SMA—The Smyle Modeling Approach*

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

Abstract. This paper introduces the model-based software development methodology *SMA*—the Smyle *Modeling Approach*—which is centered around *Smyle*, 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. The learning approach is complemented by *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 complemented by an effective analysis phase that allows for detecting design flaws at an early design stage. This paper describes the approach and reports on first practical experiences.

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

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.

Problems in software engineering cycles occur if, e.g., the number of requirements is abundant, they are ambiguous, contradictory, and change over time. Evolving requirements may be due to changing user requirements or to anomalous system behavior detected at later design stages and thus occur at all stages of the development process.

This paper presents a *novel* software engineering lifecycle model 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

* This work is partially supported by EGIDE/DAAD (Procope 2008), DOTS (ANR-06-SETIN-003), and P2R MODISTECOVER.

been adopted by the UML, are standardized by the ITU [22], and are part of several requirements elicitation techniques such as CREWS [26].

At the heart of our approach—called the Smyle Modeling Approach (SMA)— dedicated *learning techniques* support the engineer to interactively obtain implementation-independent design models from MSCs exemplifying the system's behavior. These techniques are implemented in the tool *Smyle* (Synthesizing models by learning from examples, cf. [5]). 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 uniquely determine a (minimal) system model. As abstract design models, *Smyle* synthesizes distributed finite-state automata (referred to as communicating finite-state machines, or CFMs for short) [5]. This model is implementation-independent and describes the local control flow as finite automata which communicate via unbounded order-preserving channels.

The learning approach is complemented by so-called *scenario patterns* where the engineer can specify *clearly* desired or unwanted behavior 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* suggests an intensive analysis of the obtained model, first by comprehensive simulation and second by checking elementary correctness properties of the CFM, for example by means of model checking or dedicated analysis algorithms [6]. This allows for an early detection of design flaws. In case of a flaw, 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 [17] from which executable code is automatically generated using tools as Matlab/Simulink. The final stage of *SMA* is a model-based testing phase [10] in which it is checked whether the software conforms to the high-level design description. The MSCs used for formalizing requirements now serve as abstract test cases. Moreover, 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 [2] that can easily be plugged in into our design cycle. Again, any test failure 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. In [32, 31], Uchitel et al. recommend the use of high-level MSCs (HMSCs) as input for model synthesis. High-level MSCs aim at specifying the overall system behavior, yet are hard to adapt when unwanted behavior has to be removed or wanted behavior has to be defined. The same problem arises for Live Sequence Charts or related formalisms [13, 19, 8]. In general, whenever modeling the overall global system behavior, a modification due to changing requirements is cumbersome and error prone.

The approaches taken in [25] and [12] are, similarly as *Smyle*, based on learning techniques. The general advantage of learning techniques is that changing requirements can be incorporated into the learning process. However, the algorithms of [25] and [12] both have the drawback that the resulting design model does not necessarily conform to the given examples and requires that unwanted "[...] implied scenarios should be detected and excluded" [12], manually, while *Smyle* does conform to the given examples.

A very interesting prospect is described in [18] 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.
An extended journal version of this paper will be available as [7].
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 message sequence charts (MSCs), communicating finite-state machines, describe the gist of *Smyle* and present a logic for specifying sets of MSCs.

2.1 Message Sequence Charts

Message Sequence Charts (MSCs) are an ITU standardized notation [22] 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, cf. Figure 1 (a).

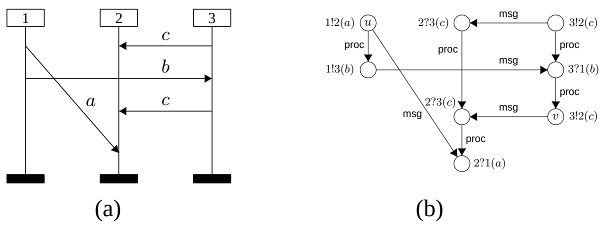


Fig. 1. An MSC (a) and its graph (b)

An MSC can be understood as a graph whose nodes represent communication actions, e.g., 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. This graph can be represented as a partial order of communication events.

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. 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* . Regularity of MSC languages is based on linearizations: A *linearization* of an MSC M is a total ordering of its events 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 $l_1 = 1!2(a)3!2(c)2?3(c)1!3(b)3?1(b)3!2(c)2?3(c)2?1(a)$ and $l_2 = 3!2(c)\ 2?3(c)\ 1!2(a)\ 1!3(b)\ 3?1(b)\ 3!2(c)\ 2?3(c)\ 2?1(a)$. Let $Lin(M)$ denote the set of linearizations of M and, for set $\mathbb{M}$ of MSCs, let $Lin(\mathbb{M})$ denote $\bigcup_{M \in \mathbb{M}} Lin(M)$.

2.2 Communicating Finite-State Machines

Regular MSC languages can be naturally and effectively implemented in terms of *communicating finite-state machines* (CFMs) [9]. 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* [24, 30]. There, a producer process (p) and a consumer process (c) exchange messages from $\{0, 1, a\}$. Transitions are labeled with communication actions such as $p!c(0)$, $p?c(a)$, etc. (abbreviated by $!0$, $?a$, and so on). For a concise description of this protocol, see Section 4. 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.

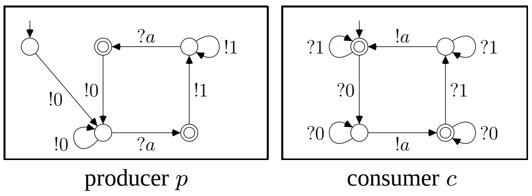


Fig. 2. Example CFM

Using CFMs, we account for the *asynchronous* communication behavior whereas usually other approaches use synchronous communication. This complicates the underlying theory of learning procedures but results in a model that exactly does what the user expects and does not represent an over-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 ([20, 15]). *Let $\mathbb{M}$ be an MSC language. There is a CFM accepting precisely the MSCs from $\mathbb{M}$, if one of the following holds:*

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

If the 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 [5]. As input, *Smyle* is given a set $\mathbb{M}^+$ of positive scenarios which are desired executions of the system to be and a set $\mathbb{M}^-$ of negative scenarios which should 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 $\bar{\mathbb{M}}^+$ and $\bar{\mathbb{M}}^-$. Otherwise, a minimal deterministic finite automaton and a corresponding CFM accepting the MSCs of $\bar{\mathbb{M}}^+$ and rejecting those of $\bar{\mathbb{M}}^-$ 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 $\bar{\mathbb{M}}^+$ or $\bar{\mathbb{M}}^-$ and reinvoking the learning procedure. This process eventually converges to any intended CFM [5].

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 [25, 12] 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, accepting that the resulting system is a (coarse) over-approximation.

2.4 MSC Patterns

In order to significantly reduce the number of scenarios the user has to classify during a learning phase, it is worthwhile to consider a formalism where (un)desired behavior can a priori be specified in terms of logical formulas.

Due to space constraints we only give a superficial description of how to apply such a logic within the SMA. A more sophisticated introduction can be found in [7].

The logic we employ will be used as follows: positive and negative sets of formulas Φ^+ and Φ^- are input by the user, either directly or by annotating MSCs. An example for a negative statement would be, say, "*there are two receipts of the same message in a row*". An annotated MSC for this example formula is given in Figure 6 (c). Then, the learning algorithm can 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 so, then the set of negative samples is updated to $\{M\} \cup \mathbb{M}^-$ and otherwise the question is passed to the user.

3 The Smyle Modeling Approach

It is common knowledge [14] that traditional engineering lifecycle models like the *waterfall model* [28, 29, 27, 16] or the V-model [27, 29] 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 be formulated at the beginning of the 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, causing high additional costs.

The nature of a typical software engineering project is, however, that requirements are usually incomplete, often contradicting, and frequently changing. 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: 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 [21] 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, a lifecycle model should support an easy adaptation of requirements and its conforming design model also at later stages. The *SMA* is a new software engineering lifecycle model that addresses these goals.

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 process. While clearly not every development project fits these needs, a considerable amount of systems especially in the automotive domain do.

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 overstrained 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 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, expressing 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 novel *learning* algorithm *Smyle* (with supporting tool) [5] 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.

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 in detail

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 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.

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]

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

[1] It is worthwhile to study the results from [23] in this context, which allow to infer MSCs from requirements documents by means of natural language processing tools, potentially yielding (premature) initial behavior.

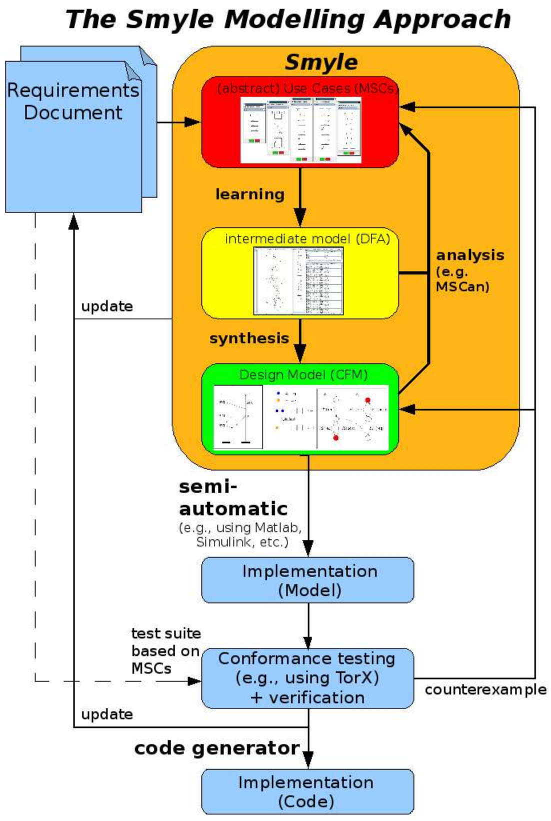


Fig. 3. The Smyle Modeling Approach: *SMA*

for an intensive analysis of the current design model. Consulting model-checking-like tools[2] as *MSCan* [6] 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* [3, 20], i.e., process divergence. The counterexamples generated by *MSCan* are again MSCs and as such can be fed back to the learning phase.

If the customer and designer are satisfied with the result the client's presence is not required anymore and their direct collaboration terminates. Note that the design model obtained at this stage may also serve as a legal contract for 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 or disallowed MSCs usually are observed during the learning phase. 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 some positive pattern is not fulfilled or

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

some negative pattern is fulfilled the scenario is be classified as negative without user interaction. Roughly speaking: *employing a set of formulas in the learning procedure will further ease the designer's task* because she has to classify less scenarios.
Transformation to an implementation model. The engineer's task now is to semi-automatically transform the design model into an implementation model. For this purpose the *SMA* proposes to employ tools like *Matlab Simulink* which takes as input for example a so-called *Stateflow diagram* [17] and transforms it into an implementation model. Hence, the manual effort the designer has to perform in the current phase reduces to transforming the CFM (as artifact of the design phase) into the input language (e.g., Stateflow).
Conformance testing. As early as possible the implementation model should pass a testing phase before being transformed into real code to lower the risk of severe design errors and supplementary costs. *SMA* employs *model-based testing* [10] as it allows a much more systematic treatment by mechanizing the generation of tests as well as the test execution phase. Moreover, in the *SMA* the MSCs classified during the learning phase and contained in the requirements document enriched by additional MSCs form a natural test suite (a set of tests). If the designer detects a failure during the testing phase, counterexamples 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 refine the model. In practice, model-based testing has been implemented in several software tools and demonstrated its power in various case studies [11, 10].
Synthesis of code 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 such that afterwards the software can finally be installed at the client's site. If new requirements arise after some operating time of the system the old design model can be upgraded by restarting the *SMA*.

3.3 *SMA* vs. other Lifecycle Models

This section briefly compares the *SMA* to other well-known traditional and modern lifecycle models. Due to lack of space, an extended comparison including coarse descriptions of the lifecycle models mentioned below can be found in [7].

In contrast to traditional lifecycle models like the well-known *waterfall-* and V-model in *SMA* requirements need not be fixed in advance but can be derived interactively while evolving towards a final conforming and validated model. Intensive simulation and analysis phases reduce the need for costly and time-consuming backward steps during the software development process. While in many processes the documentation is not regularly updated the *SMA* provides means for extending this documentation whenever additional information becomes available. Compared to several modern lifecycle models like the *spiral model* [4] and *rapid prototyping* [14, 29], *SMA* adapted the feature of periodic prototype generation in order to iteratively improve the design model by constantly learning from the insights achieved during the previous iteration. But to our opinion it has the extra benefit of only demanding a classification for automatically derived scenarios whereas in other models these scenarios have to be derived manually, first. However, the spiral model describes a more general process as it aims at developing large-scale projects while the main application area for *SMA* is to be seen in developing software for embedded systems where the number of communication entities is fixed a priori. Another advantage of *SMA* compared to, e.g., rapid prototyping is that for closing the gaps between requirements and design model there is no mandatory need for highly experienced and thus very expensive design personnel. Requirements engineers with specific domain knowledge, however, are sufficient because design questions are mainly solved automated by the learning procedure. A last model we would like to compare *SMA* to is the *extreme programming model* [1] where, similarly, in each iteration *user stories* (i.e., scenarios) are planned for implementation and regular and early testing phases are stipulated. As a further risk reduction technique both models employ *designing- and programming-in-pairs*, thus lessening the danger of errors and lowering the costs of possible redesign or implementation.

4 *SMA* by example

Our goal now is to derive a model for the well-known *Alternating Bit Protocol* (ABP). Along the lines of [24, 30], 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 [30]. 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.

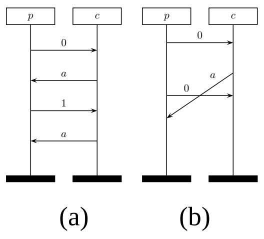


Fig. 4. Two input scenarios for *Smyle*

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 received acknowledgment a that was received at the producer side.

Applying the 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). Let us now consider the

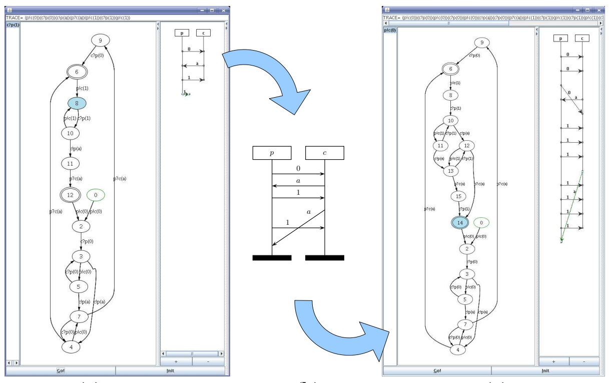


Fig. 5. *Smyle*'s simulation window: (a) intermediate internal model with missing behavior (b) missing scenario (c) final internal model

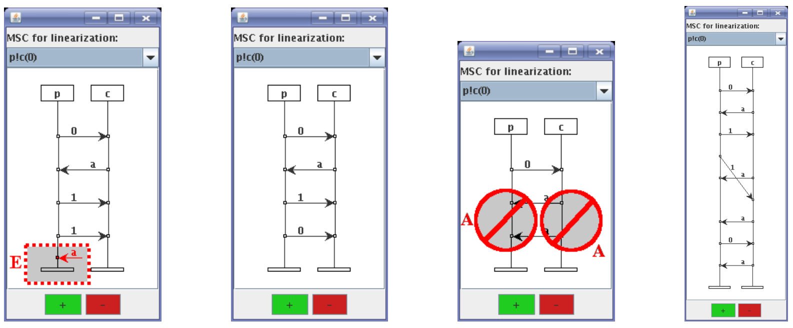
behavior caused by the non-reliability of the channel. We could imagine that p sends a message 0 but, due to channel latency, does not receive a confirmation within a certain time bound and thus sends a second 0 while the first one is already being acknowledged by c. This yields the MSC in Figure 4 (b).

Within the learning phase, *Smyle* asks us to classify further scenarios —most of which we are easily able to negate—before providing a first 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 $\mathsf{b} = 1$). 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, we enter the *scenario extraction phase* again, formulate the missing scenario (cf. Figure 5 (b)), and input it into *Smyle* as a counterexample.

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 the patterns:

1. *Every system run has to finish with an acknowledgement* a.
2. *There must never be two subsequent sends or receipts of an acknowledgement* a.

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 system run has to end with an acknowledgment* (cf. (1)) and its formal specification. With the help of these two patterns,



(a) (b) (c) (d)

Fig. 6. Some patterns for (un) desired behavior

we continue our learning effort and end with the next hypothesis after a total of 55 user queries. Without patterns, we would have needed 70 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. 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 an intensive simulation (cf. Figure 5 (c)) 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. 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*.

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 be reordered to minimize the CCU's downtime. If no hardware failure is detected a software update (if available) is installed and the CCU reset.

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.

Lessons learned. Throughout the entire 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 was found to be a major gain because 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 which 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 in some cases out to be a difficult task. This could be eased in the future by integrating an approach proposed in [23] where scenarios represented as MSCs are derived from natural language specifications. These could then smoothly be fed to *Smyle*. Moreover, the simulation facilities have to be improved allowing for random simulations etc.

Additional details on lessons learned can be found in [7].

6 Conclusion

This paper presented a software engineering lifecycle model centered around learning and early analysis in the design trajectory. Our model is described, has been compared with the main development models, and applied to a toy, as well as an industrial example. Further applications are planned to show its feasibility and to refine the method.

References

1. Ambler, S. W., Jeffries, R.: *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley (2002)
2. Belinfante, A., Frantzen, L., Schallhart, C.: Tools for test case generation. In: *Model-Based Testing of Reactive Systems*. (2004) 391–438
3. Ben-Abdallah, H., Leue, S.: Syntactic detection of process divergence and non-local choice in message sequence charts. In: *TACAS. LNCS*, Vol. 1217. Springer (1997) 259–274
4. Boehm, B. W.: A spiral model of software development and enhancement. *IEEE Computer* **21** (1988) 61–72
5. 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. LNCS*, Vol. 4424. Springer (2007) 435–450
6. Bollig, B., Kern, C., Schlütter, M., Stolz, V.: MSCan: A tool for analyzing MSC specifications. In: *TACAS. LNCS*, Vol. 3920. Springer (2006) 455–458

7. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M.: SMA—The Smyle Modeling Approach. Computing and Informatics, to appear (2009)
8. Bontemps, Y., Heymand, P., Schobbens, P.-Y.: From live sequence charts to state machines and back: a guided tour. *IEEE TSE* **31** (2005) 999–1014
9. Brand, D., Zafiropulo, P.: On communicating finite-state machines. *J. of the ACM* **30** (1983) 323–342
10. Broy, M., Jonsson, B., Katoen, J. P., Leucker, M., Pretschner, A. (eds.): *Model-based Testing of Reactive Systems*. *LNCS*, Vol. 3472. Springer (2005)
11. Craggs, I., Sardis, M., Heuillard, T.: Agedis case studies: Model-based testing in industry. In: *Eur. Conf. on Model Driven Softw. Eng.* (2003) 106–117
12. Damas, C., Lambeau, B., Dupont, P.: Generating annotated behavior models from end-user scenarios. *IEEE TSE* **31** (2005) 1056–1073
13. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* **19:1** (2001) 45–80
14. Easterbrook, S. M.: Requirements engineering. Unpub. manuscript: `http://www.cs.toronto.edu/~sme/papers/2004/FoRE-chapter03-v8.pdf` (2004)
15. Genest, B., Kuske, D., Muscholl, A.: A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *I&C* **204(6)** (2006) 920–956
16. Ghezzi, C., Jazayeri, M., Mandrioli, D.: *Fundamentals of Software Engineering*. 2 edn. Prentice-Hall (2002)
17. Hamon, G., Rushby, J. M.: An operational semantics for Stateflow. In: *FASE*. *LNCS*, Vol. 2984. Springer (2004) 229–243
18. Harel, D.: Can programming be liberated, period? *Computer* **41** (2008) 28–37
19. Harel, D., Marelly, R.: *Come, Let's Play*. Springer (2003)
20. Henriksen, J. G., Mukund, M., Kumar, K. N., Sohoni, M., Thiagarajan, P. S.: A theory of regular MSC languages. *Inf. and Comput.* **202(1)** (2005) 1–38
21. Holzmann, G. J.: The theory and practice of a formal method: Newcore. In: *IFIP Congress (1)*. (1994) 35–44
22. ITU: ITU-TS Recommendation Z.120 (04/04): Message Sequence Chart (2004)
23. Kof, L.: Scenarios: Identifying missing objects and actions by means of computational linguistics. In: *15th IEEE RE*. (2007) 121 – 130
24. Lynch, N.: *Distributed Algorithms*. Morgan Kaufmann (1997)
25. Mäkinen, E., Systä, T.: MAS – An interactive synthesizer to support behavioral modeling in UML. In: *ICSE*. IEEE Computer Society (2001) 15–24
26. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: *ICSE*. ACM (2000) 35–46
27. Pressman, R. S.: *Software Engineering: A Practitioner's Approach*. McGraw-Hill (2004)
28. Royce, W.: Managing the development of large software systems: concepts and techniques. In: *ICSE*. IEEE CS Press (1987) 328–338
29. Sommerville, I.: *Software Engineering*. 8th ed. edn. Addison-Wesley (2006)
30. Tanenbaum, A. S.: *Computer Networks*. Prentice Hall (2002)
31. Uchitel, S., Brunet, G., Chechik, M.: Behaviour model synthesis from properties and scenarios. In: *ICSE*. IEEE Computer Society (2007) 34–43
32. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. *IEEE TSE* **29** (2003) 99–115