavior whereas usually other approaches use synchronous communication. This, of course, complicates the underlying theory of learning procedures but results in a model that exactly does what the user expects and does not represent any approximation.

The formal justification of using regular MSC languages is given by the following theorem, which states that a set of MSCs is implementable as a CFM if its set of linearizations is regular, or if it can be *represented* by a regular set of linearizations.

Theorem 1 ([24, 19])**.** Let $\mathtt{M}$ be an MSC language. There is a CFM accepting precisely the MSCs from $\mathtt{M}$, if one of the following holds:

1. The set $\mathit{Lin}(\mathtt{M})$ is a regular set of words.
2. There is a channel bound $B \in \mathbb{N}$ and a regular subset L of $\mathit{Lin}(\mathtt{M})$ such that (i) any MSC from $\mathtt{M}$ exhibits a linearization that does not exceed the channel bound B, and (ii) L contains precisely the linearizations from $\mathit{Lin}(\mathtt{M})$ that do not exceed the channel bound B.

If the respective regular languages are given as finite automata, we can compute a corresponding CFM effectively.

2.3 The Gist of *Smyle*

Smyle is the learning procedure underlying *SMA* and has recently been described in [8]. As input, *Smyle* is given a set $\mathtt{M}^+$ of scenarios (called *positive*) which are desired executions of the system to be and a set $\mathtt{M}^-$ of scenarios (called *negative*) which may not be observed as system executions. If the given examples do not indicate a *single* conforming model, *Smyle saturates* both sets by asking further *queries* which are successively presented to the user who in turn has to classify each of them as either positive or negative resulting in $\bar{\mathtt{M}}^+$ and $\bar{\mathtt{M}}^-$. Eventually, the minimal deterministic finite automaton and a corresponding CFM accepting the MSCs of $\bar{\mathtt{M}}^+$ and rejecting those of $\bar{\mathtt{M}}^-$ are computed.

If a subsequent analysis of the obtained CFM shows that it does not conform to the user's intention, it can be refined by providing further examples to be added to $\bar{\mathtt{M}}^+$ or $\bar{\mathtt{M}}^-$ and reinvoking the learning procedure. It can be shown that this process eventually converges to any intended CFM [8].

At first sight, one might think that inconsistencies could be introduced by the classifications of the presented MSCs. However, this is not possible due to the simple nature of MSCs: We do not allow branching, if-then-else or loop constructs. Thus they cannot overlap and generate inconsistencies. Note moreover that the learning algorithm is deterministic in the following sense: For every (saturated) set of examples, the learning algorithm computes a *unique* CFM. This allows, within *SMA*, to rely only on all classified MSCs within a long-term project and to resume learning whenever new requirements arise. Moreover, reclassification in case of user errors is likewise simple.

An important aspect that distinguishes *Smyle* from others [29, 15] is that the resulting CFM is consistent with the set of MSCs that served as input. Other approaches project their learning result onto the processes involved, with the price that the resulting system is some over-approximation of the desired one.

2.4 MSC Patterns

When applying the *SMA*, it is useful to provide expressive though concise means to describe MSC languages, e.g., to specify mandatory or unwanted system behavior.

Over words, temporal logics such as LTL have emerged as an important ingredient in the verification and synthesis of reactive systems. For MSCs, only few attempts to define a suitable temporal logic exist. The lack of temporal logics probably traces back to the complexity of MSCs: even simple temporal logics over MSCs have an undecidable satisfiability problem. For *SMA*, we adopt the logic PDL recently proposed in [10], which is inspired by Propositional Dynamic Logic (PDL) by Fischer and Ladner [18], but adapted to MSCs. We choose PDL for three reasons: i) It is expressive (subsuming, e.g., Peled's temporal logic TLC^- [31]); ii) it combines easy to understand concepts such as regular expressions and Boolean operators; iii) its membership problem (i.e., to decide if a given MSC satisfies a given formula) can be solved in polynomial time (as we show). Note that PDL is a quite expressive logic and that satisfiability for PDL is undecidable. Fortunately, the *SMA* only builds on deciding membership problems, as they allow us to determine whether a given scenario belongs to a property that represents good or undesired behavior. This is the principal reason why PDL is better suited for our purposes than, say, high-level MSCs, whose membership problem is NP-complete [1].

The building blocks of formulas in our logic PDL are regular expressions and Boolean connectives, which can be nested arbitrarily. Instead of a formal account to PDL, which can be found in [10], we provide some example formulas demonstrating the expressive power of PDL and its usage. Essentially, we distinguish *existential* and *universal* formulas. An existential formula is of the form $\mathbf{E}\varphi$. It expresses that there is some event at which φ is satisfied. The universal formula $\mathbf{A}\varphi$, in contrast, requires that φ holds at *all* events in a given MSC. The subformula φ in $\mathbf{E}\varphi$ or $\mathbf{A}\varphi$ is thus interpreted at events of an MSC. It might be of the form $<\pi>\varphi'$ meaning that, starting in the event under consideration, there is a π-labeled path to another event that satisfies φ'. The dual construct $\sqsubset\pi\sqsupset\varphi'$ expresses that the property φ' has to hold in any event that can be reached following a π-labeled path. E.g., consider the following PDL formulas:

$$
\begin{aligned}
\varphi_1 &= \mathbf{A}\big(<(\mathsf{proc} + \mathsf{msg})^*>(\mathbf{procmax} \wedge 2?1(a))\big) \\
\varphi_2 &= \mathbf{E}\big(\mathbf{procmax} \wedge 2?1(a)\big) \\
\varphi_3 &= \mathbf{A}\big(\sqsubset 2?3(c); \mathsf{proc}; 2?3(c) \sqsupset \mathbf{false}\big).
\end{aligned}
$$

The universal formula φ_1 describes that, from any event, there is an (arbitrarily labeled) path through the MSC (expressed by $<(\mathsf{proc} + \mathsf{msg})^*>$) to another event that is maximal on its process (**procmax**) and labeled with $2?1(a)$. In other words, there shall be a greatest event in the MSC at hand and this greatest event shall be labeled with $2?1(a)$. Indeed, φ_1 is satisfied by the MSC from in Figure 1. Note that the existential formula φ_2 is not equivalent to φ_1, as φ_2 only requires the existence of an event that is *maximal* on process 2; but this event need not be the greatest in the MSC. The universal formula φ_3 forbids two consecutive events both labeled with $2?3(c)$. It is refuted by the MSC from Figure 1.

Essential for our setting is the result that we can efficiently check whether a given MSC adheres to a given PDL formula, which can be proven easily:

Theorem 2. The membership problem for *PDL* is in *PTIME*, even if the number of processes is part of the input. More precisely, given a *PDL* formula φ and an MSC M, we can decide in time $\mathcal{O}(|M| \cdot |\varphi|^2)$ if $M \models \varphi$, where $|M|$ denotes the number of events in M and $|\varphi|$ denotes the length of φ.

The logic will be used in our setting as follows: positive and negative sets of formulas Φ^+ and Φ^- are given by the user, either directly or by annotating presented MSCs. An example for a negative statement would be, say, "*there are two receives of the same message in a row*", which corresponds to the negation of the PDL formula φ_3 above. An annotated MSC for this example formula is given in Figure 6 c). Then, the learning algorithm can autonomously and efficiently check for all formulas $\varphi^+ \in \Phi^+$, $\varphi^- \in \Phi^-$ and unclassified MSCs M whether $M \not\models \varphi^+$ or $M \models \varphi^-$. If one of the two cases occurs then the set of negative samples is updated to $\{M\} \cup \mathtt{M}^-$ and in all other cases the question is passed to the user.

3 THE SMYLE MODELING APPROACH (*SMA*)

It is common knowledge [17] that traditional engineering lifecycle models like the well-known *waterfall model* [33, 34, 32, 20] or the V-model [32, 34] suffer from some severe deficiencies, despite their wide use in today's software development. One of the problems is that both models assume (implicitly) that a complete set of requirements can indeed be formulated at the beginning of the software engineering lifecycle. Although in both approaches it is possible to revisit a previously passed phase, this is considered a backwards step involving time-consuming reformulation of documents, models, or code produced in the previous and current phases, causing high additional costs for redesign.

The nature of a typical software engineering project is, however, that requirements are usually incomplete, often contradicting, and frequently changing during the project evolution. A high-level design, on the other hand, is typically a complete and consistent model that is expected to conform to the requirements. Thus, especially the step from requirements to a high-level design is a major challenge within a software engineering lifecycle: The incomplete set of requirements has to be made complete and inconsistencies have to be eliminated. An impressive example for inconsistencies in industrial-size applications is given by Holzmann [25] where for the design and implementation of a part of Signaling System 7 in the 5ESS® switching system (the ISDN User-Part protocol defined by the CCITT) "almost 55 % of all requirements from the original design requirements [...] were proven to be logically inconsistent [...]".

Moreover, also later stages of the development process often require additional modifications of requirements and the corresponding high-level design, either due to changing user requirements or due to unforeseen technical difficulties. Thus,

besides the step of generating a complete and consistent set of requirements and a conforming design model in the initial stages of a development process, a lifecycle model should support an easy adaptation of requirements and its conforming design model also at later stages.

The *SMA* is a novel software engineering lifecycle model that addresses these goals. However, as we discuss later, it may also be employed to enrich existing lifecycle models.

3.1 A Bird's-Eye View on *SMA*

The *Smyle Modeling Approach* (*SMA*) is a software engineering lifecycle model tailored to communicating distributed systems. A prerequisite is, however, that the participating units (processes) and their communication actions can be fixed in the first steps of the development process, before actually deriving a design model. Requirements for the behavior of the involved processes, however, may be given vaguely and incomplete first but are made precise within the modelling process. While clearly not every development project fits these needs, a considerable amount of systems especially in the automotive domain do, which actually motivated the development of *SMA*.

Within *SMA*, our goal is to round-off requirements, remove inconsistencies and to provide methods catering for modifications of requirements in later stages of the software engineering lifecycle. One of the main challenges to achieve these goals is to come up with simple means for concretizing and completing requirements as well as resolving conflicts in requirements. We attack this intrinsically hard problem using the following rationale:

> While it is hard to come up with a complete and consistent formal specification of the requirements, it is feasible to classify exemplifying behavior as desired or illegal. (*SMA* rationale)

This rationale builds on the well-known experience that human beings prefer to explain, discuss, and argue in terms of example scenarios but are often over-strained when having to give precise and universally valid definitions. Thus, while the general idea to formalize requirements, for example using temporal logic, is in general desirable, this formalization is often too cumbersome and therefore not cost-effective – and the result is, unfortunately, often too error-prone. This is because it is hard to have a clear (complete and consistent) picture of the system to develop right at the beginning of the software engineering lifecycle.

This also justifies our restriction to MSCs without branching, if-then-else, and loops, when learning design models: It may be too error-prone to classify complex MSCs as either wanted or unwanted behavior.

Our experience with requirements documents shows that especially requirements formulated in natural language are often explained in terms of scenarios, showing wanted or unwanted behavior of the system to develop. Additionally, it is evident

that it is easier for the customer to judge whether a given simple scenario is intended or not, in comparison to answering whether a formal specification matches the customer's needs.

The key idea of *SMA* is therefore to incorporate the *learning* algorithm *Smyle* (with supporting tool) [8] for *synthesizing* design models based on scenarios explaining requirements. Thus, requirements- and high-level design phase are interweaved. *Smyle*'s nature is to extend initially given scenarios to consider, for example, corner cases: It *generates* new scenarios whose classification as desired or undesired is indispensable to complete the design model and asks the engineer exactly these scenarios. Thus, the learning algorithm actually causes a natural iteration of the requirements elicitation and design model construction phase. Note that *Smyle* synthesizes a design model that is indeed consistent with the given scenarios and thus does precisely exhibit the scenario behavior.

SMA is tailored to component-based systems communicating with each other to achieve a common goal. A natural design model for such systems are CFMs [12]. Exemplifying behavior (*scenarios*) of such systems is best given in terms of MSCs. Thus, *SMA*, similar as its learning algorithm *Smyle*, is designed to derive CFMs based on either positively or negatively classified MSCs, representing wanted or unwanted behavior of the software system to build.

While *SMA*'s initial objective is to elaborate on the inherent correspondence of requirements and design models by asking for further exemplifying scenarios, it also provides simple means for modifications of requirements later in the design process. Whenever, for example in the testing phase, a mismatch of the implementation's behavior and the design model is witnessed which can be traced back to an invalid design model, it can be formulated as a negative scenario and can be given to the learning algorithm to update the design model. This will, possibly after considering further scenarios, modify the design model to disallow the unwanted behavior. Thus, necessary modifications of the current software system in later phases of the software engineering lifecycle can easily be fed back to update the design model. This high level of automation is aimed at an important reduction of development costs.

3.2 The *SMA* Lifecycle Model

The *Smyle Modeling Approach*, cf. Figure 3, consists of a requirements phase, a high-level design phase, a low-level design phase, and a testing and integration phase. Following modern *model-based* design lifecycle models, the implementation model is transformed semi-automatically into executable code, as it is increasingly done in the automotive and avionics domain.

In the following, the main steps of the *SMA* lifecycle model are described in more detail, with a focus on the phases depicted in Figure 3 and a brief discussion on testing and integration phases.

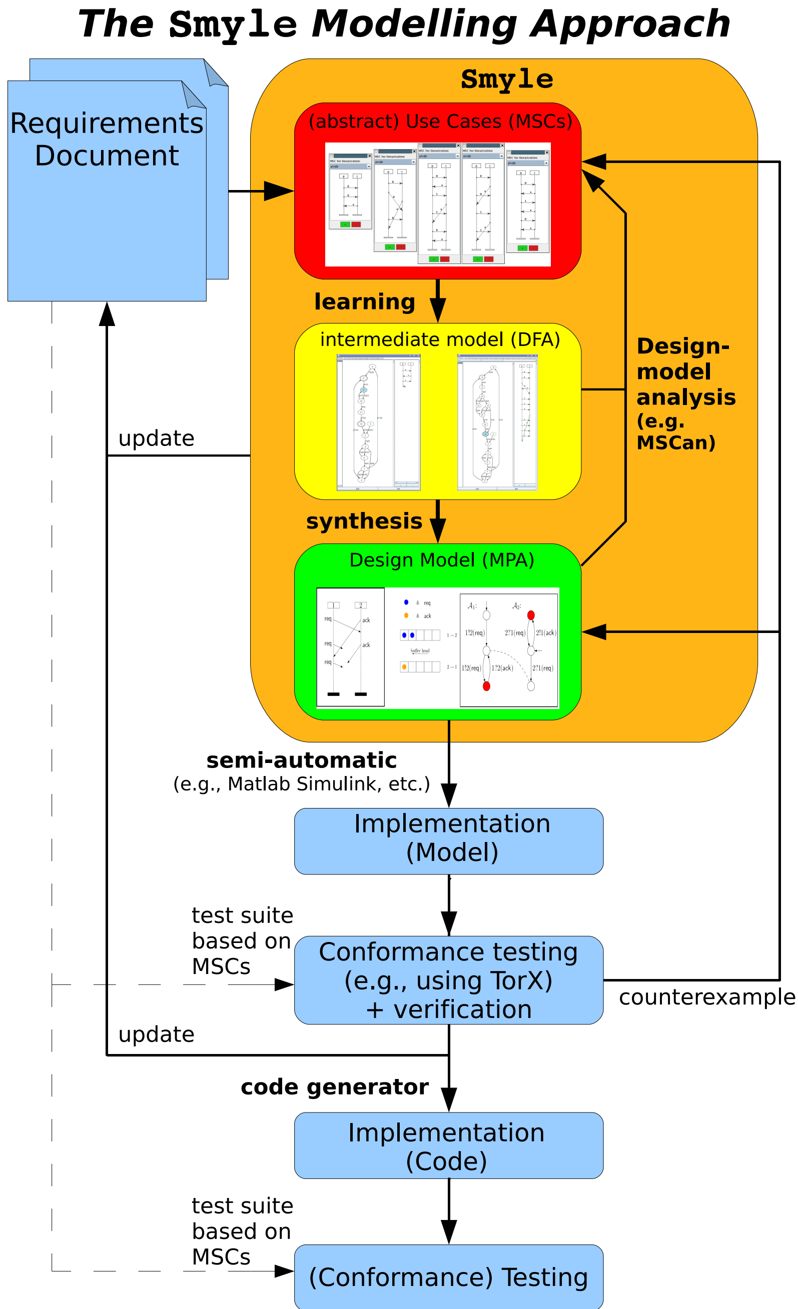

Fig. 3. The Smyle Modeling Approach: *SMA*

3.2.1 Derivation of a Design Model

According to Figure 3, the derivation of design models is divided into three steps: The first phase is called *scenario extraction phase*. Based on the usually incomplete system specification the designer has to infer a set of scenarios which will be used as input to *Smyle*.[1] After collecting this initial set of MSCs representing desired and undesired system behavior, the second phase initiates the learning algorithm to learn a system model based on these scenarios.

In the *learning and simulation phase*, the designer and client (referred to as *stakeholders* in the following) will work hand in hand according to the *designing-in-pairs* paradigm. The advantage is that both specific knowledge about requirements (contributed by the customer) and solutions to abstract design questions (contributed by the designer) coalesce into one model. With its progressive nature, *Smyle* attempts to derive a model by interactively presenting new scenarios to the stakeholders which in turn have to classify them as either positive or negative system behavior. Due to the evolution of requirements implied by this categorization the requirements document should automatically be updated incorporating the new MSCs. Additionally, the most important scenarios are to be user-annotated with the reason for the particular classification to complement the documentation. When the internal model is complete and consistent with regard to the scenarios classified by the stakeholders, the learning procedure halts and *Smyle* presents a frame for simulating and analyzing the current system. In this dedicated simulation component – depicted in Figure 5 a) and c) – the designer and customer pursue their designing-in-pairs task and try to obtain a first impression on the system to be by executing events and monitoring the resulting system behavior depicted as an MSC. In case missing requirements are detected the simulator can extract a set of counterexample MSCs which should again be augmented by the stakeholders to complete documentation. These MSCs are then introduced to *Smyle* whereupon the learning procedure continues until reaching the next consistent automaton.

The designer then advances to the *synthesis and analysis phase* where a distributed model (a CFM) is synthesized in an automated way. To get diagnostic feedback as soon as possible in the software engineering lifecycle, a subsequent analysis phase asks for an intensive analysis of the current design model. Consulting model-checking-like tools[2] as *MSCan* [9] which are designed for checking dedicated properties of communicating systems might lead to additional knowledge about the current model and its implementability. With *MSCan* the designer is able to check for potential deficiencies of the forthcoming implementation, like *non-local choice* or *non-regularity* [6, 24], i.e., process divergence. The counterexamples generated by *MSCan* are again MSCs and as such can be smoothly fed back to the learning

[1] It is worthwhile to study the results from [27] in this context, which allow to infer MSCs from requirements documents by means of natural language processing tools, potentially yielding (premature) initial behavior.

[2] Note that currently there are no general purpose model checkers for CFMs available.

phase. Instead of employing tools, the engineer could of course also try to alter the distributed model itself. Nevertheless, this is not encouraged as it obviously violates and breaks the learning-based lifecycle and conceals the danger of accidentally adding or deleting system behavior because distributed systems are easily misunderstood.

If the customer and designer are satisfied with the result the client's presence is not required any more and their direct collaboration terminates. Note that the design model obtained at this stage may also serve for a legal contract describing the system to be built.

Enhancing the learning process. While it is hard to come up with a universally valid specification right in the beginning of the design phase, typical *patterns* of clearly allowed of disallowed MSCs are usually observed during the learning phase. In this case, the logic PDL (cf. Section 2.4) can be applied to decrease the number of MSCs to classify, for reducing the designer's efforts. If the desirable or undesirable properties obey a certain structure these so-called *patterns*, representing temporal-logic properties, should be expressed in PDL directly or marked – as in Figure 6 – within an MSC featuring this behavior. In case patterns were marked in the MSCs they will automatically be transformed into PDL formulas. Afterwards, they have to be categorized as either positive or negative, as in the case of classifying MSCs. An unclassified MSC has to fulfill all positive patterns and must not fulfill any negative pattern in order to be passed to the designer. In case any positive pattern is not fulfilled or any negative pattern is fulfilled the scenario can be classified as negative without user interaction. Roughly speaking: *employing a set of formulas in the learning procedure will further ease the designers task* because she has to classify less scenarios.

3.2.2 Transformation to an Implementation Model

The output of the design model derivation phase is an abstract model of the communication behavior. It completely and correctly describes the communication structure but lacks any further functional behavior. Let us, for example, consider a web server. The model of the communication behavior of this application is learned using our tool *Smyle*. The model abstractly describes all interactions of the web server with a client (e.g., the login to a password-protected web site, or the exchange of documents). Functionality like how to check a password or how to access documents on the disk is not given. The engineer's task now is to manually or semi-automatically transform the design model into an implementation model. Of course, as no software lifecycle can claim to be a universal remedy, *SMA* also requires manual effort and human ingenuity.

For this purpose the *SMA* proposes to employ a tool suite like *Matlab, Simulink, and Stateflow. Stateflow diagrams*, for example, allow on one hand to express communication of different entities like CFMs, but do moreover allow to express detailed functional behavior. Thus, the CFMs learned (as artifact of the design phase) may

be refined to implementation models. Thanks to automatic code generators, such implementation models may automatically be translated into executable code.

Another possibility is to employ the approach described in Balarin et al. [4] where `C` code is synthesized from *cooperating finite-state machines* (CFSMs), a communicating automaton model related to the CFM model used in this article. The manual effort in this case consists in transforming the CFM as an artifact of our learning process into a CFSM as input for the synthesis process by Balarin et al. As the authors state, this method is only applicable to a restricted class of embedded systems for which features like loop bounds, which are determined at runtime, recursion, etc., must not be used. As these limitations are rather severe, this method can, in general, not be applied to all kinds of embedded software. Note that Balarin et al. use CFSMs as high-level representation and input to their synthesis method. The authors, however, do not state how to obtain these communicating automata. Thus, the *SMA* approach could be of interest for their setting, too, because it would allow for correct CFM derivation.

To sum up: depending on the complexity and requirements of the system to be either of the above procedures should be performed in order to evolve to the next phase of the software development cycle.

3.2.3 Conformance Testing

As early as possible the implementation model should be tested before being transformed into real code to lower the risk of severe design errors and supplementary costs. *SMA* suggests conformance testing as the next phase of the development lifecycle.

In general, conformance testing means to check the conformance of given standards with regard to the implementation model or implementation. Here, we consider the requirements document and the design model as the standard that an implementation should conform to.

Now, it is important to consider to which extent the implementation model has been generated automatically from the design model. Clearly, any test comparing the design model and the implementation model on automatically transformed parts is mainly suited for showing errors in the transformation rules. Whenever the implementation model is encoded manually, however, conformance testing becomes essential.

On this level of abstraction we want to perform our tests with respect to the implementation model. To this end, we can almost directly use the MSCs drawn from the requirements document augmented by the MSCs that were classified during the learning phase and apply them to the implementation.

Moreover, *SMA* suggests *model-based testing* [13] as an important building block of the conformance testing phase, as it allows for a cost-efficient testing process because the generation of tests as well as the test execution phase can be automated to a large extent. Model-based testing contains the following steps: first an abstract model of the system has to be derived. In our context this model is inferred in the

learning phase yielding an abstract model of the specification. From this abstract model we can then, in a second step, generate (abstract) test cases and finally in the third step test the (partially) manually coded implementation model.

As mentioned before, testing based on the MSCs used for learning and further ones derived from the design model using model-based testing techniques is essential whenever the implementation model is derived manually. When the communication part of the implementation model is *correctly* generated automatically, it does not have to be tested. This is because usually we assume that the user classifies MSCs correctly, and code generators generate correct output.

Nevertheless, the implementation model contains many more details in comparison to the design model. More specifically, it contains the functional behavior in detail and it is important that this functional behavior is tested. To this end, the MSCs taken from the requirements document or generated by model-based testing techniques from the design model should be refined to the level of detail to allow to check also for functional correctness. In other words, also in the case of (correctly) generated communication skeletons for the implementation model, the generation of MSCs from the design model is a valuable task, when these MSCs are (semi-automatically) refined to the level of detail of the implementation model and enriched by correctness information.

The conformance testing phase can be summarized as follows. We want to detect the bugs employing conformance testing using as a natural test suite (i.e., a set of tests) the MSCs from the requirements document, i.e., the MSCs originally contained in this document, augmented by the MSCs that were classified during the learning phase and by further ones generated from the design model using model-based testing techniques.

If, on one hand, the designer detects a communication failure during the testing phase, counterexamples in form of abstract system run and, thus, MSCs are automatically generated, and again the requirements document is updated accordingly, enclosing the new scenarios and their corresponding requirements derived by the designer. At last, the generated scenarios are introduced into *Smyle* to derive the next model.

On the other hand, the considered test cases may also be used to check the expected functional behavior to assure correctness before switching to the next phase.

In practice, model-based testing has been implemented in several software tools and has demonstrated its power in various case studies [14, 13]. For the testing phase, the *SMA* recommends tools like *TorX* or *TGV* [5].

3.2.4 Synthesis of Code, Testing, and Maintenance

Having converged to a final, consistent implementation model, a code generator is employed for generating code skeletons or even entire code fragments for the distributed system. These fragments then have to be completed by programmers e.g. by code for memory management, file handling, network communication, etc. such that, afterwards, the software can finally be installed at the client's site.

As stated in the previous paragraph, as soon as human beings contribute real code to the implementation, conformance testing should be stipulated. As in the current phase the automatically derived code skeletons were augmented by hand-written code, we schedule a second phase of testing. Similar to the last phase, we can use MSCs from the requirements document, or automatically generated ones to perform parts of the tests on an abstract level in order to receive diagnostic feedback. Again, in this phase of the software development also concrete test cases have to be created, for which the abstract tests can serve as templates to implement real test cases.

Concerning the extensibility and maintenance of the system, we obtain the following result: if new requirements arise after some operating time of the system, the old design model can be upgraded by resurrecting the *SMA* on basis of the already classified, or even partially reclassified set of scenarios, learning the new model, synthesizing the new system as explained and, thus, closing the *SMA* lifecycle.

3.3 *SMA* within the Plethora of Software Engineering Lifecycle Models

This section compares the *SMA* to other well-known traditional and modern lifecycle models.

The waterfall- and V-model. As mentioned before, major drawbacks of several traditional models like the waterfall or V-model are:

1. All requirements have to be fixed in advance.
2. As testing phases are scheduled at the end of the software development process, expensive and time-consuming backwards steps are to be expected.
3. Typically, in industrial practice, during the development phase but especially during the maintenance phase only the code is improved but the underlying documents and models are not extended or updated, to avoid overwhelming work. Then, however, significant problems arise if on the basis of the current software a new version needs to be developed.

The *SMA*, however, overcomes the first problem by interactively deriving new scenarios while models evolve towards a final conforming and validated model. The second shortcoming is addressed by intensive simulation and analysis on the design model level before actually synthesizing code. Moreover, using model-based testing techniques to check conformance of the design model and the implementation model, programming errors resulting when completing the partially generated code can be found with a huge degree of automation. The important fact here is that any deficiency encountered can usually be formulated in terms of a (mis-)behavior, expressed as an MSC. In case the misbehavior is due to an invalid design model, it can be documented in the requirements documents as well as fed back to the learning phase to improve the design model. This, on one hand, reduces the probability

of having design or implementation flaws substantially, yet allows to keep requirements and design models up-to-date. Similarly, during the maintenance phase, the modified behavior of the software system can be expressed by adding new MSCs or reclassifying previously classified MSCs to update the design model and corresponding design documents, addressing the third shortcoming identified in the waterfall model.

Note that *SMA* coincides with the waterfall model and the V-model on major milestones of these lifecycle models, namely *requirements elicitation*, *design-model elaboration*, *implementation*, *testing* and *maintenance*.

The spiral model. One of the first models which overcame the severe problems mentioned above was Boehm's *spiral model* [7, 17, 34]. For large software systems it is usually impossible to fix all requirements in advance. The spiral model therefore supports an iterative development of requirements and system prototypes, employing the following main phases: starting with the detection of goals, alternatives and constraints, an evaluation of the alternatives and risks is performed until reaching a development and testing phase. Each cycle is concluded by the planning of the next iteration. It also allows for development of incremental versions of software resolving the third drawback of the waterfall model.

The *SMA* adapted the *progressive* character of the spiral model but to our opinion has the extra benefit of easing the requirements elicitation and derivation of a design model: only a classification has to be provided. This significantly lowers the engineer's burden to define requirements. Nevertheless, the spiral model aims at developing large-scale projects while the main application area for the *SMA* is to be seen in developing software for embedded systems where the number of communicating entities is fixed a priori.

To benefit from both models one could also integrate the *SMA* partially into the spiral model. Parts of the system are then learned employing the *SMA* and the resulting components can afterwards be integrated in the overall system using the spiral model. In other words, the *SMA* would correspond to one iteration within the spiral model.

Rapid prototyping. *Rapid prototyping* (RP) [17, 34] also resolves the traditional models' deficits of defining all requirements of a system in advance. This kind of software engineering lifecycle is employed for getting deeper insight in the requirements. In several iterations prototypes are generated. The knowledge about requirements and design that is gained throughout these iterations is used as input for improving the prototypes of the next iterations. If a satisfactory prototype was created, it may serve as system specification and the software development is continued by an iterative lifecycle model (e.g., using the waterfall model) to build the final system. Note that analysis and testing phases can early be integrated into the process. This results in early feedback and, hence, less problems in later phases that would cause expensive redesign.

In software engineering several kinds of prototyping are distinguished. Our lifecycle model very much resembles the *evolutionary prototyping* approach where a prototype is refined in several iterations and in each iteration the knowledge acquired during the previous iterations is used to enhance the current prototype. However, as learning is based on exemplifying behavior that is expected to be documented within the learning process, the *SMA* does not suffer from the disadvantage of not having a formal requirements document, which is usually the case in RP, as due to time and financial constraints, the effort on such a document is often stinted.

Moreover, bridging the gap from requirements to a design model – that apparently exists in rapid prototyping as well as in most software engineering lifecycle models – is a highly creative task that involve requirements as well as design engineers with expensive expertise. The *SMA*, however, is not necessarily dependent on highly experienced design personnel but rather on requirements of engineers with domain knowledge because, as mentioned before, design questions are to a great extent solved automatically by the learning algorithm.

Agile models. Agile models [2] are a popular approach to iterative software development where the main focus is changed from creating documents at the end of each phase as in traditional software engineering to immediate interaction of human beings. Due to short development cycles the model stays highly adaptive to changes in requirements.

SMA exhibits both the direct communication with the customer while inferring the design model and the automatic creation of formal documentation, i.e., the iterative update of the requirements document. An example agile model that resembles the *SMA* in a certain way is the so-called *extreme programming model.* Extreme programming uses so-called *user stories* (i.e., scenarios). In each iteration some of these user stories are planned for implementation and, as in the *SMA*, an early integration of testing is provided. After the test case creation phase a loop of implementing, integrating and testing is performed until converging to a release. This in turn can be analyzed again to gather new user stories closing the cycle to the next iteration. Similar to the *SMA*, regular and early testing phases are stipulated. Design and implementation flaws are detected early, allowing for substantial cost reduction and considerably shorter time-to-market phases. The designing and programming-in-pairs paradigm supported by both approaches is less error-prone and thus another risk-reduction technique which lowers costs of possible redesign or reimplementation.

4 *SMA* BY EXAMPLE

Mainly for illustration, we apply the *SMA* on a concrete yet simple example. More specifically, our goal is to derive a model for the well-known *Alternating Bit Protocol* (ABP). Along the lines of [28, 35], we start with a short requirements description in natural language. Examining this description, we will identify the participating

processes and formulate some initial MSCs exemplifying the behavior of the protocol. These MSCs will be used as input for *Smyle* which in turn will ask us to classify further MSCs, before deriving a first model of the protocol. Eventually, we come up with a design model for the ABP matching the model from [35]. However, we refrain from implementing and maintaining the example, due to resource constrains.

Problem description. The main aim of the ABP is to assure the reliability of data transmission initiated by a *producer* through an *unreliable* FIFO (first-in-first-out) channel to a *consumer*. Here, unreliable means that data can be corrupted during transmission. We suppose, however, that the consumer is capable of detecting such corrupted messages. Additionally, there is a channel from the consumer to the producer, which, however, is assumed to be reliable. The protocol now works as follows: initially a bit `b` is set to 0. The *producer* keeps sending the value of `b` until it receives an acknowledgment message a from the consumer. This affirmation message is sent some time after a message of the producer containing the message content `b` is obtained. After receiving such an acknowledgment, the producer inverts the value of `b` and starts sending the new value until the next affirmation message is received at the producer. The communication can terminate after any acknowledgment a that was received at the producer side.

Applying the *SMA*. According to *SMA*, we first start with identifying the participating processes in this protocol: the *producer* p and the *consumer* c.

Next, we turn towards the *scenario extraction phase* and have to come up with a set of initial scenarios. Following the problem description, we first derive the MSC shown in Figure 4 a). It describes that indeed p sends first 0, gets an acknowledgement from c, then sends 1, and finally gets a further acknowledgement.

Let us now consider the behavior caused by the non-reliability of the channel. We could imagine that p sends a message 0 but, due to channel latency, does not receive a confirmation within a certain time bound and thus sends a second 0 while the first one is already being acknowledged by c. This yields the MSC in Figure 4 b).

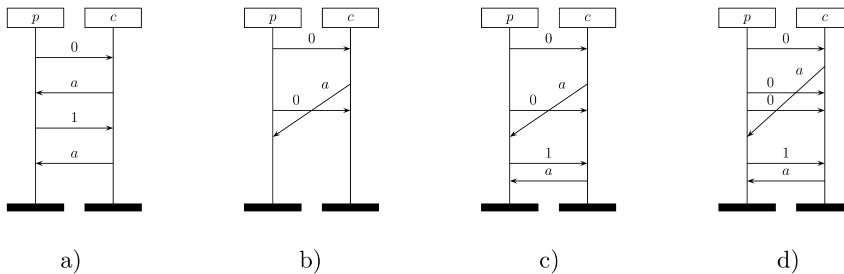

Fig. 4. The initial input scenarios for *Smyle*

The problem description tells us: *after each acknowledgment the bit* `b` *is inverted*. Thus, the previous scenario is extended by a second phase where `b` = 1 is sent and directly acknowledged, shown in Figure 4 c). To exemplify that, on the producer's side, more than one message can be corrupted, we derive a scenario that amplifies the previous one: We add one more message with content `b` = 0 to the first phase of scenario c) yielding the scenario from Figure 4 d).

We start the learning phase and feed the charts to *Smyle*, proceeding to the second step in the design phase. Within this learning phase, *Smyle* asks us to classify further 44 scenarios – most of which we are easily able to negate – before providing a first hypothesis of the design model.

Now the simulation phase is activated (cf. Figure 5 a)), where we can test the current model. We execute several events as shown in the right part of Figure 5 a) and review the model's behavior. We come across an execution where after an initial phase of sending a 0 and receiving the corresponding affirmation we expect to observe a similar behavior as in Figure 4 b) (but now containing the message content `b` = 1). According to the problem description this is a feasible protocol execution but is not contained in our system yet. Thus, we encountered a missing scenario. Therefore, instead of proceeding to the *synthesis and analysis* phase, we enter the *scenario extraction phase* again, formulate the missing scenario (cf. Figure 5 b)), and input it into *Smyle* as a counterexample to the current model.

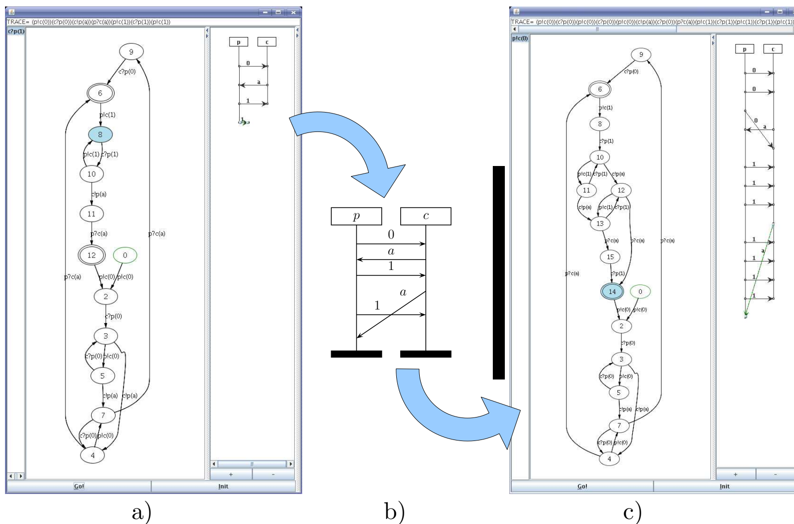

Fig. 5. *Smyle*'s simulation window: a) intermediate internal model with missing behavior b) missing scenario c) final internal model

As before, *Smyle* presents further MSCs that we have to classify: Among others, we are confronted with MSCs that (1) do not end with an acknowledgment (cf. Figure 6 a)) and with MSCs that (2) have two subsequent acknowledgment

events (cf. Figure 6 c)). Both kinds of behavior are not allowed according to the problem description. We identify a pattern in each of these MSCs, by marking the parts of the MSCs as shown in Figure 6 a) and c), yielding internally PDL formulas representing these patterns:

(1) $\mathbf{E}(\mathbf{procmax} \wedge p?c(a))$

(2) $\mathbf{A}(\sqsubset p?c(a); \mathsf{proc}; p?c(a) \; + \; c!p(a); \mathsf{proc}; c!p(a) \sqsupset \mathbf{false})$

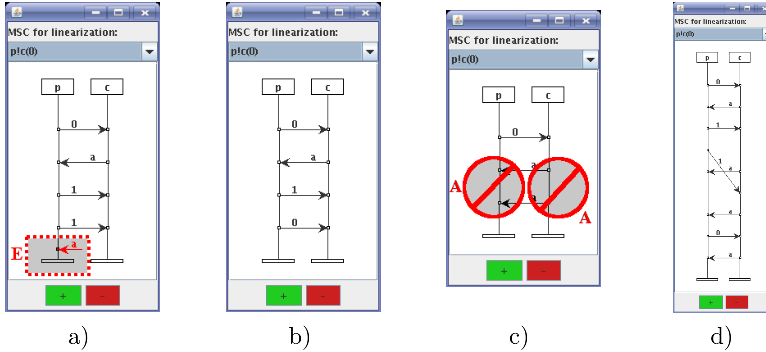

Fig. 6. Patterns for (un)desired behavior

Instead of visually annotating MSCs, the formulas can also be directly entered via a dedicated formula editor. To tell *Smyle* to abolish all MSCs fulfilling the patterns we mark them as unwanted behavior. Thus, the MSCs from Figure 6 b) and d) are automatically classified as negative later on. In addition, we reflect these patterns in the requirements documents by adding, for example, the explanation that *every message exchange has to end with an acknowledgment* and its formal specification (1). With the help of these two patterns, we continue our learning effort and end with the next hypothesis after a total of 55 user queries. Note that without adding these patterns, we would have needed 70 user queries. Moreover, identifying three more obvious patterns at the beginning of the learning process, we could have managed to infer the correct design model with only 12 user queries in total. Of course one can argue that this is a high number of scenarios to classify but this is the price one has to pay for getting an exact system and not an approximation (that indeed can be arbitrarily inaccurate) as in related approaches.

At the end of the second iteration through the learning phase we presented the simulation frame (Figure 5 c)) again. An intensive simulation does not give any evidence of wrong behavior. Thus, we enter the analysis phase to check the model with respect to further properties. For example, we check whether the resulting system can be implemented fixing a maximum channel capacity in advance. *MSCan* tells us that the system does not fulfill this property. Therefore we need to add a (fair) scheduler to make the protocol work in practice. According to Theorem 1 a CFM is constructed which exactly is the one from Figure 2.

5 *SMA* IN AN INDUSTRIAL CASE STUDY

This section examines a real-world industrial case-study derived within a project with a Bavarian automotive manufacturer. Our task was to derive an initial design model for the problem described below. The main goal of this section is not to present a detailed report of the underlying system and the way the *SMA* was employed but to share insights acquired while inferring the design model using the *SMA*.

Problem description. The case study describes the functionality of the automotive manufacturer's onboard diagnostic service integrated into their high-end product. In case the climate control unit (CCU) of the automobile does not operate as expected a report is sent to the *onboard diagnostic service* which in turn initiates a CCU-self-diagnosis and waits for response to the query. After the reply the driver has to be briefed about the malfunction of the climate control via the car's multi-information-display. The driver is asked to halt at the next gas station where the onboard diagnostic service communicates the problems to the automotive manufacturer's central server. A diagnostic service is downloaded from the server and executed locally on the vehicle's on-board computer. The diagnostic routine locates the faulty component within the CCU and sends the problem report back to the central server. In case of a hardware failure a car garage could be informed and the replacement part could be reordered immediately to keep the CCU's downtime as short as possible. If no hardware failure is detected a software update (if available) is installed and the CCU reset.

Inferring a system model. This section briefly describes the main steps that were necessary to infer a design model for the described use case. Throughout the whole process a project member of our client was present to discuss the requirements with our designer.

First, all the processes involved had to be identified. It was easy to see that *CCU service*, *onboard diagnostic service*, *display*, *driver* and *central server* were the main participating units. Of course there could be a large number of cars trying to contact the server but because *Smyle* can only deal with a bounded number of processes the abstraction to one car was used to establish a proof of concept for this project.

After extracting some positive and negative scenarios (MSCs) from the problem description *Smyle* was started by introducing the collection of scenarios to the tool. These MSCs were used to guide the learning process towards a final design model as described in Section 3.2.

Lessons learned. By applying *SMA* to the given problem we were able to infer a system model in less than one afternoon fulfilling exactly the requirements imposed by our customer.

Throughout the whole process we applied the designing-in-pairs paradigm to minimize the danger of misunderstandings and resulting system flaws. The early feedback of the simulation and analysis resulted in finding missing system behavior and continuously growing insights – even on our customers site – about the client's needs. The automated scenario derivation of the integrated tool *Smyle* was found to be very helpful for our client. The major gain is that s/he does not have to come up with scenarios by him/herself. Thus, even corner cases (i.e., exceptional scenarios the client did not consider) were covered. As requirements in the *SMA* are accumulated in an iterative process, growing system knowledge could be applied to derive new patterns easing the design task and to obtain increasingly more elaborate design models. Last but not least the *on-the-fly* completion of the requirements document resulted in a complete system description after finishing the design phase. This description could then be used as contract for the final implementation.

Besides all the positive issues we also faced inconveniences using the *SMA*. Finding an initial set of scenarios turned out in some cases to be a difficult task. This could be eased in the future by integrating an approach proposed in [27] where scenarios represented as MSCs are derived from natural language specifications. These could then smoothly be fed to *Smyle*.

When we entered the simulation phase we recognized that the simulation facilities *Smyle* offered were still too rudimentary for larger size systems. For future projects we have to extend the simulator to be able to generate random simulations and test suites for the testing phase and include components for specifying and checking correctness criteria for certain runs through the system.

During the first meeting with our customer an initial version of the *SMA* did not allow for logical formulas; but due to the high amount of user queries (almost 400) we proposed the idea of using PDL which both parties found very promising. Therefore it was integrated into our approach and directly yielded a substantial reduction of user queries (gathering 10 patterns resulted in less than 70 queries).

In general, this case study testified the viability of the *SMA* for inferring a design model for a real-world example.

6 CONCLUSION

The *SMA* provides the possibility to gradually develop and refine requirements, naturally supports evolution of requirements, and allows for a rather inexpensive redesign in case anomalous system behavior is detected during analysis, testing, or maintenance. Design questions are solved automatically by the learning algorithm. The engineer only has to classify the presented scenarios, once an appropriate architecture has been fixed. The design phase is complemented by a powerful simulation and analysis phase that allows for detecting design flaws at an early stage. The design, analysis and testing phases are supported by dedicated tool-suites. By applying these tools, design or implementation errors are detected as early as possible causing substantial reduction of costs and the overall expected time-to-market. Also,

the intermediate design models can be used to broaden insights on the system and reduce the probability of missing important system properties in the final implementation. An important feature of the *SMA* which we could also sense while performing the presented case study was that there is no need for mandatory expertise. The *SMA* does not necessarily require highly experienced personnel for inferring a design model as design questions are resolved by the learning algorithm autonomously. The overall distinguishing feature of *SMA* is that later changes of the software product (to be) in the implementation, integration, or the maintenance phase, can easily be incorporated into the design model thanks to the learning approach, by reflecting the changes in terms of scenarios.

So far, the *SMA* is restricted to the development of distributed communicating systems with a bounded number of processes, as it is usually the case when developing embedded systems, for example in the automotive, avionics, or telecommunication domain.

This paper reports on positive practical experiences showing benefits of the suggested methodology, yet leaves detailed empirical studies as future work.

REFERENCES

[1] Alur, R.—Etessami, K.—Yannakakis, M.: Realizability and Verification of MSC Graphs. TCS, Vol. 331, 2005, No. 1, pp. 97–114.

[2] Ambler, S. W.—Jeffries, R.: Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. Wiley, 2002.

[3] Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Inf. Comput., Vol. 75, 1987, No. 2, pp. 87–106.

[4] Balarin, F.—Chiodo, M.—Giusto, P.—Hsieh, H.—Jurecska, A.—Lavagno, L.—Sangiovanni-Vincentelli, A. L.—Sentovich, E.—Suzuki, K.: Synthesis of Software Programs for Embedded Control Applications. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, 1999, No. 6, pp. 834–849.

[5] Belinfante, A.—Frantzen, L.—Schallhart, C.: Tools for Test Case Generation. In: Model-Based Testing of Reactive Systems, pp. 391–438, 2004.

[6] Ben-Abdallah, H.—Leue, S.: Syntactic Detection of Process Divergence and Non-Local Choice in Message Sequence Charts. In TACAS, Vol. 1217 of LNCS, pp. 259–274, Springer 1997.

[7] Boehm, B. W.: A Spiral Model of Software Development and Enhancement. IEEE Computer, Vol. 21, 1988, No. 5, pp. 61–72.

[8] Bollig, B.—Katoen, J. P.—Kern, C.—Leucker, M.: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning. In TACAS, Vol. 4424 of LNCS, pp. 435–450, Springer 2007.

[9] Bollig, B.—Kern, C.—Schlütter, M.—Stolz, V.: MSCan: A Tool for Analyzing MSC Specifications. In TACAS, Vol. 3920 of LNCS, pp. 455–458, Springer 2006.

[10] Bollig, B.—Kuske, D.—Meinecke, I.: Propositional Dynamic Logic for Message-Passing Systems. In FSTTCS, Vol. 4855 of LNCS, pp. 303–315, Springer 2007.

[11] Bontemps, Y.—Heymand, P.—Schobbens, P. Y.: From Live Sequence Charts to State Machines and Back: A Guided Tour. IEEE TSE, Vol. 31, 2005, No. 12, pp. 999–1014.

[12] Brand, D.—Zafiropulo, P.: On Communicating Finite-State Machines. Journal of the ACM, Vol. 30, 1983, No. 2, pp. 323–342.

[13] Broy, M.—Jonsson, B.—Katoen, J. P.—Leucker, M.—Pretschner, A. (Eds.): Model-Based Testing of Reactive Systems. Vol. 3472 of LNCS, Springer 2005.

[14] Craggs, I.—Sardis, M.—Heuillard, T.: Agedis Case Studies: Model-Based Testing in Industry. In Eur. Conf. on Model Driven Softw. Eng., pp. 106–117, 2003.

[15] Damas, C.—Lambeau, B.—Dupont, P.: Generating Annotated Behavior Models from End-User Scenarios. IEEE TSE, Vol. 31, 2005, No. 12, pp. 1056–1073.

[16] Damm, W.—Harel, D.: LSCs: Breathing Life Into Message Sequence Charts. Formal Methods in System Design, Vol. 19, 2001, No. 1, pp. 45–80.

[17] Easterbrook, S. M.: Requirements Engineering. Unpublished manuscript available at: `http://www.cs.toronto.edu/~sme/papers/2004/FoRE-chapter03-v8.pdf`, 2004.

[18] Fischer, M. J.—Ladner, R. E.: Propositional Dynamic Logic of Regular Programs. J. Comput. System Sci., Vol. 18, 1979, No. 2, pp. 194–211.

[19] Genest, G.—Kuske, D.—Muscholl, A.: A Kleene Theorem and Model Checking Algorithms for Existentially Bounded Communicating Automata. I & C, Vol. 204, 2006, No. 6, pp. 920–956.

[20] Ghezzi, C.—Jazayeri, M.—Mandrioli, D.: Fundamentals of Software Engineering. Prentice-Hall, 2nd edition, 2002.

[21] Hamon, G.—Rushby, J. M.: An Operational Semantics for Stateflow. In FASE, Vol. 2984 of LNCS, pp. 229–243, Springer 2004.

[22] Harel, D.: Can Programming Be Liberated, Period? Computer, Vol. 41, 2008, No. 1, pp. 28–37.

[23] Harel, D.—Marelly, R.: Come, Let's Play. Springer 2003.

[24] Henriksen, J. G.—Mukund, M.—Kumar, K. N.—Sohoni, M.—Thiagarajan, P. S.: A Theory of Regular MSC Languages. Inf. and Comput., Vol. 202, 2005, No. 1, pp. 1–38.

[25] Holzmann, G. J.: The Theory and Practice of a Formal Method: Newcore. In IFIP Congress (1), pp. 35–44, 1994.

[26] ITU-TS Recommendation Z.120 (04/04): Message Sequence Chart, 2004.

[27] Kof, L.: Scenarios: Identifying Missing Objects and Actions by Means of Computational Linguistics. In 15th IEEE RE, pp. 121–130, 2007.

[28] Lynch, N.: Distributed Algorithms. Morgan Kaufmann, 1997.

[29] Mäkinen, E.—Systä, T.: MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML. In ICSE, pp. 15–24, IEEE Computer Society 2001.

[30] Nuseibeh, B.—Easterbrook, S.: Requirements Engineering: A Roadmap. In ICSE, pp. 35–46, ACM, 2000.

[31] Peled, D.: Specification and Verification of Message Sequence Charts. In FORTE/PSTV, Vol. 183 of IFIP Conference Proceedings, pp. 139–154, Kluwer, B. V. 2000.

[32] Pressman, R. S.: Software Engineering: A Practitioner's Approach. McGraw-Hill 2004.

[33] Royce, W.: Managing the Development of Large Software Systems: Concepts and Techniques. In ICSE, pp. 328–338, IEEE Computer Society 1987.

[34] Sommerville, I.: Software Engineering. Addison-Wesley Longman, Amsterdam, 8th edition, June 2006.

[35] Tanenbaum, A. S.: Computer Networks. Prentice Hall 2002.

[36] Uchitel, S.—Brunet, G.—Chechik, M.: Behaviour Model Synthesis from Properties and Scenarios. In ICSE, pp. 34–43, IEEE Computer Society 2007.

[37] Uchitel, S.—Kramer, J.—Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE TSE, Vol. 29, 2003, No. 2, pp. 99–115.

Benedikt Bollig received his Ph. D. degree from Aachen University of Technology (RWTH Aachen) in 2005. A revised version of his thesis on automata models and logics for message sequence charts has been published by Springer. In 2003/2004, the German Academic Exchange Service (DAAD) funded his six-month research stay at Birmingham University. Since 2005, he is a CNRS full-time researcher. His research interests are centered around logics for specification, formal languages and automata, with a focus on applications in the synthesis and verification of concurrent and timed systems.

Joost-Pieter Katoen is a Full Professor at the RWTH Aachen University and is associated to the University of Twente. His research interests are concurrency theory, model checking, timed and probabilistic systems, and semantics. He co-authored more than 100 journal and conference papers, and recently published a comprehensive book (with Christel Baier) on "Principles of Model Checking".

Carsten Kern received his diploma in computer science in 2005 from RWTH Aachen University. In August 2009 he successfully defended his dissertation at the chair of Software Modeling and Verification at the RWTH Aachen University where he is employed for another two months. His current research interests cover learning of nondeterministic and communicating automata.

Martin Leucker is currently a Professor at the Technische Universität München for theoretical computer science and software reliability. He obtained his Ph. D. at RWTH Aachen, Germany, and worked afterwards as a postdoc at the University of Philadelphia, USA, and, within the European Research and Training Network on Games, at Uppsala University, Sweden. He pursued his habilitation at TU München while being a member of Manfred Broy's group on Software and Systems Engineering. He is the author of more than 60 reviewed conference and journal papers ranging over software engineering, formal methods, and theoretical computer science.