Empirical Software Engineering (2021) 26: 124
https://doi.org/10.1007/s10664-021-10015-3

Single-state state machines in model-driven software engineering: an exploratory study

Nan Yang[1] · Pieter Cuijpers[1] · Ramon Schiffelers[1,2] · Johan Lukkien[1] · Alexander Serebrenik[1]

Accepted: 1 July 2021 / Published online: 10 September 2021

Abstract

Context Models, as the main artifact in model-driven engineering, have been extensively used in the area of embedded systems for code generation and verification. One of the most popular behavioral modeling techniques is the state machine. Many state machine modeling guidelines recommend that a state machine should have more than one state in order to be meaningful. However, single-state state machines (SSSMs) violating this recommendation have been used in modeling cases reported in the literature.

Objective We aim for understanding the phenomenon of using SSSMs in practice as understanding why developers violate the modeling guidelines is the first step towards improvement of modeling tools and practice.

Method To study the phenomenon, we conducted an exploratory study which consists of two complementary studies. The first study investigated the prevalence and role of SSSMs in the domain of embedded systems, as well as the reasons why developers use them and their perceived advantages and disadvantages. We employed the sequential explanatory strategy, including repository mining and interview, to study 1500 state machines from 26 components at ASML, a leading company in manufacturing lithography machines from the semiconductor industry. In the second study, we investigated the evolutionary aspects of SSSMs, exploring when SSSMs are introduced to the systems and how developers modify them by mining the largest state-machine-based component from the company.

Results We observe that 25 out of 26 components contain SSSMs. Our interviews suggest that SSSMs are used to interface with the existing code, to deal with tool limitations, to facilitate maintenance and to ease verification. Our study on the evolutionary aspects of SSSMs reveals that the need for SSSMs to deal with tool limitations grew continuously over the years. Moreover, only a minority of SSSMs have been changed between SSSM and multiple-state state machine (MSSM) during their evolution. The most frequent modifications developers made to SSSMs is inserting events with constraints on the execution of the events.

Communicated by: Georgios Gousios and Sarah Nadi

This article belongs to the Topical Collection: *Mining Software Repositories (MSR)*

✉ Nan Yang
n.yang1@tue.nl

Extended author information available on the last page of the article.

Conclusions Based on our results, we provide implications for developers and tool builders. Furthermore, we formulate hypotheses about the effectiveness of SSSMs, the impacts of SSSMs on development, maintenance and verification as well as the evolution of SSSMs.

Keywords Model-driven engineering · Single-state state machines · Modeling practice

1 Introduction

Models play a central role in model-driven engineering (MDE) (Whittle et al. 2014). While models are typically used to facilitate team communication and serve as implementation blueprints, in the area of embedded systems modeling, models have been extensively used for such goals as code generation, simulation, timing analysis and verification (Liebel et al. 2014). One of the most popular modeling techniques used to specify the behavior of software are *state machines*.

Many guidelines have been proposed on how one should model system behavior using state machines (Ambler 2005; Dennis et al. 2009; Prochnow 2008; Schaefer 2006). One of the recommendations commonly repeated both in books (Ambler 2005; de San Pedro and Cortadella 2016; Dennis et al. 2009) and online resources,[1][2] is that a state machine model is only meaningful if it contains more than one state, and if each state represents different behavior. The intuition behind this guideline is that a model should contain non-trivial information, otherwise it merely clutters the presentation of ideas (Ambler 2005). Single-state state machines (SSSMs)—affectionately known as "flowers" due to their graphical representation shown in Fig. 1—violate this recommendation, yet they are known to have been used, e.g., as models of decision making in conversational agents (Kronlid 2006), and in the supervisory control of discrete event systems (Chen and Lafortune 1995). From the growing body of software engineering literature we know that software developers do not always follow recommendations or best practices and often have valid reasons not to do so (Businge et al. 2013; Palomba et al. 2018; Tufano et al. 2017).

We believe that understanding why a widespread recommendation is not followed in practice is the first step towards improvement of modeling tools and practice. In this paper, we extend our previous study on understanding the use of SSSMs in practice (Yang et al. 2020). In our previous study, we conducted an exploratory case study at ASML, the leading manufacturer of lithography machines. We employed the sequential explanatory strategy (Easterbrook et al. 2008). We first mined the archive for 26 components totalling 1500 models to understand the *prevalence of SSSMs* (**RQ1**) as well as *the role played by SSSMs* (**RQ2**). Then we discussed our quantitative findings with software architects to understand *why they opt for SSSMs* (**RQ3**) and *what advantages and disadvantages of SSSMs they perceive* (**RQ4**).

We observed that SSSMs make up 25.3% of the models considered. These SSSMs are often used with other models as *design patterns* to achieve developers' goals. We identified five such design patterns that are repeatedly used in multiple components. The used SSSMs and design patterns provided industrial evidence on how developers deal with existing code base and tool limitations that are the common problems in MDE adoption (Liebel et al. 2014). Given ASML has a large portion of its code base developed with the traditional

[1] GYAN http://gyan.fragnel.ac.in/~surve/OOAD/SCD/SC_Guide.html
[2] https://www.stickyminds.com/article/state-transition-diagrams

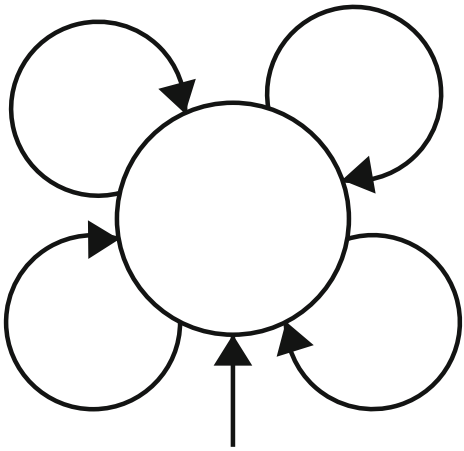


Fig. 1 A flower model (SSSM). The circle represents the single state and the arrows going from and to the same state represent the transitions. The incoming arrow indicates the initial state

software engineering practices, 20.3% of SSSMs are used on boundary of "model world" to interface model-based components with *existing code-based components*. Most SSSMs (64.7%) are used to circumvent the *limitations* of the modeling tools used by ASML, e.g., lack of means to specify data-dependent behavior. As a workaround, developers have to implement the intended behavior with hand-written code. Because of that, the majority of the SSSMs for this purpose is also used on the boundary to interface models with hand-written code *inside* the components. Apart from dealing with the common MDE challenges, around 7.6% of SSSMs are designed to ease long-term *maintenance* of the models. Our interviews also revealed that SSSMs pass verification easily, which is considered as both an advantage and a disadvantage by developers. This implies the trade-off between the effort spent on designing a model that maximizes the advantage of verification and the extra cost caused by downstream problems due to inadequate verification.

Building on our previous study (Yang et al. 2020), we explored how SSSMs evolve in this study (**RQ5**) with the aim of obtaining a complementary view of how developers use SSSMs in practice. Particularly, we investigated for a representative component when SSSMs are introduced in the systems (**RQ5.1**) and how developers modify SSSMs (**RQ5.2**). We answered these questions by mining the historical data of the largest state-machine-based component in the company and manually inspecting the modifications developers made during the evolution of SSSMs. We observed that the SSSMs introduced to ease maintenance and verification appeared in the early phase of component development and their number did not increase over the years. However, over the years more and more SSSMs are needed to deal with tool limitations. Particularly, encapsulating data-dependent behavior implemented with hand-written code is the main reason why developers introduce additional SSSMs in the recent years. This observation suggests that practitioners should thoroughly evaluate the strengths and limitations of modeling tools, taking the future development of their applications into account. Furthermore, we observed that less than 6% of models were changed between SSSM and MSSM during their evolution, implying that most SSSMs are stable. The stability of these SSSMs is also reflected in the number of transitions. SSSMs are more likely to become MSSMs than the other way around. The predominance of evolution from SSSMs to MSSMs can be seen as an example of increasing complexity of a system, suggesting possible applicability of Lehman's laws of software evolution (Lehman 1979) to models operating in a hybrid model/code context, and calling for further research

into this topic. By comparing work-in-progress revisions (available on Git) and integration revisions (available on ClearCase), we observed that developers often have a series of modifications on SSSMs in response to the review that occurs before integration. This indicates that the changes of SSSMs might be driven by peer discussion in the review process, suggesting future research on model review practice. When it comes to the modification developers made to SSSMs, the typical modification we found is adding events with constraints and conditions to the execution order of the events and removing events, as opposed to modifying the execution order of the *existing* events. This observation suggests that the tool builders should consider prioritizing and facilitating the addition and removal of events when designing a user interface.

Based on our results from these two studies, we formulate some implications for developers who would like to adopt state-machine-based solutions, as well as for tool builders and researchers.

The remainder of this paper is organized as follows. Section 2 presents the preliminaries related to this study. In Section 3, we present our study context. In Section 4, we present our first study aimed at understanding the prevalence of SSSMs, the role played by them, why developers use them and the advantage and disadvantage perceived by developers. In Section 5, we present the study of evolution of SSSMs. We discuss threats to validity in Section 6. We then discuss the implications in Section 7. The related work is discussed in Section 8. Finally, the conclusions are presented in Section 9.

2 Preliminaries

We introduce the notion of SSSM and the relevant parts of the tool-chain used at ASML.

2.1 Single-State State Machine

Intuitively, in its simplest form a state machine is a collection of states and transitions between them. Some state machine modeling languages, such as UML state machines, have additional mechanisms (e.g., nested states and state variables) that can represent state information. We exclude the nested states and state variables from consideration as the nested states and the values of state variables can be flattened into simple states (Petrenko et al. 2004; Kim et al. 1999).

In our study, we consider a state machine as a single-state state machine (SSSM) if the state machine has syntactically only one state. We call any other state machine a multi-state state machine (MSSM). For example an MSSM can have more than one state, nested states or make use of state variables.

2.2 A State Machine Modeling Tool: ASD

Analytical Software Design (ASD) is a commercial state machine modeling tool developed by Verum (2014). It provides users with means of designing and verifying the behavior of state machines, and subsequently generating code from the verified state machines.

2.2.1 Model Type and Relation

There are two types of components in a system developed with ASD, namely an ASD component and a foreign component. The ASD components depend on each other in a

Client-Server manner where a client component uses its server components to perform certain tasks. The ASD components consist of *Interface Models* (IM) and *Design Models* (DM) which are specified by means of state machines. The DM implements the internal behavior of a component, specifying how it *uses* its server components. The relation *uses* is realized by three types of events: call event, reply event and notification event (Fig. 2, left). According to the ASD manual, an event is analogous to a method or callback that component exposes. The declaration of a call event contains the event name, parameters and the return type. A call event with a "void" return type has "VoidReply" reply event, while the one with a "valued" return type can use all user-defined reply events. For instance, call event *task([in]p1:string, [out]p2:int):void* is a void type call event with an input and an output parameter. Notification events with output parameters are used to inform clients in synchronous or asynchronous ways, similar to callback functions in such programming languages as C and Python. The IM specifies the external behavior of a component. It prescribes the client components of the ASD component in which order the events can be called and what replies they can expect, i.e., interface protocol. The same IM can be implemented by multiple DMs. In cases such as component reuse, ASD components interact with *foreign components*, non-model components implemented as hand-written code. To support communication between ASD components and foreign components, the external behavior of a foreign component is represented by an IM. Figure 2 (right) shows an ASD-based alarm module where ASD component *Alarm* uses ASD component *Sensor* and a foreign component *Siren*. In the remainder of the paper, we also refer to foreign components as *code-based components*.

2.2.2 Verification and Code Generation

One of the major benefits of using ASD is the possibility to formally verify behavior of the models.

For each component, the type of verification performed by ASD can be summarized into two steps. First, ASD verifies whether each DM has correct behavior, in the sense that its behavior is deterministic and does not contain any deadlocks, or livelocks. It should also not perform illegal sequences of calls. The role of the IM in this check is just to provide the verification tool with information on which calls are considered illegal. For our alarm module example, ASD checks whether DM *Alarm* calls occur in the order specified in IMs *ISensor* and *ISiren*. Second, ASD verifies whether the DM of a component, together with the

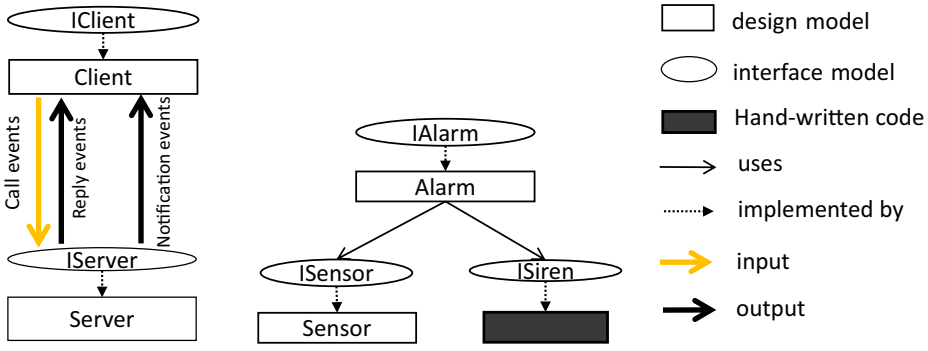


Fig. 2 Model relations. Left: type of events. Right: example of an ASD module . I*** stands for an IM

interfaces of its servers, correctly refines the IM of this component. It does this by constructing a so-called Failures-Divergence Refinement relation (FDR) (Fdr homepage 2014) between the DM and IM. Verifying this refinement guarantees that the IM can be used as an abstract representation of the DMs behavior in further analysis of the system. For our alarm module example, ASD verifies whether DM *Alarm*, together with IMs *ISensor* and *ISiren* refines IM *IAlarm* correctly. Code in the selected target language (e.g., C++) can be automatically generated once the system is free of behavioral errors.

Note that the IM and DM have different roles, not only in system modeling but also in the verification and code generation. The IM provides an abstract view of the behavior of a component while DM provides a detail view. Both IM and DM are used to understand software, communicate between engineers, and verify the behavioral correctness. However, only the DM contains the implementation details that are used to generate code.

3 Study Context

To get a deeper understanding of the use of SSSMs in embedded systems industry, we conducted an exploratory case study. that consists of two complementary studies present in Sections 4 and 5. Case study is an empirical method aimed at investigating contemporary phenomena in a context (Runeson and Höst 2009; Yin 1994).

We follow the recommendation of Runeson and Höst and intentionally select a case of analysis to serve the study purpose (Runeson and Höst 2009). We conduct our exploratory case study at ASML. The company uses the commercial state machine modeling toolchain Analytical Software Design (ASD) developed by Verum (Verum 2014), described in Section 2.2, to develop the control software of their embedded systems, providing a paradigmatic context to our study. The company uses ASD to design and verify the behavior of state machines, and subsequently generate code from the verified state machines.

We obtain all components developed with ASD in the system, except for those that are not accessible due to international legislation or contain strategic intellectual property. These 26 components are continuously maintained; code generated based on these models runs on the machines produced by ASML. Each component is formed by multiple interacting IMs and DMs. In total, we obtain 924 IMs and 576 DMs, with the number of IMs per component ranging from 2 to 349, and DMs from 0 to 284. Table 1 gives an overview of the 26 components. For the sake of confidentiality, we refer to these components as A, ..., Z and cannot share the models. Note that, other than these 26, components developed with traditional software engineering still make a large portion of the software system of the machines. Therefore, these 26 components have to interact with the existing code-based components.

4 Understanding the Use of SSSMs (Yang et al. 2020)

In this section, we present our previous study that investigated the prevalence of SSSMs (**RQ1**), the role SSSMs play (**RQ2**), the reason why developers use them (**RQ3**) and the advantages and disadvantages of them perceived by developers (**RQ4**) (Yang et al. 2020). In Section 5 we extend this study to explore the evolutionary aspects of SSSM. We present our method in Section 4.1 The results for **RQ1-4** are presented in Sections 4.2, 4.3 and 4.4. We then discussed threats to validity in Section 6.

Table 1 Overview, prevalence of SSSM and frequency of the identified terms for the selected state machine based projects

Component ID	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Overview																										
#IMs	19	349	22	98	15	6	22	10	10	12	29	12	29	43	3	12	16	3	41	15	49	11	2	77	3	16
#DMs	9	284	10	72	6	3	9	4	3	6	11	4	11	9	0	6	6	2	17	13	31	3	1	46	1	9
Total	28	633	32	170	21	9	31	14	13	18	40	16	40	52	3	18	22	5	58	28	80	14	3	123	4	25
Prevalence of SSSM																										
#SSSM-IMs	1	109	9	42	14	3	10	6	8	6	17	8	14	27	2	5	13	0	15	1	18	8	2	11	2	3
%SSSM-IMs	*5*	*31*	*41*	*43*	*93*	*50*	*45*	*60*	*80*	*50*	*59*	*67*	*48*	*63*	*67*	*42*	*81*	*0*	*37*	*7*	*37*	*73*	*100*	*14*	*67*	*19*
#SSSM-DMs	0	11	0	4	0	0	0	0	0	1	1	2	2	1	0	1	1	0	1	0	1	0	0	0	0	0
%SSSM-DMs	*0*	*4*	*0*	*6*	*0*	*0*	*0*	*0*	*0*	*17*	*9*	*50*	*18*	*11*	*-*	*17*	*17*	*0*	*6*	*0*	*3*	*0*	*0*	*0*	*0*	*0*
Frequency of the identified terms																										
#Exclusive	1	50	8	24	20	4	12	7	14	2	16	11	21	27	3	7	21	0	9	3	16	8	4	11	3	2
#Exclusive & Frequent	0	0	0	3	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
#Shared	0	73	3	22	2	1	2	4	4	9	8	3	6	19	0	4	3	0	12	1	13	5	0	9	1	3
#Shared & OR>1	0	45	2	14	0	0	1	0	0	5	3	1	3	5	0	1	0	0	3	1	7	1	0	8	0	3
#Shared & OR>1 & Frequent	0	14	0	3	0	0	1	0	0	0	3	0	0	4	0	0	0	0	1	0	2	1	0	2	0	0

-indicates that the percentage cannot be computed as the component does not include DMs

4.1 Methods

We employed sequential explanatory strategy which consists of a quantitative phase and a qualitative phase (Easterbrook et al. 2008). Figure 3 gives a high-level overview of our research method. To answer **RQ1**, we study the prevalence of SSSMs by analysing models of the 26 components. To answer **RQ2**, i.e., to understand the role played SSSMs we combine two complementary approaches. On the one hand, according to Wittgenstein (Wittgenstein 2009), the meaning is determined by use. Thus we exploit structural dependencies (cf. (Antoniol et al. 1998; Dong et al. 2007)) to identify the *implemented by* and *uses* relations between IMs and DMs, i.e., the use of models. On the other hand, we expect the role of the SSSM to be reflected in its name, in the same way the names of objects have been extensively used to uncover the responsibilities of software objects (Garcia et al. 2013; Nurwidyantoro et al. 2019; Kuhn et al. 2007).

In the qualitative phase, we conduct a series of interviews to answer **RQ3** and **RQ4**. The interviews were recorded and audio was transcribed. To derive and refine the theory based on the obtained qualitative data, we employ Straussian grounded theory because it allows us to ask under what conditions a phenomenon occurs (Stol et al. 2016). We opt for an iterative process to reach the saturation. It is important to note that in the sequential explanatory strategy the results from the quantitative phase is used to inform the subsequent qualitative phase. This means the concrete study design for **RQ3** and **RQ4**, e.g., the interview questions, is determined by the results of **RQ1** and **RQ2**. For example, depending on the number of identified SSSMs, we opt for different interview strategies; if the number of SSSMs will be small enough then we can request the experts to explain the reasons behind every SSSM. Otherwise, we need to prompt the discussion based on the findings we obtained from the analysis of structural dependencies and names. We detail the procedures of the qualitative phase in Section 4.4.1.

4.2 Prevalence Analysis (RQ1)

We answer **RQ1** by analysing the frequency of SSSMs in the 26 components in Table 1.

4.2.1 Data Analysis

We analyse 1500 ASD models corresponding to components A–Z. We first convert each model into an Ecore model (Steinberg et al. 2008) using a tool developed by ASML. The conversion process is lossless, i.e., the Ecore models can be converted back to the original ASD models. We then use EMF Model Analysis tool (EMMA) (Mengerink et al. 2017) to measure the number of states #*state* and the number of state variables #*sv*. An SSSM is a model with #*state* $= 1$ and #*sv* $= 0$.

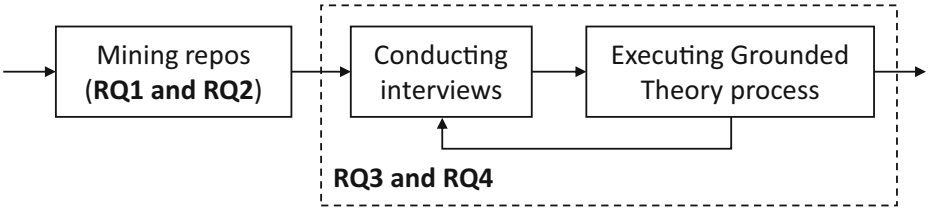


Fig. 3 Overview of our research methods

4.2.2 Results

Table 1 shows the prevalence of SSSMs in the 26 components. 25 out of 26 components contain SSSMs, making up 25.3% of the 1500 state machines. Component B is the largest component among the 26 components we consider. In component B 31% of IMs are SSSMs while only 4% of DMs.

This tendency for using SSSMs mainly for IMs can also be observed in smaller components. In 13 out of 26 components more than 50% of IMs are modeled as an SSSM. On the contrary, only 26 SSSM-DMs are present, and they are present in 11 out of 26 components. Furthermore, although SSSMs are generally popular among IMs, different components show different degrees of usage; SSSMs make up more than 70% of IMs in components E, I, Q, V and W while less than 10% in components A, R and T.

> **RQ1 summary:** Developers tend to use SSSMs mainly for modeling IMs. The use of SSSMs differs between the components: component B has the largest portion of SSSM-IMs.

4.3 Role of SSSMs (RQ2)

Since SSSM-IMs are the lion's share of SSSMs, when answering **RQ2**, **RQ3** and **RQ4** we focus exclusively on SSSM-IMs. We start with data collection of structural relations between models and the names of models, followed by an analysis of results.

4.3.1 Data Analysis

To study what roles the SSSM-IMs play, we split IMs into three mutually exclusive locations, namely:

1. **disconnected (disc):** IMs that are neither implemented nor used by a DM.
2. **boundary (bd):** IMs that are used by at least one DM but not implemented by any DMs, or IMs that are implemented by at least one DM but not used by any DMs. They are on the boundary of "model world" independent from whether code is present on the other side of the boundary.
3. **non-boundary (nb):** IMs that are implemented by at least one DM and used by at least one DM.

We use EMMA (Mengerink et al. 2017) to extract structural relations *implemented by* and *uses* from models, and classify IMs based on these three locations.

To get complementary insights, we analyse names of models. We follow commonly used preprocessing steps (cf. (Thomas et al. 2014)) including tokenization based on common naming conventions such as under_scores, camelCase and PascalCase (syntok 2014), stemming (Wiese et al. 2011) and removal of stop words and digits using the NLTK package (Tookkit 2014). We also observe that the names often contain abbreviations with the sequence of capitals, e.g., *IOStream*. Hence, prior to tokenization we manually collect a set of abbreviations from the names, compute how frequently they are used per model and remove them from the names. As a result, for each component we obtain two document-term matrices with models acting as documents. The matrices describe the frequency of terms (including the abbreviations) that occur in a collection of the names of SSSM-IMs and MSSM-IMs, respectively.

We conjecture that the terms appearing in the SSSM-IM set while not in the MSSM-IM set (*Exclusive*), and the terms that appear in both sets (*Shared*) with high frequency in the SSSM-IM set might suggest the role of SSSM-IMs. Therefore, for each component we further obtain the sets of *Exclusive* and *Shared* terms. To identify the "most important" shared terms we compute the odds ratio of each term, i.e., ratio of the share of SSSM-IMs containing term *t* and the share of MSSM-IMs containing term *t*.

4.3.2 Results

Table 2 is a contingency table showing how many SSSM-IMs and MSSM-IMs fall into each location group. We observe that overall *bd*-models are more likely to be SSSM, while *nb*-models are more likely to be MSSM.

However, such an overall assessment might obscure differences between the components, in particular since component B is much larger than the remaining components. Hence, per component we apply statistical techniques to determine whether for an IM being an SSSM depends on the location group it belongs to. Since only component B has disconnected models, we exclude *disc* from the statistical analysis. For each component, we construct a 2×2 contingency table recording the number of SSSM-IMs and MSSM-IMs for each location. To analyse the contingency tables we opt for Fisher's exact test (Fisher 1922) rather than a more common χ^2 test: indeed, many components have few IMs and the normal approximation used by χ^2 requires at least five models in each group, i.e., at least 20 IMs per component. The null hypothesis of Fisher's exact test is that the type of IM (SSSM vs. MSSM) is independent of its location (*bd* vs. *nb*). Figure 4 shows the p-values obtained: for 9 out of 26 components the p-value is smaller than the customary threshold of 0.05 and the odds ratio (i.e., the ratio of the share of SSSM-IMs from boundary and the share of MSSM-IMs from boundary) is larger than one. This means that we can reject the null hypothesis for these 9 components, i.e., the type of IM depends on whether it is on the boundary of the "model world". We also observe that the components where the null hypothesis can be rejected tend to have more IMs than those where the null hypothesis cannot be rejected.

Next, we identify the terms frequently used in names of the IMs. In total, we obtain 472 terms from the names of IMs for components A–Z. Table 1 gives an overview of the number of *Exclusive* terms, the number of *Exclusive* terms with more than five occurrences (*Exclusive*&*Frequent*), the number of *Shared* terms, and the number of *Shared* terms with an odds ratio larger than one (*Shared*&*OR*>*1*), as well as the number of *Shared* terms with frequencies higher than five and an odds ratio larger than one (*Shared*&*OR*> 1&*Frequent*).

We observe that some terms are exclusively used in SSSM-IMs. However, only components D, K, N and S contain exclusive terms with more than five occurrences as shown in Table 1. The three such terms in component D are "data", "foreign" and "barrier". Components K, N and S have one such term: "access". Based on this observation, we conjecture that developers might think SSSMs particularly suit a certain functionality related to "data",

Table 2 Number of SSSM and MSSM per location

	SSSM-IM	MSSM-IM	Total
disc	3	0	3
bd	266	195	461
nb	85	375	460
Total	354	570	924

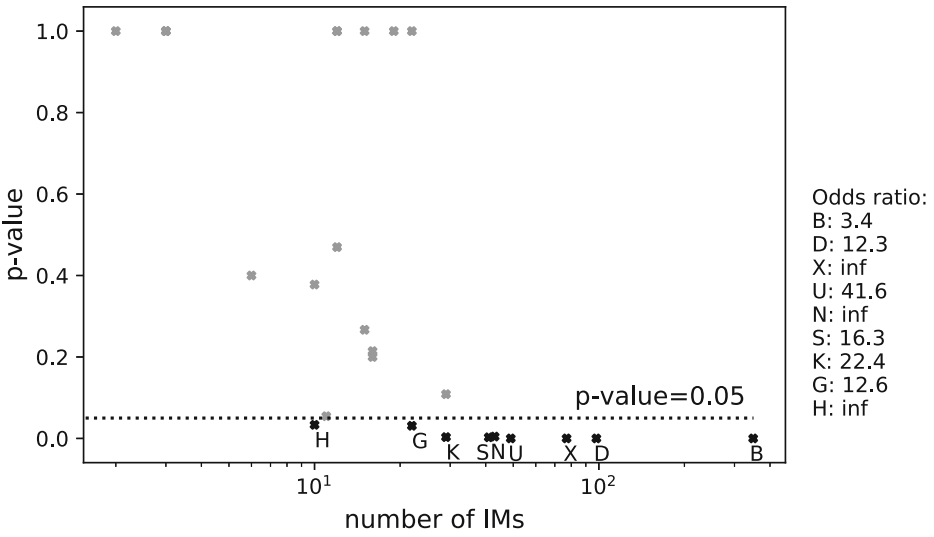


Fig. 4 *p*-values of the Fisher's exact test vs. number of IMs: the null hypothesis is more likely to be rejected for components with more IMs and the odds ratio for each rejected case is larger than one

"foreign", "barrier" and "access". We do not further investigate the low-frequency *Exclusive* terms because we expect them to be less likely to disclose the common roles SSSMs play.

Out of the 26 components, 22 have terms shared in SSSM-IMs and MSSM-IMs. 15 components have shared terms with an odds ratio larger than one, i.e., the models containing the term in their names are more likely to be SSSMs. As shown in Table 1, such terms are frequent in nine components. For component B Fig. 5 shows frequently occurring shared terms with an odds ratio greater than one. We anonymize the domain-specific terms and refer to them as t1,...,t5 for confidentiality reasons. Term "foreign" belongs to group *Shared&OR*$>$ 1&*Frequent* in component B but to group *Exclusive&Frequent* in component D. This suggests that the roles reflected by the same term might be implemented differently

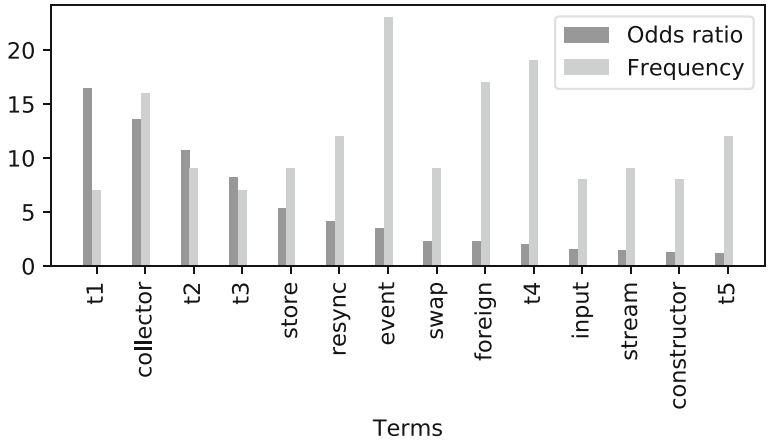


Fig. 5 Frequency and odds ratio of terms that belong to *Shared&OR*$>$ 1&*Frequent* for component B

in different projects. Moreover, it seems that domain-specific terms are very important as they are topping the odds-ratio list.

In other eight components that have a non-empty group *Shared&OR*> 1&*Frequent*, there are in total nine domain-specific terms identified as t6,...,t14 and five non-domain-specific terms "error", "servic", "seqenc", "measur" and "data". The terms from groups *Exclusive&Frequent* and *Shared&OR*> 1&*Frequent*, and the corresponding occurrences in the names of the SSSM-IMs from the 26 components are summarized in Table 3. These are the terms repeatably used in the names of SSSM-IMs.

> **RQ2 summary:** For larger components developers use SSSMs particularly often on their boundary. Furthermore, developers repeatedly prefer terms such as "data" in the names of the SSSM-IMs.

We conjecture that terms in Table 3 encode the reasons why developers use SSSM-IMs and use these terms to prompt discussion in the follow-up interviews.

4.4 Interview (RQ3 and RQ4)

4.4.1 Procedure

Following the sequential explanatory research strategy, we refine the concrete steps for the qualitative phase based on the outcomes of the quantitative phase.

Iterative process We start the process by considering the largest component (component B) as we expect it to produce the richest theory. We conduct semi-structured interviews with architects of the component under consideration, perform open coding of the interview transcripts to derive categories of SSSM-IMs, perform member check to mitigate the threat of misinterpretation (Buchbinder 2011), and label the SSSM-IMs in all components using the categories derived. If at this stage all SSSM-IMs have been labeled, saturation has been reached and the process terminates. Otherwise, we select a not yet considered component with the largest number of unlabeled SSSM-IMs and iterate. Figure 6 summarizes the process we follow.

Interview design The interview questions stem from the quantitative findings. First of all, reflecting on the findings for **RQ2** we ask *why do developers use SSSMs more often on the boundary of the "model world" than in other parts?* To discuss the goals of using disconnected, boundary and non-boundary SSSM-IMs, we provide a list of SSSM-IMs for each location and ask: *what goals do you intend to achieve with an SSSM-IM in disconnected/boundary/non-boundary parts?* Next, for each term identified either as *Exclusive&Frequent* or as *Shared&OR*> 1&*Frequent*, we provide a list of SSSM-IMs containing

Table 3 Terms that belong to groups *Exclusive&Frequent* and *Shared&OR*> 1&*Frequent* and the number of SSSM-IMs that contains the term

Term	collector	store	resync	event	swap	foreign	input	stream	constructor
#SSSM-IMs	16	9	12	23	9	26	8	9	8
Term	barrier	data	error	servic	access	sequenc	measur	t1,...,t14	
#SSSM-IMs	10	34	17	8	22	8	10	141	

the term and ask two questions: *what responsibilities does the term imply?* and *why do you use SSSMs to implement these responsibilities?* To obtain as rich information as possible, we send a list of SSSM-IMs to our interviewees before the interviews, allowing them to refamiliarize themselves with the models. We do not disclose the interview questions prior to the interview. To answer **RQ4**, we ask developers about *advantages of using single-state state machines* and the disadvantages. We have the interviews in a meeting room with a whiteboard. Interviewees can draw on the whiteboard for explanation. We take photos of the whiteboard after interviews.

Coding procedures After initial interviews, we conduct open coding on the interview transcripts, identifying the goals that developers attempt to achieve, the solutions they employ and the location of the used SSSM-IMs (boundary/non-boundary/disconnected). For example, when we ask questions about term "foreign", we obtain the following answer: "We want to create formal models that is why we use ASD. The problem here is the outside world is not formal. So it can behave as expected or unexpected, we don't know ... If people follow the rules, all boundaries need to be armored. The important aspect is that the calls from foreign side must be accepted by every state. As foreign IM, you cannot restrict anything because you don't know the behavior of foreign (components)". Based on this answer we identify the developers' goal as protecting formal models from informal and unknown foreign behavior, the solution they employ should not restrict the order of events from foreign side, and the location of the SSSM-IM is boundary.

The solution is augmented by details with photos that we took from the whiteboard. We refer to the detailed solution as *design pattern*. Each design pattern can be 1) an *SSSM-IM*, 2) a *combination* of an SSSM-IM and the DM(s) that implement it, or 3) a *set* of SSSM-IMs and other models. The open coding process results in a set of categories that consist of *goals*, *locations* and *design patterns*. For instance, category *armoring the boundaries of models* emerges from the previous example. Next, we perform axial coding to group these categories based on the *core reason* behind, i.e., why developers would like to achieve the goal? For instance, the core reason behind category *armoring the boundaries of models* is that models have to work with the existing code base. In addition, we also identify the advantages and disadvantages from our interviewees' answers.

Member check The first author conducts the coding tasks. In order to ensure that the categories are correctly identified, we perform member check (Buchbinder 2011) with our interviewees. The member check is a validation activity that requests informant feedback to improve the accuracy of the derived the theory. This resulting adjustment on categories is represented by the dashed line in Fig. 6.

Label SSSM-IMs The first author reviews each SSSM-IM and labels it based on the derived categories. For instance, we can determine whether a model is an instance of category *armoring the boundaries of models* by checking if it is on boundary and implements the design pattern we identified for this category.

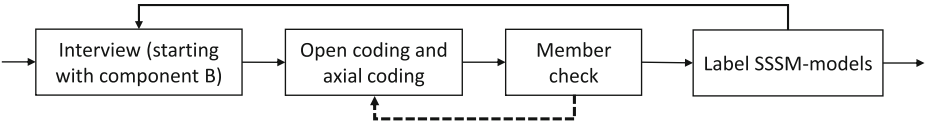


Fig. 6 Steps in the qualitative phase

4.4.2 Reasons of Using SSSM-IMs (RQ3)

We reach saturation with three face-to-face interviews and two interviews through emails. Table 4 provides an overview of our results. We identify four core reasons why developers use SSSM-IMs: 1) using models together with *existing code base*, 2) dealing with *tool limitations*, 3) facilitating *maintenance* and 4) easing *verification*. For each core reason, developers have at least one goal to achieve with SSSM-IMs. 353 out of 354 SSSM-IMs can be explained by the core reasons and goals listed in Table 4. Before discussing Table 4, we briefly review the model that cannot be explained by it. It is a disconnected SSSM-IM that should have been removed once it was no longer used ("dead code").

Table 4 Why developers use SSSM-IMs identified from the 26 components: the core reason, goal, location, design pattern and the number of instances (SSSM-IMs)

Core reason		Goal	Location	Design pattern	#instances
Existing code base		*ModelArmor:* protecting verified behavior from non-verified behavior	boundary	D1	77
Tool limitations	Unable to specify data-dependent behavior	*DataEncapulation:* encapsulating data-dependent behavior into functions	boundary and non-boundary	D2	183
	Unable to select a subset of notification events	*EventCollector:* specifying individual interest for multiple clients	boundary	D3	30
	Lack of common libraries	*LibraryReuse:* reusing libraries available in general-purpose programming languages	boundary	An SSSM-IM	31
	Unable to specify global literal values	*GlobalLiteralValue:* specifying global literal values	non-boundary	Combination	2
Maintenance		*CallMapping*: reducing coupling between clients and servers	non-boundary	D4	16
		FeatureSelection: isolating product-specific features from common features	non-boundary	D5	9
		EaseRefactoring: easing event renaming	non-boundary	-	2
		Documentation: documenting events for communication within teams	disconnected	An SSSM-IM	2
Verification		*EaseVerfication:* avoiding a large state space	non-boundary	An SSSM-IM	1

We refer the design patterns that involve a set of models to D1,...,D5 as shown in Fig. 7. For the sake of generalizability, we do not explain the design pattern that is used to achieve goal *EaseRefactoring* because it is specific to the semantics of the modeling language provided by ASD suite

In the remainder of this section we discuss the reasons, goals and design patterns shown in Table 4.

4.4.3 Using Models together with Existing Code Base

As mentioned, a large portion of software base was developed with the traditional software engineering methods. Hence, the model-based components need to interact with the existing code-based components. The behavior of the models is formally verified and can only interact with each other according to the protocol specified in the IMs. By nature, when communicating with foreign components, model-based components operate under the assumption that foreign components behave as specified. However, due to the lack of formal specification, the behavior of code-based components is not formally verified and often unknown. This means that developers need a mechanism to "protect" models from non-verified and unexpected behavior of code-based components.

To achieve the goal, developers come up design pattern D1 shown in Fig. 7. The core idea of this pattern is to create a layer which accepts any order of calls from the code side at first and then only forwards the allowed order of the calls to the model side. By implementing this idea, both code-based components and model-based components are not aware of the presence of each other.

Next we discuss how the elements in the pattern work together. Developers would like to protect *Core* which is a group of models from the non-verified of code-based components *Foreign Client* and *Foreign Server*. IMs *IForeign* are SSSM-models which allow any order of input events while DMs *Armor* forward the allowed calls specified in IMs *IProtocol* which describes the order of events expected by *Core*. In order to trace the unexpected behavior from *Foreign Client* and *Foreign Server*, DMs *Armor* also record protocol deviations with *Logger* so that it is easier to distinguish failures caused by protocol violations from failures caused by functional errors.

4.4.4 Dealing with tool limitations

ASD suite has several limitations preventing developers from specifying the intended behavior of models. As workarounds, developers have to manually implement the behavior with general-purpose programming languages. This also results in the use of code between models inside a model-based component and raises the need of interfacing with the code.

DataEncapsulation One of the limitations of ASD suite, is the lack of a way to specify data-dependent behavior: one can declare parameters for the events in models to pass data transparently from one model to the other but the control decision cannot be made based on a parameter value.[3] The pass-by data eventually ends up in code where the data-dependent behavior can be programmed. To work around this limitation, developers store and manage data in hand-written code known as *data stores* inside the model-based components. The developers' goal is to have a mechanism allowing the models to read and write each piece of data. Design pattern D2 in Fig. 7 is used to achieve the goal.

In the system under study, each piece of data in a data store is associated with an ID. For the sake of example, assume that a control decision has to be made based on the comparison of two data values associated with ID *d1* persistently stored in *DataStore1* and *DataStore2*

[3]This limitation is intentional in order to avoid the state space explosion problem.

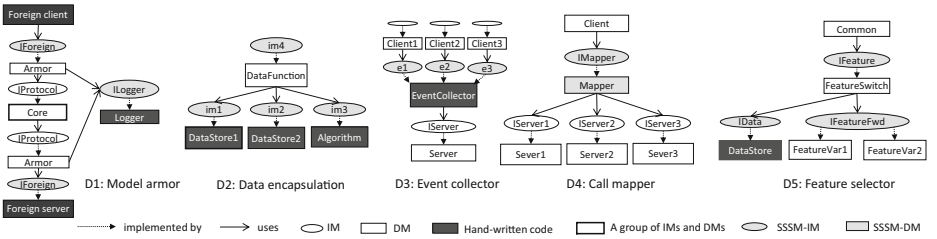


Fig. 7 Identified design patterns D1,...,D5

respectively. Because models can only pass data transparently, there is a need to implement hand-written code known as *Algorithm* which offers call events triggering the comparison task, and returns reply events that inform about the result. To obtain the control decision based on the comparison, DM *DataFunction* is used to fetches the data corresponding to *d1* from *DataStore1* and *DataStore2*. Then it passes the fetched data to *Algorithm* to obtain the result.

Based on the received reply, *DataFunction* synchronously returns a reply to the client models that ask for a decision. For complex applications, *DataFunction* needs to intensively interact with data stores and *Algorithm* in order to derive results. To reduce the coupling between data-aware code and data-independent models, IM *im4* is an SSSM which only specifies the call events and the possible replies so that the underlying data-related interactions between code and *DataFunction* are hidden from the models that only expect a decision. Similar to IM *im4*, IM *im3* only specifies the signatures of independent functions implemented with code.

When it comes to data access, a write operation for data associated with a specific ID is required to be performed before a read operation for the corresponding data. Naturally, developers would like to specify the required order in IMs *im1* and *im2* so that the interaction protocol between *DataFunction* and these IMs is explicitly defined, and subsequently verified before code generation. However, since data-dependent behavior is not supported by ASD, *im1* and *im2* are SSSMs which only specify the signatures of call events and replies for the intended data operations. The interaction protocol, in this case, is implicitly encoded in code for these data stores, requiring test efforts to examine correctness.

EventCollector Another tool limitation that influences how developers design software is that client models cannot select a subset of notification events to receive from their server models. This means that the client models have to receive *all* notification events from their server models even though some of notification events are out of their interest. To model a case where multiple client models are interested in different subsets of notification events from the same server model, design pattern D3 in Fig. 7 is used. Instead of interfacing with the server model directly, clients interface with a hand-written *EventCollector* which works as a router forwarding each notification event to the corresponding client according to the events that developers specify with SSSM-IMs *e1*,*e2* and *e3*. Because each DM can only implement one IM developers have to inject the hand-written router between models.

LibraryReuse ASD suite provides reusable libraries, such as a timer, implemented by models that can be used across different applications. However, the available libraries are limited compared to their counterparts available for general-purpose programming languages. For instance, one of missing libraries is timestamp library. As a workaround, developers use

hand-written code to wrap the timestamp-related operations (e.g., converting timestamp format) into functions with output parameters (e.g., for obtaining converted timestamp). The SSSM-IMs specify the signatures of the hand-written functions so that the generated code from the models can seamlessly reuse these libraries.

GlobalLiteralValue Since ASD suite does not provide means of specifying global constants as most programming languages have, developers have to use the actual literal values wherever they need them. For example, assume that we would like to use a global constant *Size* to store the value of the buffer size set to *100*. To avoid the errors that could be introduced by hard-coding this value, developers implement SSSM-IMs and SSSM-DMs to store the value which can be obtained by calling corresponding events. Developers specify an SSSM-IM that offers call event *getBufferSize([out]p:int):void*. In the corresponding SSSM-DM, the call is augmented with the corresponding output integer,i.e., *getBufferSize(100)*. In this case, by calling event *getBufferSize(n)*, other models that need the value can obtain variable *n* that holds integer *100*.

4.4.5 Facilitating Maintenance

In four cases, SSSM-IMs are used to facilitate maintenance.

CallMapping Client models often need to call a sequence of events on different server models. To reduce the coupling between the client model and its server models, developers implement a mapper which consists of an SSSM-IM and an SSSM-DM between the client and its servers (see D4 in Fig. 7). The SSSM-IM only specifies the signature of a void call event that can be triggered by the client model. The mapping of the call event triggered by the client model to a sequence of intended call events on other server models is specified in the corresponding SSSM-DM.

FeatureSelection As the system under study is specified using principle from software product line engineering, developers separate features shared by all products from product-specific features to be configured at runtime (Capilla et al. 2014). D5 in Fig. 7 shows a design pattern supporting this separation. For the sake of an example, assume a system needs to construct different sequences of actions for the same task based on the runtime configuration of the product type. For each product, the sequence construction is triggered by the same call event *Construct*. To hide the product-specific details from the common models, *IFeatureFwd* specifies the signature of *Construct* which is implemented by *FeatureVar1* and *FeatureVar2*. *Common*, as the common feature shared by all products, needs to call *Construct* to trigger the sequence construction on the correct variant based on the runtime configuration. However, involving *Common* in this feature selection breaks the separation of concerns, i.e., *Common* has to be aware of that different products exist. To avoid this, *FeatureSwitch* is implemented. At runtime *FeatureSwitch* reads the product type from a data store and forwards *Construct* to the appropriate product-specific implementation (i.e., *FeatureVar1* or *FeatureVar2*).

Since *IFeature* has to hide the feature selection and product-specific details from *Common*, it is identical to *IFeatureFwd* acting as an interface offering *Construct*. When *Common* calls *Construct*, the feature selection is performed, followed by the sequence construction based on the selection. *Common* is, hence, not aware of any product-specific information. Developers expect that by using this pattern the coupling between common parts and

product-specific parts can be reduced and the variants can be extended without modifying the common parts.

EaseRefactoring Developers also consider the ease of refactoring. Assume a model repeatedly triggers a task implemented by a sequence of *e1*,..., *e8*. Hard-coding this sequence at several invocation sites is error-prone. Moreover, any change to the sequence such as renaming an event, has to be performed at all invocation sites. Hence, developers use a solution akin to procedure abstraction to specify a sequence of events only once and reuse it wherever needed. Since the concrete solution is specific to the semantics of ASD, we do not disclose further details.

Documentation IMs are sometimes used to document the signatures of functions. In such cases, developers use disconnected SSSMs to communicate the design.

4.4.6 Easing Verification

The efficiency of verification is another concern in modeling. Prior to the verification step typically carried out by a model checker, the tool-chains need to convert state machine specifications into a model checker formalism which represents the state space of the models. Behavioral correctness of models with a large state space takes a lot of time to verify. Hence, the verification step slows down the design and maintenance of the models. In our case study, we found a situation where an SSSM-IM is used to avoid verification on a large state space.

The intention of the developers was to create an interface such that the number of triggers on event *a* should be larger than the number of triggers on event *b*. The corresponding state space contains all possible combinations such that *a* is triggered exactly one more time than *b*, two more times, etc. During the verification step, the model checker has to visit every single state in the state space. To ease the verification step, developers simplify the model to an SSSM with events *a* and *b*, dropping the requirement that the number of triggers on event *a* should be larger than the number of triggers on event *b*: "Scalability is a good reason to not verify this explicitly, as it does not matter if the max difference between #a - #b is 1, 2, 9 or 100. Abstracting from the exact difference makes the verification scalable, at the cost of less guaranteed correctness.

> **RQ3 summary:** Developers use SSSMs for four reasons: 20.3% of SSSMs are used to interface models with the *existing code base*; 64.7% of SSSMs—for dealing with *tool limitations* such as incapability of specifying data-dependent behavior. Around 7.6% of SSSMs are designed for the purpose of easing long-term *maintenance*. The last concern pertains to *verification* efficiency. To achieve their goals, developers often use SSSMs together with other models as *design patterns*.

4.4.7 (Dis)advantages of SSSM-IMs (RQ4)

When it comes to the advantages and disadvantages of using SSSM-IMs, the interviewees share the same opinion. The main perceived advantage of SSSMs is the *ease* of verification: "The main advantage is that a flower model is stateless, it imposes no restrictions so verification passes easily and perhaps more importantly: it is easier to implement a Foreign

component faithfully". Moreover, since SSSM-IMs impose no restrictions on the order of events, changes to the calling order on the client side also easily pass the verification, reducing the maintenance effort. However, the ease of verification also means that the model *"will likely always pass verification"* hiding potential bugs and compromising potential verification benefits. Taking both the advantage and the disadvantage of SSSM-IMs into account interviewees recommend caution when using SSSM-IMs: *"people (developers) need to have a very good reason for it because it does not check anything"*. Furthermore, according to the observations of the interviewed architects, it usually takes a lot of time for developers to learn *how to design models in a way that development, maintenance and verification can be facilitated.*

RQ4 summary: The property of SSSMs—passing verification easily—is perceived as an advantage for easier development and maintenance but also a disadvantage that might hide bugs.

5 Evolution of SSSMs

As discussed in Section 4, SSSMs are widely used in different components for various reasons, although the widespread modeling guidelines suggest not to use them. Our discussion with developers implies that SSSMs can pass verification easily, which may ease the development but also potentially hide defects. However, it is unknown yet when SSSMs are introduced in the components, and whether and how they have been modified by the developers. Understanding the life-cycle of SSSMs and the actions taken by developers to modify them can help us better understand the phenomenon of how developers use SSSMs in practice, and provide suggestions to researchers and tool builders. Therefore, to obtain a complementary view of under what circumstances SSSMs are being used, we analyze the evolution of SSSMs in the change histories of software components. Specifically, we posed the following questions:

RQ 5.1 *When were SSSMs introduced?* This question aims at understanding when the need for SSSMs occurs. Specifically, we study whether SSSMs were introduced as soon as the development starts or whether they surged into systems due to certain maintenance needs. To this aim, we investigated the trends in the history of the SSSMs.

RQ 5.2 *How do developers modify SSSMs?* With this question, we aim to understand whether and how developers modify SSSMs. We conjecture that there might be several evolutionary scenarios; developers might only add or remove the transitions of SSSMs, add or remove states or combinations of them. In particular, we study following questions:

- **5.2a** Do SSSMs become MSSMs, and vice versa?
- **5.2b** Do developers modify transitions of models that stay SSSM throughout their entire history?
- **5.2c** What modifications are involved when SSSMs are modified to become MSSMs and when MSSMs are modified to become SSSMs?

We answer these questions by mining model repositories and manually categorizing the changes that developers made to SSSMs.

5.1 Study Subject

To study the evolution of SSSM, we examined the availability of the historical data for 26 components from Table 1. After an investigation, we selected component B as our study subject. The decision is made because other components have little historical data available. The lack of historical data is attributed to the way developers version their models and infrequent modifications requested by customers. Next, we elaborate on these two reasons.

Currently, the company uses two ways of working, Git-based and Break-Out-Archive (BOA)-based, illustrated in Fig. 8. Both ways of working combine two types of version control systems: Git and IBM Rational ClearCase. The component developed with Git-based way of working has a dedicated Git repository that tracks revisions made by developers. When a certain feature of the component is finished and verified using ASD, developers submit the snapshot[4] to the ClearCase repository of the component. This snapshot is then integrated with the rest of the system. Differently, the component developed with BOA-based way of working does not have a dedicated Git repository. When one or more such components needs to be modified, developers create a new Git repository and import a snapshot of all the relevant components. Once the modification is finished, developers submit the snapshot to the ClearCase repository and abandon the Git repository. For the 26 components that we listed in Table 1, three components (B, C, and D) are developed with the Git-based way of working; the other components are developed with the BOA-based way of working. Only very few revisions (less than five) are available on ClearCase for these BOA-based components. We confirmed our observation with the developers who are reponsible for these components. Indeed, some components do not evolve, as stated in one of the replies: "we basically only have a single version created when the model was first introduced." We therefore further investigated the components that were developed with Git-based method (i.e., components B, C, and D), by collecting the revisions from the master branch of their Git repositories and from the integration stream of their ClearCase repositories.

Table 5 shows the number of model revisions and the average number of revisions (per model) available from ClearCase, as well as for the data available from the Git repositories. Considering the average number of revisions, models from components C and D have only few revisions available for each model from their ClearCase and Git repositories. We confirmed this information with the developers responsible for these components: "Component D is running at the customer for quite some time. No issues so far, so that's why it doesn't have many versions".

It can be seen that component B has the largest (average) number of revisions available because it has the longest maintenance history and it is the first ASD-based component in the company. Our previous study (Section 4) has shown that studying component B as the first step is an efficient way of deriving a theory that can be applied to other components. Based on our observations, we decided to conduct an exploratory study with component B.

5.2 Data Collection and Analysis

To answer **RQ 5.1**, we collected the snapshots from the Git repository of component B. The chronological order of commits from the master branch [5] is not necessarily the order of actual commits because the history of a Git repository is represented by a graph of commits

[4] A snapshot is the state of the system after a commit.

[5] We adhere to the terminology as used at ASML.

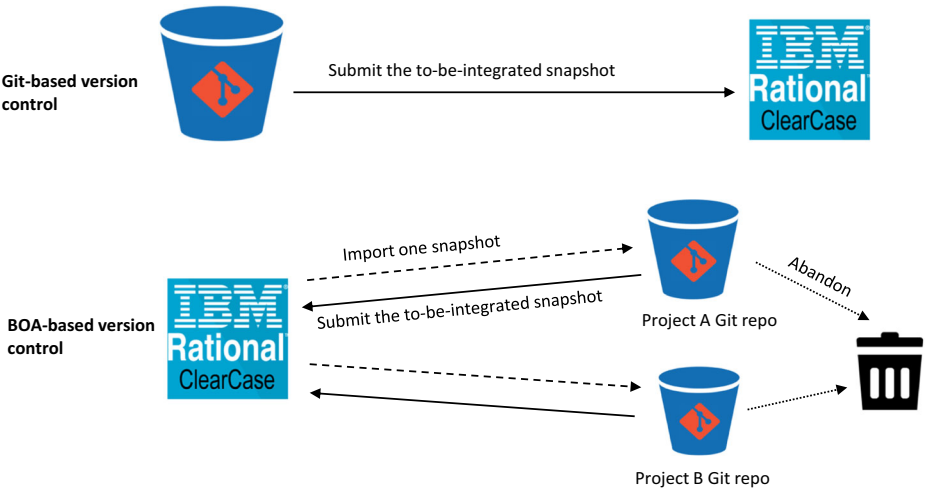


Fig. 8 Git-based and BOA-based ways of working

rather than a linear chain of commits (Bird et al. 2009). However, in this study we limited our scope to the master branch due to the differences between master branch the other branches. First, the master branch versions the models that are ready to be reviewed by other developers or to be submitted to the ClearCase repository while other branches version the development of machine-specific features and different releases, or the fix of certain bugs. According to the developers responsible for the components, these branches can be deleted or merged when a certain development task is finished. Second, the submitted models to the development branches may not be complete or executable (e.g., exhibiting syntactic errors). Third, developers have a different habit of committing to their own development branches (e.g., some developers commit at the end of the working day while some commit when a certain task is finished). These differences require different interpretations for the mined results. As an exploratory study on the evolution of SSSMs, we investigated the master branch, leaving the evolutionary differences present in other branches out of our scope.

We collected the snapshots of the Git repository of component B based on the order they appeared in the master branch. We applied the method discussed in Section 4.3.1 to identify SSSMs. For each snapshot, we measured the number of MSSM-IMs, MSSM-DMs, SSSM-IMs, SSSM-DMs as well as the number of SSSM-IMs that are used for achieving the goals we discussed in Table 4. By analyzing the growth of the number of these models over the years, we aim to understand whether the trends differ between SSSMs and MSSMs, and between different SSSMs used by developers for achieving different goals.

Table 5 Number of model revisions in total and the average number of revisions from Git and ClearCase repositories

Component ID	ClearCase		Git	
	# model revisions	average	#model revisions	average
B	8993	14.2	25181	39.8
C	64	2	246	7.7
D	170	1	544	3.2

To answer **RQ 5.2**, for each model from the ClearCase repository of Component B, we collected all the revisions in chronological order. It should be noted that the history on ClearCase is a subset of the history on Git. That is, some of the commits on Git eventually appear on ClearCase for integration. The developers of component B tag the Git commits that are submitted to ClearCase. Analyzing data from both Git and ClearCase helps us understand how SSSMs evolve during development and integration. Based on the method described in Section 4.3.1, we classified each revision into SSSM or MSSM. To understand whether developers modify transitions of SSSMs, we measured the number of transitions for each model revision. Next, we identified the revisions which are classified differently from their previous revision in the master branch.

Figure 9 illustrates the classification with an artificial example. The model has seven revisions *r1-7* from the master branch. Revision *r1* is an SSSM and the first revision of the model in the repository. The modification by developers results in revision *r2*, which is also an SSSM. Similarly, revisions *r4-5* and *r7* also belong to the same class as their previous revisions. Revisions *r3* and *r6*, however, fall into a different class compare to their previous revisions. We define the *life-cycle* of a model as a series of revisions that introduce the model to the systems, transform the model from an SSSM into an MSSM or transform the model from an MSSM into an SSSM. In the remainder of this paper, we refer to such transformations as SSSM-MSSM-changes. By identifying these revisions, we extracted the *life-cycle* of models in the component. The life-cycle of the example shown in Fig. 9 is SSSM→ MSSM → SSSM.

Next, we categorized SSSM-MSSM-changes following an open-coding process based on the Git commit message associated with the SSSM-MSSSM-changes and the differences between the before-change model revision (i.e., *r2* and *r5* in Fig. 9) and the after-change model revision (i.e., *r3* and *r6* in Fig. 9). This open-coding task is conducted by the first author who has the necessary knowledge of ASD. Since most of SSSM-MSSM-changes were made before 2015, and since then many of these developers who made the changes already work in other development groups or left the company, we found it not feasible to conduct member checking.

5.3 When were SSSMs Introduced? (RQ 5.1)

Figure 10 shows the number of MSSM-IMs, MSSM-DMs, SSSM-IMs and SSSM-DMs present in the Git repository over time. The figure shows an initial surge in 2013 because the first two Git commits are two large squashes of commits from an SVN repository which was used for the initial development of component B and has been removed after importing the latest snapshot into the Git repository.

Overall, the total number of models in this component is growing over the years after the deployment of the component in the machines. As we learned from the developers of component B, component B is the central controller of the machines, coordinating different machine actions. Therefore, the component is likely to be extended or modified when a new

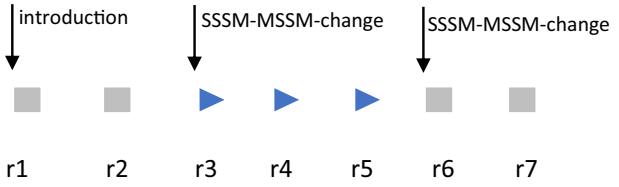


Fig. 9 A set of revisions for a model. Rectangular represents SSSM. Triangle represents MSSM

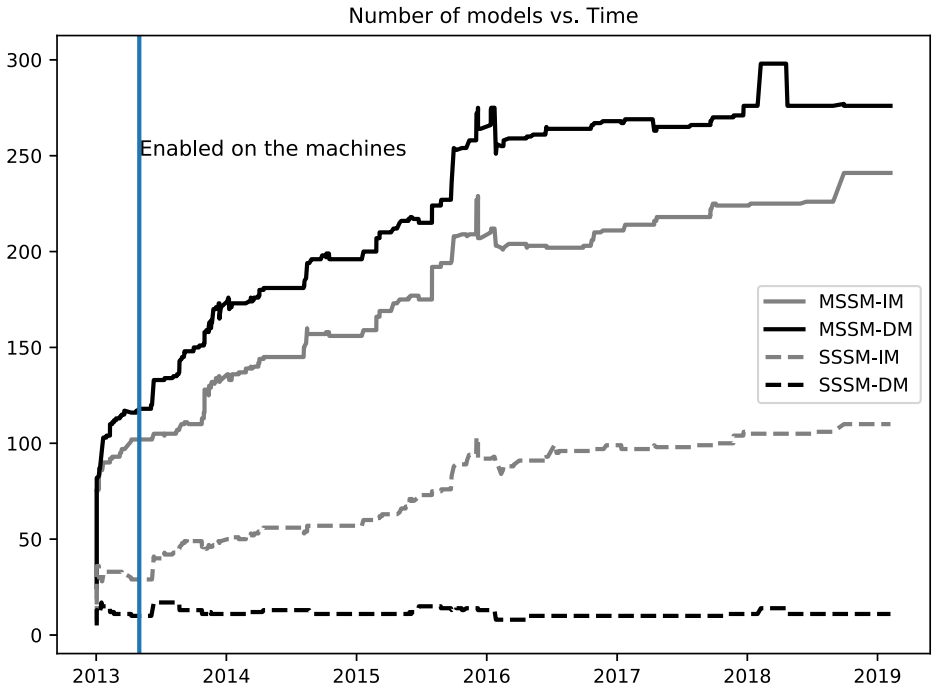


Fig. 10 Growth of the number of models in the Git repository of component B

feature is added to the machines. Developers started using SSSMs before the first deployment of the component and continuously introduced more SSSM-IMs over the years. The growth of all these types of models slowed down noticeably after 2016. This indicates that the component is gradually matured. In contrast, SSSM-DMs were introduced before the first deployment and their usage remains stable throughout the history.

With Fig. 11, we zoom in on the trend for the SSSM-IMs that are used by developers for the core reasons presented in Table 4. After the initial development of the component, eight SSSM-IMs used for easing maintenance and verification were introduced in June 2013, and the number of the SSSM-IMs for this purposes did not grow significantly afterward. A closer look at the commit that contributes to the significant increase in June 2013 reveals that developers introduced the SSSMs when developing a machine-specific feature. These SSSMs abstract machine-specific details away from the client models (see pattern *Feature-Selection* in Fig. 7). Differently, the number of SSSM-IMs that are used to work with the existing code base mainly increased in the period of 2015 and 2017. This implies that the need for interfacing component B with foreign components increases during the period. The number of SSSM-IMs that serves as a workaround solution to tool limitations grew continuously over the years. By further zooming in on the trends for the SSSMs for dealing with different tool limitations as shown in Fig. 12, we found that the demand for SSSMs for different tool limitations varies over time. The implementation of patterns *EventCollector* and *DataEncapsulation* is the main drive behind the growth. The need for the SSSMs from pattern *EventCollector* grew strikingly in 2016 and became relatively stable afterward. By inspecting the related commits, we found that the rapid growth was caused by the implementation of a system design that requires component B to subscribes to a bunch of events, receive the events during runtime, and perform the corresponding actions based on the

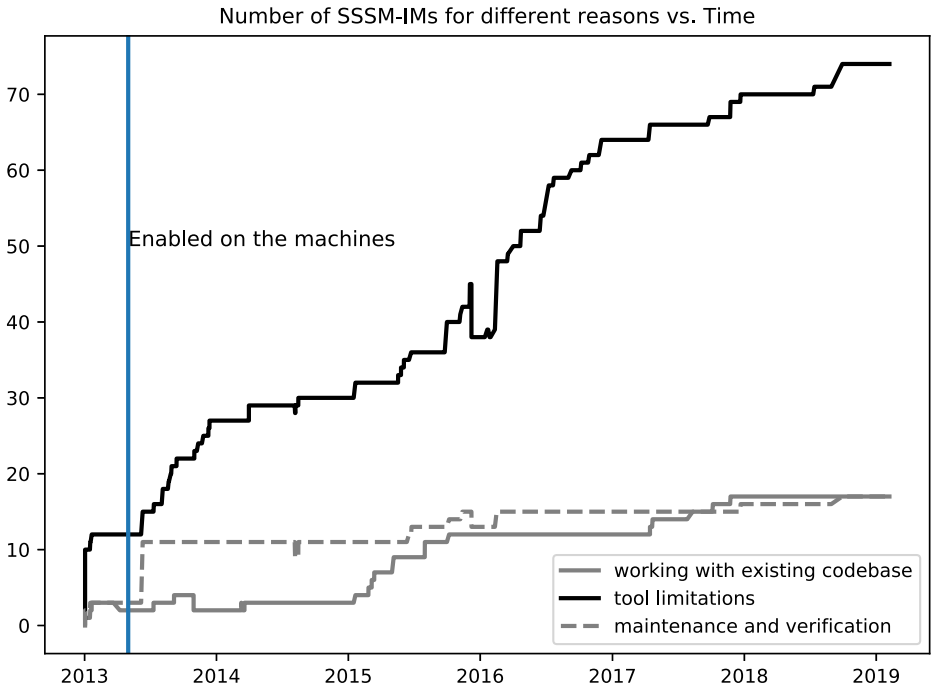


Fig. 11 Growth of the number of SSSM-IMs that are used for different reasons

received events. The introduced SSSMs forward the events to the target parts of component B that are responsible for the corresponding actions (Table 4).

Due to another tool limitation, developers cannot specify data-dependent behavior. The SSSMs in pattern *DataEncapsulation* are used to encapsulate data-dependent behavior implemented in the foreign code (Table 4). The need for data encapsulation with SSSMs appeared from the early phase and continuously grew as developers extend the functionalities of the component. Particularly, it became the main reason for introducing more SSSMs to the component in the recent years.

> **RQ 5.1 summary:** The SSSM-IMs used for hiding machine-specific features were introduced after the initial deployment. The SSSM-IMs used for working with the existing code base were gradually introduced as the new features were developed. The need for SSSM-IMs to deal with tool limitations continuously increases over the years. Particularly, data encapsulation is the main reason why developers introduce additional SSSMs to the component in the recent years.

5.4 How Do Developers Modify SSSMs? (RQ 5.2)

RQ 5.2a: Do SSSMs become MSSMs, and vice versa? Table 6 shows how many IMs and DMs are *always SSSM*, *always MSSM* or *with SSSM-MSSM-changes* from the Git and ClearCase repositories. Note that the total number of models present in the table is 630 rather than 633 that we reported in the previous study (Section 4) because three models were removed from the repositories since our previous data collection activities.

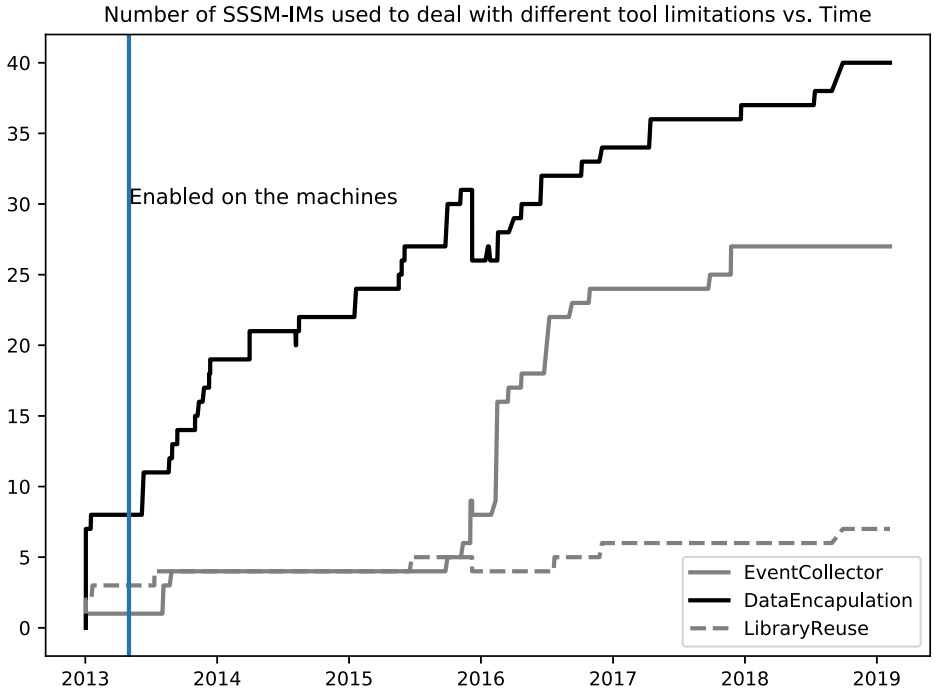


Fig. 12 Growth of the number of SSSM-IM used to deal with different tool limitations

A glance at this table shows that the models from the Git and ClearCase repositories evolve differently. We validate this observation by applying the χ^2 test to the contingency table (Table 6). The null hypothesis is that the evolution of models (i.e., *always SSSM*, *always MSSM* or *with SSSM-MSSM-changes*) is the same regardless the source of models (ClearCase vs. Git). The computed *p-value* is 0.002892 which is smaller than the customary threshold of 0.05. Therefore, we can reject the null hypothesis, concluding that the models from the Git and ClearCase repositories evolve differently. The difference can be attributed to the fact that developers use these two VCSs differently, as we explained in Section 5.1. As can be seen from Table 6, the models from the ClearCase repository are more likely to stay always SSSM or MSSM while the models from the Git repository are more likely to change between SSSM and MSSM. This is explained by the fact that the Git repository stores the work-in-progress revisions, therefore, it is disclosing more modifications.

Moreover, IM and DM have also evolved differently as shown in Tables 7 and 8. To validate this observation, we applied χ^2 test to contingency tables that show how many IMs and DMs are *always SSSM*, *always MSSM* or *with SSSM-MSSM-changes* from the Git and ClearCase repositories (Tables 7 and 8). The null hypothesis of the test is that the life-cycle of models (i.e., *always SSSM*, *always MSSM* and *with SSSM-MSSM-changes*) is independent of the type of models (IM vs. DM). The test result shows that the relation between life-cycle of models and the type of models is significant. The computed *p-values* obtained for the models from both repositories and the adjusted *p-value* with Bonferroni correction are all smaller than 0.00001. Since the adjusted *p-value* is smaller than the customary threshold of 0.05, we can reject the null hypothesis, concluding that IM and DM have evolved differently. Most DMs stay MSSM throughout their history. However, a DM is more likely to be modified with SSSM-MSSM-changes when it is not always an MSSM; In the Git

Table 6 Number of models from the Git and ClearCase repositories that are *always SSSM*, *always MSSM* and *with SSSM-MSSM-changes*

	always SSSM	always MSSM	with SSSM-MSSM-changes	Total
ClearCase	112	506	12	630
Git	108	487	35	630
Total	220	993	47	1260

repository, among 22 DMs that are not always an MSSM, 12 of them (i.e., 54.5%) were modified with SSSM-MSSM-changes. In contrast, among 121 IMs that are not always an MSSM, only 23 of them (i.e., 19%) were modified with SSSM-MSSM-changes.

The common message conveyed by data from both repositories is that for most of the models developers did not make SSSM-MSSM-changes during their maintenance activities; In the ClearCase repository, 506 out of the 630 models (i.e., 80.3%) are MSSM when they were created and have not been changed into SSSMs during their evolution. One hundred twelve models (i.e., 17.7%) were SSSM when they were created and remain to be SSSM throughout their evolution history, leaving 12 models (i.e., 2%) evolving with SSSM-MSSM-changes. Similarly, only 35 models (i.e., 5.6%) from the Git repository have been modified with SSSM-MSSM-changes.

Figure 13 shows the life-cycles of models modified with SSSM-MSSM-changes. The most frequent life-cycle followed by the models is SSSM→ MSSM. Moreover, the models can be switching between SSSM and MSSM multiple times during their evolution. For example, as shown in Fig. 13, there is an IM modified with four SSSM-MSSM-changes. A closer look at the corresponding revisions and commit messages reveals that the last three SSSM-MSSM-changes were made by the same developer within 10 days for redoing a bug fix. In these commits, the developer first reverted the model to the before-fixing revision and then further modified the model for fixing the bug.

In particular, we observed that the SSSM-MSSM-changes for 23 models are not present in the ClearCase repository. For example, in the Git repository, there are two models transformed from MSSM to SSSM and later back to MSSM (i.e., MSSM→ SSSM→ MSSM), which is not shown in the ClearCase repository. This is because consecutive SSSM-MSSM-changes committed to the Git repository might not be visible in the ClearCase repository, as only the to-be-integrated revisions are committed to the ClearCase repository. Figure 14 shows the 35 models that have been modified with SSSM-MSSM-changes from the Git repository. We can observe that SSSM-MSSM-changes are often made one after another within a short period of time; 17 models have been modified with consecutive SSSM-MSSM-changes within one month. For example, model *m2.im* was transformed from an SSSM into an MSSM 19 minutes after its creation. Such quick changes were made before introducing the model to the ClearCase repository. Therefore, the model was an MSSM when it first appeared in the ClearCase repository. Since this model was not modified with

Table 7 Number of IMs and DMs from the Git repository that are *always SSSM*, *always MSSM* and *with SSSM-MSSM-changes*

	always SSSM	always MSSM	with SSSM-MSSM-changes	Total
IM	98	225	23	346
DM	10	262	12	284
Total	108	487	35	630

Table 8 Number of IMs and DMs from the ClearCase repository that are *always SSSM*,*always MSSM* and *with SSSM-MSSM-changes*

	always SSSM	always MSSM	with SSSM-MSSM-changes	Total
IM	102	235	9	346
DM	10	271	3	284
Total	112	506	12	630

SSSM-MSSM-changes after the first integration, it appears to be *always MSSM* in the ClearCase repository. In total, 27 SSSM-MSSM-changes made to 23 models (i.e., m1-23) are only visible in the Git repository, while 20 SSSM-MSSM-changes made to 12 models (i.e., m24-35) are visible in both repositories. The SSSM-MSSM-changes that are only visible in the Git repository reflect the intermediate decisions or corrections that developers made before integrating their changes into the systems.

Additionally, for those models that were modified with SSSM-MSSM-changes, they often start with being an SSSM and later undergo revisions that transformed them into an MSSM. This observation is particularly reflected by the consecutive SSSM-MSSM-changes that developers committed to the Git repository after the creation of SSSMs. As can be seen from Fig. 13, four out of six life-cycles start with SSSM. In total, 31 out of 35 models follow these four life-cycles transforming models from an SSSM into an MSSM, and possibly going back and forth multiple times between SSSM and MSSM throughout their evolution. This observation implies that the behavioral restrictions are not necessarily specified when the model is created. Instead, developers may create an SSSM as the initial implementation and refine the behavior of the model with more states.

RQ 5.2b: Do developers modify transitions of models that stay SSSM throughout their entire history? 112 models from the ClearCase repository and 108 from the Git repository remain to be SSSMs throughout their history. For these models, we observed the stability

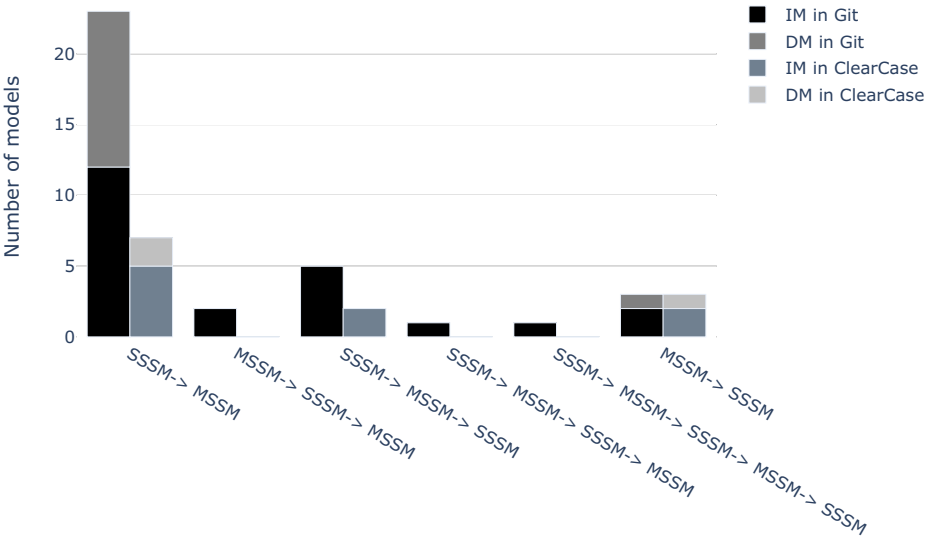


Fig. 13 Life-cycle of SSSMs from the Git and ClearCase repositories

Table 9 Actions that developers take to modify SSSMs

ID	Action	Revision	Comments	#Revisions
1	Condition insertion	SSSM→ MSSM	Add conditions to the execution of events.	4
2	Constraint insertion	SSSM→ MSSM	Add constraints to the execution order of the existing events.	1
3	Event insertion with constraints	SSSM→ MSSM	Add new events with constraints to the execution order of the events.	14
4	Event insertion with conditions	SSSM→ MSSM	Add new events with conditions on the execution of the events.	9
5	Constraint removal	MSSM→ SSSM	Remove the constraints on the execution order of the existing events.	3
6	Condition removal	MSSM→ SSSM	Remove the conditions on the execution of the existing events.	1
7	Event removal	MSSM→ SSSM	As a consequence of removing events, the constraint on the execution order of the events or the conditions of the execution of events is also removed.	9
8	Development continuation	SSSM→ MSSM	Continue the development that was not finished. For example, the models were disconnected to the rest of the models in the component and thus cannot fulfill any roles in the systems.	6

Column #Revisions indicates the number of revisions that are the result of the corresponding action

not only in terms of the number of states, but also the number of transitions. For 74 out of 112 modes obtained from the ClearCase repository and 64 out of 108 models from the Git repository, developers have not changed their number of transitions after creating them.

RQ 5.2c: What modifications are involved when SSSMs are modified to become MSSMs and when MSSMs are modified to become SSSMs? Next, we discuss what actions developers take to modify the models. Table 9 reports the result of our open-coding task. Action *Event insertion with constraint* is the most frequent action developers take, followed by *Event removal* and *Event insertion with conditions*. Figure 14 shows when the actions occur in the evolution of SSSMs.

These actions are based on the concept of conditions and constraints. We first explain the differences between conditions and constraints with an example shown in Fig. 15. Figure 15 (a) shows an SSSM-IM with two call events *initialize* and *stop* and one reply event *ok*. The client model of this SSSM-IM can call *initialize* and *stop* in any order and receive reply *ok*. After adding a condition to the existing events as shown in Fig. 15(b), the model gives reply event *ok* and transits to state *idle* in response to call event *stop* only if it is in state *busy*. Otherwise, the model has no responses, ignoring event *stop*. Similarly, the model does not give responses to event *initialize* when it is in state *busy* as the model has already

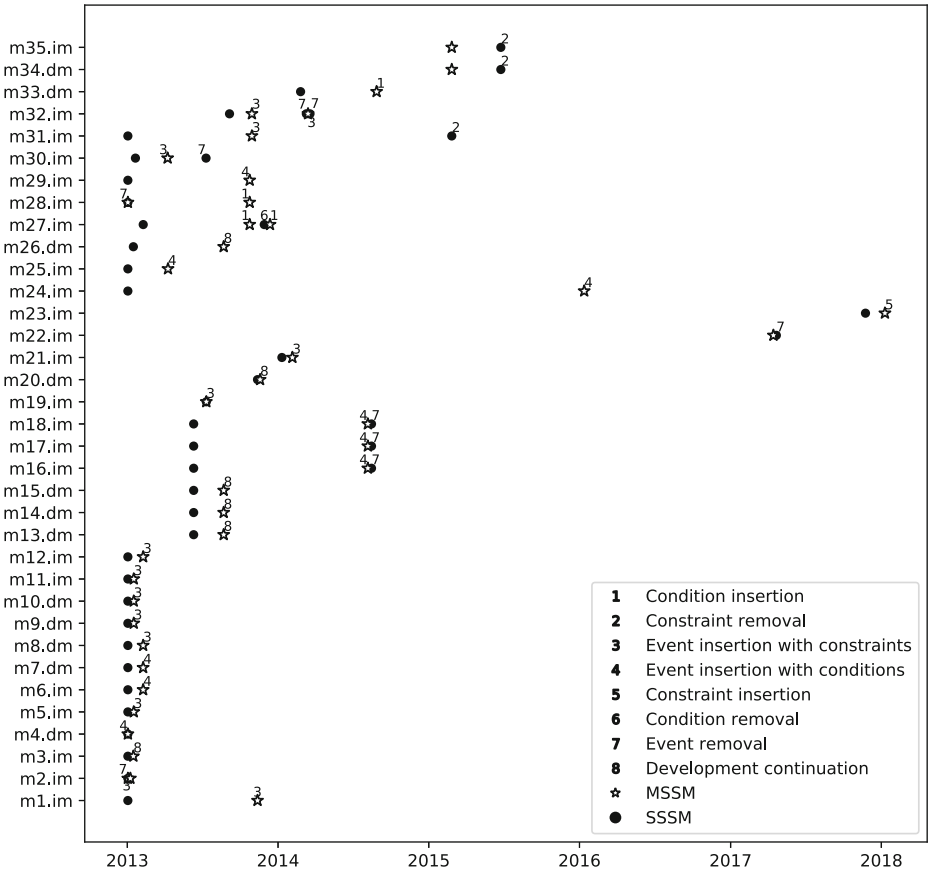


Fig. 14 Evolution of SSSMs present in the Git repository. The numbers indicate the modification actions shown in Table 9 and the shapes indicate the type of models

been initialized. Conditions created by the developers with multiple states allow models to accept all the call events, but give different replies to their client models based on their own state. Adding this condition does not require the client model to change the calling order of events *initialize* and *stop*, thus, no change is propagated from the IM to its upper-layer client models. Differently, adding a constraint to an IM requires the client models to call events in a certain order, which specifies under which circumstances a certain call event can trigger exceptions. In the example shown in Fig. 15(c), the model throws an exception if its client model calls event *initialize* when it is in state *busy*. When the exception behavior is explicitly specified in the model, the verification tool checks if the client model calls events in the expected way. The need for co-change depends on how the client model calls the events; to satisfy the verification tool, the client model needs to be modified if it can trigger the exception under any possible circumstances.

A typical usage of action *Event insertion with constraints* is for implementing the concept of iterator that is available in many programming languages (e.g., Java). Developers intend to implement multiple FIFO (First-in-first-out) lists to store the elements that need to be processed at runtime. The lists are implemented with hand-written code. Initially, the IMs of these lists have only events *append* and *remove* which are called by the client models

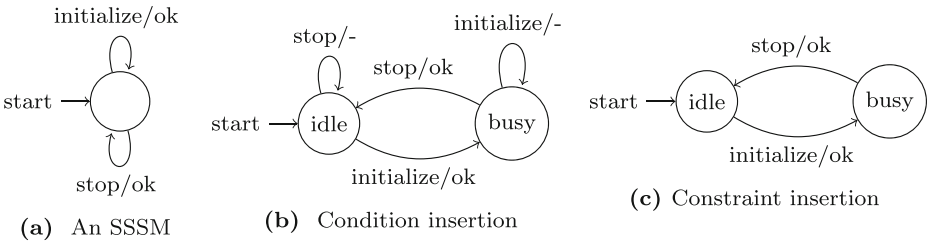


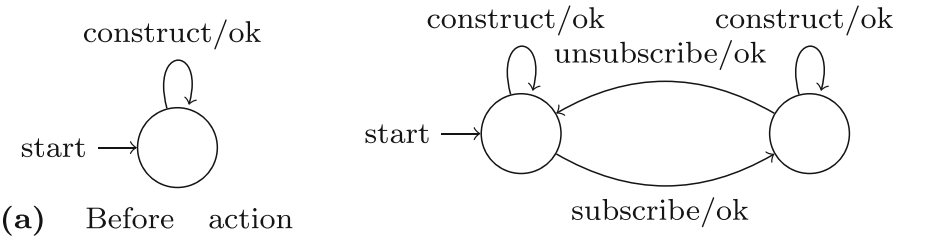
Fig. 15 An example shows the differences between constraint insertion and condition insertion. "*a/b*" indicates that event *b* is sent to the client model when event *a* is called. "-" indicates that no response is being made by the model

to add and remove elements. In the revisions, developers implement the concept of iterator by adding events *iterator* and *next_from_iterator*. The client model can instantiate an iterator by calling event *iterator*, and traverse elements by calling event *next_from_iterator*. Developers then add constraints to the model so that the client model is only allowed to call *next_from_iterator* when event *iterator* is already called (i.e., an iterator is instantiated) and the list is not empty.

We observed an interesting case (i.e., model m23) where the developer takes action *Constraint insertion* to restrict the execution order of the existing events. The before-change revision is an SSSM with commit message:" *...version for first review*" while the after-change revision is an MSSM with commit message:" *...rework after review*", indicating that the action was taken in response to the review feedback. This observation indicates that developers examine whether constraints are needed when reviewing models.

When modifying SSSMs, developers are more likely to add constraints to the execution of newly introduced events. Action *Event insertion with constraint* is taken when developers, add new events whose execution does not depend on the execution of the existing events. Figure 16 shows such an example where events *subscribe* and *unsubscribe* and the constraints on the execution order of these two events are introduced in the revision. The new events and constraints (Fig. 16b) do not impact the execution of the existing event *construct*. That is, event *construct* can still be called in any order regardless of the state of the model. In this case, to satisfy the verification tool, developers only need to ensure that the client model calls the new events *subscribe* and *unsubscribe* in the desired way so that exceptions will not be triggered. Action *Event insertion with constraints* often takes place when developers would like to add a new service which is not coupled with the existing service (i.e., the new service and the existing service can be used by their client models in an independent way). Similarly, action *Event insertion with conditions* is also widely used when a new service is introduced to the models.

When it comes to transforming an MSSM into an SSSM, developers take actions *Constraint removal*, *Condition removal* and *Event Removal*. A typical scenario of performing *Constraint removal* is when developers implement pattern *Model armor* which allows them to remove the constraints from the IMs that interface to the foreign code, and to add models that take the role of *armor* to forward the intended events to upper-layer clients (see Fig. 7). Such modifications on the boundary side of the component will not require the changes on the core parts of the component. The modification shows that pattern *Model armor* was not always implemented from the beginning. Instead, the implementation of the pattern can be a result of refinements.



(a) Before action *Event insertion with constraints*

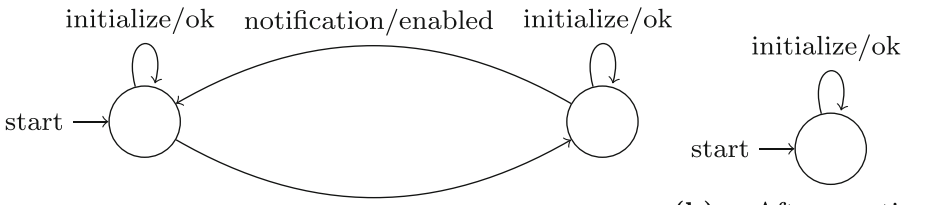
(b) After action *Event insertion with constraints*

Fig. 16 An example of applying action *Event insertion with constraints*

As can be observed, developers often perform *Event Removal* to delete unnecessary events. An interesting example shown in Fig. 17 is a revision for fixing a bug (as indicated in the commit message). Before the action takes place, the MSSM-IM has three input events *initialize*, *enable* and *enabled*. Among them, *initialize* and *enable* are events that can be triggered by its client models. The MSSM-IM sends reply *enabled* to its clients until the occurrence of a notification event from its server. This design subsequently blocks the clients from processing other critical tasks if the notification event does not happen in time. To remove this bug, developers remove events *enable* and *enabled* that block the clients, resulting in an SSSM (as shown in Fig. 17 (b)).

Our result shows that SSSM-MSSM-changes are more likely to be the consequence of adding or removing events rather than the modifications of the execution order of the existing events.

RQ5.2 summary: The main finding is that most SSSMs are stable and have not been modified into MSSMs during their evolution. We observed in total 35 models were modified with the actions that transform them from an SSSM into an MSSM or from an MSSM into an SSSM. The typical modification developers made to these models is adding or removing events rather than the modifications on the execution order of the existing events.



(a) Before action *Event deletion*

(b) After action *Event deletion*

Fig. 17 An example of applying action Event deletion

6 Threats to Validity

As any empirical study, ours is also subject to several threats of validity.

Threats to **construct validity** examine the relation between the theory and observation. Since there is no clear definition of single-state state machines in literature and guidelines, we operationalize the intuitive notion of an SSSM and provide our own definition. To ensure that our definition corresponds to the developers' perception of SSSMs, we explained our definition of SSSMs to the interviewees and made sure that they understood it. While it is possible that some MSSMs can be reduced to SSSMs according to some formal notions of equivalence (e.g., trace equivalence), developers tend not to think about those MSSMs when talking about SSSMs. This is why we exclude this case from consideration and treat MSSMs equivalent to SSSMs as MSSMs.

Threats to **internal validity** concern factors that might have influenced the results. In our interview study, we derive our interview questions and strategy from our quantitative findings, which reduces the risk of asking meaningless questions that potentially bias our interviewees. Moreover, to avoid misinterpretation on developers' ideas, we performed member checks with our interviewees on the categories emerged from the Grounded Theory process. To assure the completeness of the reasons of using SSSMs, we conduct several iterations of interviews till all SSSMs from these 26 components can be explained by the collected reasons. To answer RQ 5.2c, we manually classified the modifications developers made to SSSMs by comparing before-change revisions and after-change revisions. This open-coding process is inevitably interpretative, and hence, subjective. The open-coding was conducted only by the first author due to the required knowledge of the commercial modeling tool. We were not able to conduct member checking with the authors of the revisions because most of changes were made before 2015, and since then many authors have been working in other company units or left the company.

Threats to **external validity** concern the generalizability of our conclusions beyond the studied context. We studied 26 model-based components for the first study (Section 4). Our second study (Section 5) limited to a single component. However, this is the only component that has more than 10 revisions for each model (on average) from this company. Studying the evolution of state-machine-based software is still a challenging subject due to the lack of data. First, the use of MDE with the purpose of verification is still very limited even though the need is already evident, as surveyed by Liebel et al. (2018). Second, since the built-in verification tool formally verifies the correctness of models, the number of revisions developers made to these models might inherently lower than that they made to hand-written code. As shared by the developers that we contacted with, component D has been deployed at the customers' machines, but it does not (yet) evolve much because there is no issue found by the customers so far. Lacking of data can impact the generalizability of the findings. With this preliminary study we intend to increase the understanding of the evolutionary aspects of state-machine-based software with the evidence from industry.

Moreover, we are aware that we limited our study to the components from a single company developed with the same modeling tool. We believe the conclusions and observations derived from this context are complementary to the existing literature which mainly have broad surveys on the challenges of MDE adoption, by providing concrete industrial examples. To increase the generalizability, one of the future directions could be replicating our study in other companies or using the models developed with other tools.

7 Discussion and Implication

As the main contribution, our study identified why developers use SSSM models and how SSSMs evolve in their evolution. Based on our empirical results, we provide implications for developers (Section 7.1), tool builders (Section 7.2) and researchers (Section 7.3). Some of the implications derived from our empirical study are consistent with the findings provided by other survey and interview studies on MDE adoption. Different from these studies that provide a broad insight of MDE adoption, our study aims for more in-depth insights into a certain phenomenon in state machine modeling, by applying mixed methods (i.e., interviews and repository mining) in an industry context. Therefore, we think it is still interesting to confront their conclusions with our findings.

7.1 Implications for Developers

Consider how to integrate models with the existing code base. In our study we found that developers introduce armoring to interface model-based components with code-based components for protecting models from unexpected behavior. In addition, we observed that the usage of SSSMs for interfacing with the existing code base is increasing as more functionalities are implemented. Our observation (in Section 5.3) suggests that practitioners should consider how to integrate models with the existing code base in a scalable way if they would like to use MDE to develop only part of their systems that need to be integrated with hand-written code. Furthermore, practitioners may consider to take the quality (e.g., availability, scalability and maintainability) of the provided integration solutions into account when evaluating candidate modeling tools. This implication concurs with one of challenges that has been reported to hinder MDE adoption in companies (MacDonald et al. 2005; Mohagheghi and Dehlen 2008; Staron 2006; Jolak et al. 2018): using MDE together with the existing code base.

Be aware of the trade-off between domain-specificity and general-purpose programming language constructs. The trade-off between general-purpose modeling languages and domain-specific ones (Van Der Straeten et al. 2008) is a frequently discussed concern about MDE. Domain-specific languages, on the one hand, often offer a higher degree of specialisation for a certain modeling domain or purpose. One the other hand, they might be less flexible and expressive (van Deursen et al. 2000). We observed a large share of SSSM-IMs are used to interface with the hand-written code whose behavior cannot be modelled with ASD because of the tool limitation (Table 4). Particularly, as we observed in our evolution study (Section 5.3), due to the lack of means to specify data-dependent behavior with the tool, the need for encapsulating data-dependent behavior implemented with hand-written code is continually growing over the years, and has become the main reason for using SSSMs in the recent years. Under-specifying the order of events for data manipulation operations require additional review and test efforts. This implies that before adopting a certain modeling language and tool, practitioners need to evaluate the benefit gained from the domain-specificity and the cost caused by the loss of general-purpose language constructs, based on their application domain, while, taking their long-term development and maintenance needs into account. This implication agrees with the suggestion provided by Corcoran (2010) that "one must determine whether a given MDE approach reduces complexity visible to the developer, or whether it simply moves complexity elsewhere in the development process."

Create reusable design using the modeling tool. Apart from developing patterns for interfacing with the existing codebase and dealing with tool limitations, we observed that developers also invest effort in creating patterns that are expected to ease long-term maintenance. They use SSSM-related design patterns to realize such software design principles as low coupling (e.g., *CallMapping*) and separation of concerns (e.g., *FeatureSelection*). Furthermore, future refactoring is facilitated with SSSMs implementing the idea of "packaging up sub-steps". We observed that these patterns were introduced in the early phase of the maintenance of component B and widely reused in other components. Our observation implies that practitioners can consider to build up reusable design patterns when using a certain modeling tool, to ease their development in future projects developed with the same tool. This implication is inline with earlier findings on MDE adoption (Hutchinson et al. 2014) and software engineering practice in general (Ampatzoglou et al. 2011).

Balance modeling trade-off between the ease of modeling activity and the verification adequacy. As discussed by Chaudron et al. (2012), developers who work with traditional UML modeling, i.e., use models merely for analysis, understanding and communication, have to make a trade-off between effort in modeling and the risk of problems caused by imperfections (e.g., incompleteness, redundancy and inconsistencies) in downstream development. For instance, when a model serves as a blueprint of the protocol between two components, the under-specified parts in the model might be implemented inconsistently due to different interpretations by different developers, later incurring repair costs. However, investing a lot of effort in continuously refining such blueprints is not always possible (Lange et al. 2006a). Our results imply a similar trade-off that developers need to make in the context of using models for verification. Under-specifying the behavior of models might hide defects from the verification tools. However, spending too much effort in creating a more precise model with a restricted order of events slows down development process. Moreover, developers might need to spend more effort in performing changes on such models because passing verification becomes non-trivial. Our study on the evolution of SSSMs shows that developers are more likely to change an SSSM into an MSSM than the opposite (Section 5.4). Sometimes, developers consecutively make multiple SSSM-MSSM-changes within a short period of time, transforming the models back and forth between SSSM and MSSM. These work-in-progress changes often are not eventually integrated into the systems, implying that a series of refinements have to take place before integration.

7.2 Implications for Tool Builders

Help developers with integration. Our work calls for improving the support of integration of models and code-based components. The need to integrate models with the existing code base (Liebel et al. 2014; Hutchinson et al. 2011; Whittle et al. 2013b) and to integrate models from different domains (Tolvanen and Kelly 2010; Torres et al. 2019) has been often mentioned. However, not many studies propose how this integration can be facilitated by improving modeling tools. To provide suggestions to MDE tool builders about integration, Greifenberg et al. survey eight design patterns proposed for integrating generated and hand-written object-oriented code (Greifenberg et al. 2015). One of the discussed design patterns is the GoF design pattern *Delegation* (Gamma et al. 1993) which allows generated code (delegator) to invoke methods of the hand-written code (delegate) declared in an explicit interface (delegate interface). The *ModelArmor* design pattern we identified (Fig. 7) implements a similar idea; DM *Armor* takes the role of delegator invoking methods of code-based components specified in IM *IForeign*. However, as opposed to *Delegation*,

ModelArmor takes into account the different properties of models and code (i.e., verified behavior vs. non-verified and unpredictable behavior), ensuring that models are protected from the unexpected behavior of the code. Our work implies that while selecting design patterns for integration, tool builders should consider different properties of generated and handwritten code. Furthermore, tool builders can (partially) automate the implementation of the integration patterns, reducing the manual development effort.

Facilitate library reuse. Apart from interfacing with existing code-based components, we have observed that developers have to use code to implement what cannot be expressed by models (Section 4.4.4). For example. due to the lack of reusable common libraries, developers implement in code the behavior that requires such libraries. To address this challenge the tool builders can work on two directions. First, one can consider enriching common functionalities often used in different applications with built-in models to reduce the needs of interfacing with libraries provided by general-purpose programming languages. Second, given rich reusable libraries in general-purpose programming languages, tools should provide a way to easily reuse these libraries, similar to the wrapping mechanism that allows, e.g., Python programs to communicate with C/C++ (Beazley 1996).

Meet wider specification and verification needs. We have observed that developers attempt to implement global constants with SSSMs (Section 4.4.4). This practice indicates the need to support concepts shared by multiple models. However, implementing such concepts is hindered by a well-known verification challenge: state explosion problem (Clarke et al. 2001; Baldoni et al. 2018). Such modeling tools as Uppaal (Behrmann et al. 2006) support the use of global variables (e.g., bounded integers and arrays) that can influence the control flow in the models. However, such tools have larger risk of facing state explosion when dealing with real-life applications (Doornbos et al. 2012). This implies that a trade-off between supporting global variables and the risk of state explosion has to be resolved by tool designers. A possible resolution could be adopting hybrid solutions (Doornbos et al. 2012; Xing et al. 2010) that translate models from one tool to another, to meet wider verification needs.

7.3 Implications for Researchers

As befitting an exploratory case study (Runeson and Höst 2009), we propose hypotheses about the use of SSSMs in modeling practice. These hypotheses should be verified in a follow-up study.

H1: *The design patterns in Section 4.4.2 help developers to achieve the corresponding goals*. We have seen that SSSMs are extensively used for various reasons and goals.

The studies on the effectiveness of GOF design patterns in OOP languages (Gamma et al. 1993) have shown that design patterns do not always achieve the claimed advantages (Ampatzoglou et al. 2015; Zhang and Budgen 2011). Moreover, passing verification easily with SSSMs might be a potential risk. This suggests a need to investigate effectiveness of these SSSM-related design patterns in order to confidently apply them.

H2.1: *SSSMs shorten the development time and ease modification tasks of their client models, compared to MSSMs.* **H2.2:** *The models that use or implement SSSM-IMs have more post-release defects compared to the models that work with MSSM-IMs.* These two hypotheses are derived from our interviewees' perception (**RQ4**, Section 4.4.7). It is, however,

unknown how SSSMs actually impact development, maintenance and verification activities. Investigating the impacts of SSSMs, the type of model that minimizes modeling effort, is a starting point toward better understanding of a trade-off between the effort spent on designing a model that maximizes the advantage of verification and the extra cost caused by downstream problems due to inadequate verification. We expect that the investigation of this trade-off can broaden the ongoing discussion of modeling trade-offs that is currently focusing on UML modeling (Chaudron et al. 2012; Raghuraman et al. 2019b).

H3: *Most models either remain SSSMs or MSSMs and are not modified with SSSM-MSSM-changes.* The validation of this hypothesis may provide suggestions for tool builders. If, both hypotheses **H2.2** and **H3** hold, then it may indicate that there is a need to detect the SSSMs that might be associated with post-release defects during commit activities to avoid problems.

H4: *Most of the SSSM-MSSM-changes are related to the introduction or removal of events rather than to the modification on the execution order of the existing events.* We observed the tendency in our study on a single component from a single company. It therefore requires empirical validation. In particular, validating **H4** can help us understand what SSSM-MSSM-changes are more likely to occurs, and further investigate how SSSM-MSSM-changes to a model impact other models that depend on it and whether any tools are required to support the evolution. Many studies have investigated API breaking changes (Brito et al. 2018; Mostafa et al. 2017) in the context of traditional coding, proposing suggestions and tools for library and client developers. In MDE, breaking changes also deserve attention. Adding events or adding conditions to the existing events are non-breaking changes as they do not force client models to change. However, other modifications such as removing events or adding constraints to the existing events are breaking changes that require changes on client models. A further investigation is required to understand how likely developers introduce breaking/non-breaking changes when modifying SSSMs. Moreover, a futher exploration on what kind of modifications occurs more often than others can help tool builders prioritize and facilitate certain actions (e.g., addition and removal of events) when designing a user interface.

Beyond the specific hypotheses, we suggest researchers to further study the evolution of models. We observed that SSSMs are a minority and most of them are SSSM since their introduction to the systems. Particularly, SSSMs are more likely to become MSSM models than the other way around. The predominance of evolution from SSSMs to MSSMs can be seen as an example of increasing complexity of a system. This implies possible applicability of Lehman's laws of software evolution to models operating in a hybrid model/code context, and suggesting further research into this topic. By comparing Git history (work-in-progress revisions) and ClearCase history (integration revisions), we observed (in Section 5.4) that multiple SSSM-MSSM-changes often occur consecutively within a short period of time before the final revisions are available in the integration repository (ClearCase). Based on the commit messages, it can be inferred that some of them were made in response to review feedback or request of redoing a bug fix. However, it remains unclear why the previous revision was unsatisfying, due to the lack of the explanation from the authors of the commits. The observation also implies that the changes of models might be driven by peer discussion in the review process, suggesting future research on the role and practice of peer review in model evolution. In addition, given the caused permissive verification is perceived as a risk by our interviewees, we suggest proposing possible alternatives to SSSM-IMs by investigating the order in which events are *actually* being called during system operation. One

can consider analysing the execution traces of the generated code with pattern mining techniques widely studied in the field of model learning (Yang et al. 2019; Wieman et al. 2017; Aslam et al. 2018), specification mining (Lemieux et al. 2015; Lo et al. 2011) and process mining (van der Aalst 2011; van der Werf et al. 2009; Gupta et al. 2018).

8 Related Work

8.1 MDE Adoption and Practice

Our study is closely related to a series of empirical studies on MDE adoption and practice. Mohagheghi and Dehlen (2008) identified the need for more empirical evidence on MDE subjects by reviewing 25 papers. Twenty-one of these papers were experience reports from single projects, while four report comparative studies. The review study attempted to identify the benefits and limitations of MDE. As a result, the study found that the improvement of software quality, productivity gains and losses are not well-reported in these papers, making it hard to generalize the results. Therefore, the authors call for more empirical evidence on MDE subjects to help researchers understand MDE adoption, practice, and experience. Since then, many empirical MDE studies have been conducted to understand how MDE is being adopted and applied in practice (Hutchinson et al. 2011; Whittle et al. 2013a; Hutchinson et al. 2014; Whittle et al. 2013b; Farias et al. 2013; Pourali and Atlee 2018; Chaudron et al. 2012; Liebel et al. 2014; Mohagheghi et al. 2013). These papers explored different dimensions of MDE adoption and practice, using mostly interviews and surveys.

Liebel et al. (2014) and Liebel et al. (2018) conducted a survey with 113 MDE practitioners to assess the current state of practice and the challenges in the development of embedded systems. The study found embedded software engineers use MDE mainly for simulation, code generation and documentation. The overall benefits gained from MDE outweigh the negative effects of MDE. The challenges perceived by engineers mainly lie in the sufficiency and interoperability of tools.

To understand the impact of tools on MDE adoption, Whittle et al. (2013b) conducted 20 interviews with MDE practitioners, resulting in a taxonomy of tool-related considerations. In addition, the study also reveals that MDE tools, in many cases, add complexity to the development, although it was expected to help developers deal with complexity of systems. One of the problems that contributes to the insufficiency of tools is a lack of consideration for how developers actually work and think. To resolve this problem, there is a need to study how developers model systems and what challenges they face.

Several studies investigated challenges developers face in modeling (Pourali and Atlee 2018; Chaudron et al. 2012). Pourali and Atlee (2018) identified the gap between users' expectation on UML modeling tools and their actual experience. The study evaluates eight modeling tools by recruiting 18 students who are experienced with UML modeling to conduct four modeling tasks. The study found that the students mainly have difficulties in fixing inconsistencies which are most in need of consideration from tool builders. The inconsistencies and other forms of imperfection (e.g., redundancy and incompleteness) might cause downstream problems, as discussed by Chaudron et al. (2012) based on a series of surveys and interviews, raising a question of how much modeling is good enough in the context of using UML as communication vehicle and implementation blueprint. Our study further reveals that this question remains when extending the use of models to verification.

Furthermore, several studies went beyond the technical aspects of MDE adoption and practice, exploring the organizational, managerial and social factors that lead to successful

adoption of MDE (Hutchinson et al. 2014; Hutchinson et al. 2011; Whittle et al. 2013a). Based on a series of survey and semi-structured interviews with MDE practitioners from industry, the authors conclude that an iterative and progressive approach, organizational commitment, and motivated users are required to successfully adopt MDE in industry.

Similar to these studies on MDE adoption and practice, we aimed for obtaining empirical evidence to help researchers and tool builders better understand how developers use MDE in practice. Specifically, we enriched the existing knowledge of MDE practice through the lens of why developers use SSSMs that is not recommended by a widespread modeling guideline, and how developers use SSSMs.

8.2 Guideline Adherence

Our study is inspired by the literature on how and why software developers (do not) follow programming and modeling guidelines or best practices.

A large body of literature has investigated the occurrence of violations to the common wisdom in traditional coding practice. These study observed a phenomenon that the violations often occur when the code is first introduced to the system. Tufano et al. (2015) studied when and why code smells are introduced by mining software repositories. The result shows that most of the time code smells are introduced in the development phase rather than in the evolution phase that common wisdom expects, which implies that potential poor design can be detected by performing quality checks during commit activities to avoid worse problems in future. Similarly, a study on Eclipse interface usage by Eclipse third-party plug-ins found that a significant portion of Eclipse third-party plug-ins uses "bad" interfaces and the bad usage was not removed from the systems (Businge et al. 2015). This phenomenon is further discovered by the study on how code readability changes during software evolution (Piantadosi et al. 2020). The result shows that unreadable code is a minority and most of the unreadable pieces are unreadable since their creation. Following the same strategy, our study investigated the reasons behind violations of a widespread modeling recommendation — not to use SSSMs, and the evolution of these SSSMs. We observed the same phenomenon that the violations occur when the models are created and only a small share of models are changed between SSSM and MSSM.

The studies on guideline adherence have also been conducted to understand UML modeling practice. Lange and Chaudron (2004) formulated a collection of rules to asses the completeness of UML models, and further explored to what extent developers violate these rules in practice. The result shows a large amount of rule violations, suggesting that the incompleteness of models should be addressed. Lange et al. (2006b) further conducted a controlled experiment to explore the effect of modeling conventions on defect density and modeling effort. The results show that the defect density in UML models is reduced when using modeling conventions, although the improvement is not statistically significant. Different from these studies, our study explored the *reasons* behind the violations in state-machines modeling practice.

8.3 Evolution in MDE

Our evolution study on SSSMs is related to the studies of evolution in MDE.

Mens et al. (2005) proposed a framework to support the evolution of UML models. The framework includes a classification of model inconsistencies and the formalism of description logic that can be used to formulate logic rules detecting model inconsistencies. In MDE practice, not only models evolve, but their meta-models in which the models are expressed

also evolve (Mengerink et al. 2018; Mens et al. 2007; Favre 2005). A bunch of studies has investigated the evolution of meta-models (Mengerink et al. 2018; Gruschko et al. 2007; Etzlstorfer et al. 2017; Sprinkle et al. 2009). Mengerink et al. (2018) empirically studied how domain-specific languages (DSL) evolve by mining an industrial repository. The study distinguishes between syntactic changes and semantic changes, and found that most of DSL evolution is redefinition of its semantics. An interesting extension of our study could be investigating the syntactic and semantic changes of state-machine models.

Co-evolution between different model artifacts is one of the challenges in model evolution. The approaches have been proposed to facilitate the co-evolution between meta-models and conforming models (Jongeling et al. 2020; Mengerink et al. 2016; Hebig et al. 2016a). Moreover, the recent work from Khelladi et al. (2020b) and Khelladi et al. (2020a) proposed an approach to support the co-evolution of code and metamodels, i.e., when changing mata-models, the co-evolution propagates the metamodel changes to the code that depends on the metamodel. Our study observed that many SSSMs are used for interfacing with the existing code. It remains an interesting study to explore the co-evolution between the SSSMs on boundaries of the model world and the hand-written code that interface with these SSSMs.

8.4 Model Repository Mining

Our study is also related to the studies that mine model repositories. Pattern and clone detection is one of the goals to mine model repositories (Babur 2018; La Rosa et al. 2015; Stephan and Rapos 2019; Stephan and Cordy 2015). Similar to our work, Stephan and Cordy (2015) mine model repositories to detect patterns. The study predefined a set of patterns using models and identified the models that are similar with the patterns within a given threshold. Differently, our exploratory study identifies the patterns by mining a type of model that is not recommended by modeling guidelines and discussing the mined results with developers. As one of the main findings, we discovered several design patterns as shown in Fig. 7. Our study can further be extended with the pattern mining approach to detect instances of discovered patterns in the entire model base.

Some studies mined MDSE repositories to investigate the quality of hand-written code and generated code from models. He et al. (2016) mined 16 MDE projects and concluded that the generated code from models present more code smells than what developers usually produce in their hand-written code. By mining MDSE repositories and non-MDSE repositories, Rahad et al. (2021) further identified that hand-written code fragments from MDSE repositories suffer more from technical debt and code smells, compares to hand-written code in non-MDSE repositories. These two studies pointed out that the traditional coding guidelines are violated by code generators and developers in MDSE practice. Our study empirically shows that developers violate a widespread modeling guideline in order to integrate models with the existing code base. These studies imply that the adoption of MDSE may introduce violations to the coding and modeling guidelines that are considered to be common wisdom in software engineering practice. To improve the MDSE practice, guidelines and tools, the results of these studies call for more empirical studies to discover the workarounds and compromises that developers made when adopting MDSE.

Several studies have been conducted to mine UML models. Robles et al. (2017) and Hebig et al. (2016b) contributed datasets with UML diagrams mined from GitHub. The datasets enable several mining studies to advance the understanding and techniques in UML modeling. Osman et al. (2018) developed the techniques to automatically classify UML models into hand-made diagrams as part of the forward-looking development process and the diagrams reverse engineered from the source code. Raghuraman et al. (2019a) mined

software repositories and identified that the projects with UML models present in the repositories are less prone to defects compared to projects without UML models present in the repositories. This finding confirms the intuition that the use of UML models can improve the quality of software.

9 Conclusion

With the aim of understanding why developers violate a widespread modeling guideline, we conducted an exploratory study to understand under which circumstances developers use SSSMs in their practice. Our exploratory study consists of two complementary studies. We first investigated the prevalence and role of SSSMs in the domain of embedded systems, as well as the reasons why developers use them and their perceived advantages and disadvantages. We employed the sequential explanatory strategy, including repository mining and interview, to study 1500 state machines from 26 components at ASML, a leading company in manufacturing lithography machines from the semiconductor industry. Then, we investigated the evolutionary aspects of the SSSMs, exploring when SSSMs are introduced to the systems and how developers modify them by mining the largest state-machine-based component from the company.

We observed that 25 out of 26 components contain SSSMs. The SSSMs make up 25.3% of the model base. Our interviews suggest that SSSMs are used to interface with the existing code, to deal with tool limitations, to facilitate maintenance and to ease verification. Our study on the evolutionary aspects of SSSMs reveals that the need for SSSMs to deal with tool limitations grew continuously over the years. Moreover, we observed the majority of the SSSMs are stable and have not been changed during their evolution. The most frequent modifications developers made to SSSMs is inserting events with constraints and conditions on the execution of the events.

Based on our results, we provide implications to modeling tool builders and developers. Furthermore, we formulate four hypotheses about the effectiveness of SSSMs, the impacts of SSSMs on development, maintenance and verification as well as the evolution of SSSMs.

References

Ambler SW (2005) The elements of UML™2.0 style. Cambridge University Press, Cambridge

Ampatzoglou A, Kritikos A, Kakarontzas G, Stamelos I (2011) An empirical investigation on the reusability of design patterns and software packages. J Syst Softw 84(12):2265–2283

Ampatzoglou A, Chatzigeorgiou A, Charalampidou S, Avgeriou P (2015) The effect of gof design patterns on stability: a case study. IEEE Trans Softw Eng 41(8):781–802

Antoniol G, Fiutem R, Cristoforetti L (1998) Design pattern recovery in object-oriented software. In: Proceedings. 6th international workshop on program comprehension. IWPC'98 (Cat. No. 98TB100242). IEEE, pp 153–160

Aslam K, Luo Y, Schiffelers RRH, van den Brand MG (2018) Interface protocol inference to aid understanding legacy software components. In: MODELS workshops, pp 6–11
Babur Ö (2018) Clone detection for ecore metamodels using n-grams. In: MODELSWARD, pp 411–419
Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I (2018) A survey of symbolic execution techniques. ACM Comput Surv (CSUR) 51(3):50
Beazley DM (1996) Swig: An easy to use tool for integrating scripting languages with c and c++. In: Tcl/Tk workshop. pp 43
Behrmann G, David A, Larsen KG, Håkansson J, Pettersson P, Yi W, Hendriks M (2006) Uppaal 4.0
Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: 2009 6th IEEE International working conference on mining software repositories, IEEE, pp 1–10
Brito A, Xavier L, Hora A, Valente MT (2018) Apidiff: Detecting Api breaking changes. In: 2018 IEEE 25Th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 507–511
Buchbinder E (2011) Beyond checking: Experiences of the validation interview. Qual Soc Work 10(1):106–122
Businge J, Serebrenik A, van den Brand MG (2013) Analyzing the eclipse API usage: Putting the developer in the loop. In: 17th European conference on software maintenance and reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013. IEEE Computer Society, pp 37–46
Businge J, Serebrenik A, Van Den Brand MG (2015) Eclipse api usage: the good and the bad. Softw Qual J 23(1):107–141
Capilla R, Bosch J, Trinidad P, Ruiz-Cortés A, Hinchey M (2014) An overview of dynamic software product line architectures and techniques: Observations from research and industry. J Syst Softw 91:3–23
Chaudron MRV, Heijstek W, Nugroho A (2012) How effective is uml modeling? Softw Syst Model 11(4):571–580
Chen YL, Lafortune S (1995) Modular supervisory control with priorities for discrete event systems. In: 34th Conference on decision and control. IEEE, pp 409–415
Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2001) Progress on the state explosion problem in model checking. In: Informatics. Springer, pp 176–194
Corcoran D (2010) The good, the bad and the ugly: experiences with model driven development in large scale projects at ericsson. In: European conference on modelling foundations and applications. Springer, pp 2–2
Dennis A, Wixom BH, Tegarden D (2009) Systems analysis and design UML Version 2.0. Wiley, Hoboken
de San Pedro J, Cortadella J (2016) Mining structured petri nets for the visualization of process behavior. In: Ossowski S (ed) Proceedings of the 31st annual ACM symposium on applied computing, Pisa, Italy, April 4-8, 2016. ACM, pp 839–846
Dong J, Lad DS, Zhao Y (2007) Dp-miner: Design pattern discovery using matrix. In: 14th Annual IEEE international conference and workshops on the engineering of computer-based systems (ECBS'07), IEEE, pp 371–380
Doornbos R, Hooman J, van Vlimmeren B (2012) Complementary verification of embedded software using asd and uppaal. In: 2012 International conference on innovations in information technology (IIT). IEEE, pp 60–65
Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering. Springer, pp 285–311
Etzlstorfer J, Kapsammer E, Schwinger W (2017) On the evolution of modeling ecosystems: An evaluation of co-evolution approaches. In: MODELSWARD. pp 90–99
Farias K, Garcia A, Whittle J, Lucena C (2013) Analyzing the effort of composing design models of large-scale software in industrial case studies. In: International conference on model driven engineering languages and systems. Springer, pp 639–655
Favre JM (2005) Languages evolve too! changing the software time scale. In: Eighth international workshop on principles of software evolution (IWPSE'05), IEEE, pp 33–42
Fdr homepage (2014) http://www.fsel.com
Fisher RA (1922) On the interpretation of χ 2 from contingency tables, and the calculation of p. J R Stat Soc 85(1):87–94
Gamma E, Helm R, Johnson R, Vlissides J (1993) Design patterns: Abstraction and reuse of object-oriented design. In: European conference on object-oriented programming. Springer, pp 406–431
Garcia J, Ivkovic I, Medvidovic N (2013) A comparative analysis of software architecture recovery techniques. In: Proceedings of the 28th IEEE/ACM international conference on automated software engineering. IEEE Press, pp 486–496
Greifenberg T, Hölldobler K, Kolassa C, Look M, Nazari PMS, Müller K, Perez AN, Plotnikov D, Reiss D, Roth A et al (2015) Integration of handwritten and generated object-oriented code. In: International conference on model-driven engineering and software development. Springer, pp 112–132

Gruschko B, Kolovos D, Paige R (2007) Towards synchronizing models with evolving metamodels. In: Proceedings of the international workshop on model-driven software evolution, Amsterdam, The Netherlands, p 3

Gupta M, Mandal A, Dasgupta G, Serebrenik A (2018) Runtime monitoring in continuous deployment by differencing execution behavior model. In: Pahl C, Vukovic M, Yin J, Yu Q (eds) Service-oriented computing - 16th international conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings, Lecture Notes in Computer Science, vol 11236. Springer, pp 812–827. https://doi.org/10.1007/978-3-030-03596-9_58

He X, Avgeriou P, Liang P, Li Z (2016) Technical debt in mde: a case study on gmf/emf-based projects. In: Proceedings of the ACM/IEEE 19th international conference on model driven engineering languages and systems. pp 162–172

Hebig R, Khelladi DE, Bendraou R (2016a) Approaches to co-evolution of metamodels and models: a survey. IEEE Trans Softw Eng 43(5):396–414

Hebig R, Quang TH, Chaudron MR, Robles G, Fernandez MA (2016b) The quest for open source projects that use uml: mining github. In: Proceedings of the ACM/IEEE 19th international conference on model driven engineering languages and systems. pp 173–183

Hutchinson J, Whittle J, Rouncefield M, Kristoffersen S (2011) Empirical assessment of mde in industry. In: Proceedings of the 33rd international conference on software engineering. ACM, pp 471–480

Hutchinson J, Whittle J, Rouncefield M (2014) Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure. Sci Comput Program 89:144–161

Jolak R, Ho-Quang T, Chaudron MR, Schiffelers RRH (2018) Model-based software engineering: A multiple-case study on challenges and development efforts. In: Proceedings of the 21th ACM/IEEE international conference on model driven engineering languages and systems. pp 213–223

Jongeling R, Cicchetti A, Ciccozzi F, Carlson J (2020) Co-evolution of simulink models in a model-based product line. In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems. pp 263–273

Khelladi DE, Combemale B, Acher M, Barais O (2020a) On the power of abstraction: a model-driven co-evolution approach of software code. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering: new ideas and emerging results. pp 85–88

Khelladi DE, Combemale B, Acher M, Barais O, Jézéquel JM (2020b) Co-evolving code with evolving metamodels. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1496–1508

Kim YG, Hong HS, Bae DH, Cha SD (1999) Test cases generation from uml state diagrams. IEE Proc-Softw 146(4):187–192

Kronlid F (2006) Turn taking for artificial conversational agents. In: Klusch M, Rovatsos M, Payne TR (eds) Cooperative information agents x, 10th international workshop, lecture notes in computer science, vol 4149. Springer, pp 81–95

Kuhn A, Ducasse S, Gírba T (2007) Semantic clustering: Identifying topics in source code. Inf Softw Technol 49(3):230–243

La Rosa M, Dumas M, Ekanayake CC, García-Bañuelos L, Recker J, ter Hofstede AH (2015) Detecting approximate clones in business process model repositories. Inf Syst 49:102–125

Lange CF, Chaudron MR (2004) An empirical assessment of completeness in uml designs. In: Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE '04), IET. pp 111–121

Lange CF, Chaudron MR, Muskens J (2006a) In practice: Uml software architecture and design description. IEEE Softw 23(2):40–46

Lange CF, DuBois B, Chaudron MR, Demeyer S (2006b) An experimental investigation of uml modeling conventions. In: International conference on model driven engineering languages and systems. Springer, pp 27–41

Lehman MM (1979) On understanding laws, evolution, and conservation in the large-program life cycle, vol 1, pp 213–221. https://doi.org/10.1016/0164-1212(79)90022-0, https://www.sciencedirect.com/science/article/pii/0164121279900220

Lemieux C, Park D, Beschastnikh I (2015) General ltl specification mining, IEEE

Liebel G, Marko N, Tichy M, Leitner A, Hansson J (2014) Assessing the state-of-practice of model-based engineering in the embedded systems domain. In: International conference on model driven engineering languages and systems, Springer. pp 166–182

Liebel G, Marko N, Tichy M, Leitner A, Hansson J (2018) Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. Softw Syst Model 17(1):91–113

Lo D, Khoo SC, Han J, Liu C (2011) Mining software specifications: methodologies and applications. CRC Press, Boca Raton

MacDonald A, Russell D, Atchison B (2005) Model-driven development within a legacy system: an industry experience report. In: Australian software engineering conference. IEEE, pp 14–22
Mengerink J, Schiffelers RR, Serebrenik A, van den Brand M (2016) Dsl/model co-evolution in industrial emf-based mdse ecosystems. In: ME@ MODELS. pp 2–7
Mengerink JG, van der Sanden B, Cappers BC, Serebrenik A, Schiffelers RR, van den Brand MG (2018) Exploring dsl evolutionary patterns in practice. In: Proceedings of the 6th international conference on model-driven engineering and software development, SCITEPRESS-Science and Technology Publications, Lda, pp 446–453
Mengerink JGM, Serebrenik A, Schiffelers RRH, van denBrandMGJ (2017) Automated analyses of model-driven artifacts: obtaining insights into industrial application of mde. In: Proceedings of the 27th international workshop on software measurement and 12th international conference on software process and product measurement. ACM, pp 116–121
Mens T, Van DerStraetenR, Simmonds J (2005) A framework for managing consistency of evolving uml models. In: Software evolution with UML and XML, IGI Global. pp 1–30
Mens T, Blanc X, Mens K (2007) Model-driven software evolution: an alternative research agenda. In: The 6th BElgian-NEtherlands software eVOLution workshop (BENEVOL 2007), pp 1–7
Mohagheghi P, Dehlen V (2008) Where is the proof?-a review of experiences from applying mde in industry. In: European conference on model driven architecture-foundations and applications. Springer, pp 432–443
Mohagheghi P, Gilani W, Stefanescu A, Fernandez MA (2013) An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empir Softw Eng 18(1):89–116
Mostafa S, Rodriguez R, Wang X (2017) Experience paper: a study on behavioral backward incompatibilities of java software libraries. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis. pp 215–225
Nurwidyantoro A, Ho-Quang T, Chaudron MRV (2019) Automated classification of class role-stereotypes via machine learning. In: Proceedings of the evaluation and assessment on software engineering. ACM, pp 79–88
Osman MH, Ho-Quang T, Chaudron M (2018) An automated approach for classifying reverse-engineered and forward-engineered uml class diagrams. In: 2018 44Th euromicro conference on software engineering and advanced applications (SEAA). IEEE, pp 396–399
Palomba F, Tamburri DA, Arcelli Fontana F, Oliveto R, Zaidman A, Serebrenik A (2018) Beyond technical aspects: How do community smells influence the intensity of code smells?. IEEE Trans Softw Eng :1–1. https://doi.org/10.1109/TSE.2018.2883603
Petrenko A, Boroday S, Groz R (2004) Confirming configurations in efsm testing. IEEE Trans Softw Eng 30(1):29–42
Piantadosi V, Fierro F, Scalabrino S, Serebrenik A, Oliveto R (2020) How does code readability change during software evolution?. Empir Softw Eng :1–39
Pourali P, Atlee JM (2018) An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools. In: Proceedings of the 21th ACM/IEEE international conference on model driven engineering languages and systems. ACM, pp 224–234
Prochnow S (2008) Efficient development of complex statecharts. PhD thesis, Christian-Albrechts Universität Kiel
Raghuraman A, Ho-Quang T, Chaudron MR, Serebrenik A, Vasilescu B (2019a) Does uml modeling associate with lower defect proneness?: a preliminary empirical investigation, IEEE
Raghuraman A, Ho-Quang T, Chaudron MRV, Serebrenik A, Vasilescu B (2019b) Does UML modeling associate with lower defect proneness?: a preliminary empirical investigation. In: Storey MD, Adams B, Haiduc S (eds) Proceedings of the 16th international conference on mining software repositories, MSR 2019, 26-27 May 2019, Montreal, Canada. IEEE / ACM, pp 101–104. https://doi.org/10.1109/MSR.2019.00024
Rahad K, Badreddin O, Mohsin Reza S (2021) The human in model-driven engineering loop: a case study on integrating handwritten code in model-driven engineering repositories. Softw Pract Exper
Robles G, Ho-Quang T, Hebig R, Chaudron MR, Fernandez MA (2017) An extensive dataset of uml models in github. In: 2017 IEEE/ACM 14Th international conference on mining software repositories (MSR). IEEE, pp 519–522
Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical Softw Eng 14(2):131
Schaefer G (2006) Statechart style checking–automated semantic robustness analysis of statecharts. PhD thesis, Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik
Sprinkle J, Gray J, Mernik M (2009) Fundamental limitations in domain-specific modeling language evolution. Tech. rep., Technical report, Technical Report# Tr-0908311, University of Arizona

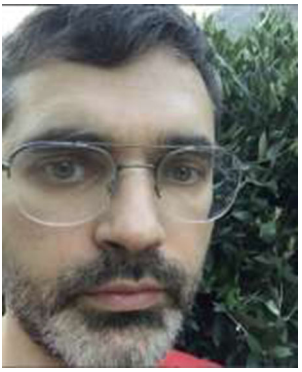
Staron M (2006) Adopting model driven software development in industry–a case study at two companies. In: International conference on model driven engineering languages and systems. Springer, pp 57–72
Steinberg D, Budinsky F, Merks E, Paternostro M (2008) EMF: eclipse modeling framework. Pearson Education, London
Stephan M, Cordy JR (2015) Identifying instances of model design patterns and antipatterns using model clone detection. In: 2015 IEEE/ACM 7th International workshop on modeling in software engineering. IEEE, pp 48–53
Stephan M, Rapos EJ (2019) Model clone detection and its role in emergent model pattern mining. Model Management and Analytics for Large Scale Systems, p 37
Stol KJ, Ralph P, Fitzgerald B (2016) Grounded theory in software engineering research: a critical review and guidelines. In: 2016 IEEE/ACM 38th international conference on software engineering (ICSE). IEEE, pp 120–131
syntok (2014). https://github.com/fnl/syntok
Thomas SW, Hemmati H, Hassan AE, Blostein D (2014) Static test case prioritization using topic models. Empir Softw Eng 19(1):182–212. https://doi.org/10.1007/s10664-012-9219-7
Tolvanen JP, Kelly S (2010) Integrating models with domain-specific modeling languages. In: Proceedings of the 10th workshop on domain-specific modeling. ACM, p 10
Tookkit NL (2014). https://www.nltk.org/
Torres W, van den Brand MG, Serebrenik A (2019) Model management tools for models of different domains: a systematic literature review. In: 2019 IEEE International systems conference (SysCon). IEEE, pp 1–8
Tufano M, Palomba F, Bavota G, Oliveto R, Di PentaM, De LuciaA, Poshyvanyk D (2015) When and why your code starts to smell bad. In: Proceedings of the 37th International conference on software engineering-volume 1. IEEE Press, pp 403–414
Tufano M, Palomba F, Bavota G, Oliveto R, Penta MD, Lucia AD, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Trans Software Eng 43(11):1063–1088. https://doi.org/10.1109/TSE.2017.2653105
van Deursen A, Klint P, Visser J (2000) Domain-specific languages: an annotated bibliography. ACM Sigplan Notices 35(6):26–36
van der Aalst WM (2011) Process mining: discovery, conformance and enhancement of business processes, vol 2. Springer, Berlin
Van Der Straeten R, Mens T, Van BaelenS (2008) Challenges in model-driven software engineering. In: International conference on model driven engineering languages and systems. Springer, pp 35–47
van der Werf JMEM, van Dongen BF, Hurkens CAJ, Serebrenik A (2009) Process discovery using integer linear programming. Fundam Inform 94(3-4):387–412. https://doi.org/10.3233/FI-2009-136
Verum (2014) http://www.verum.com
Whittle J, Hutchinson J, Rouncefield M (2013a) The state of practice in model-driven engineering. IEEE Softw 31(3):79–85
Whittle J, Hutchinson J, Rouncefield M, Burden H, Heldal R (2013b) Industrial adoption of model-driven engineering: Are the tools really the problem? In: International conference on model driven engineering languages and systems. Springer, pp 1–17
Whittle J, Hutchinson JE, Rouncefield M (2014) The state of practice in model-driven engineering. IEEE Softw 31(3):79–85
Wieman R, Aniche MF, Lobbezoo W, Verwer S, van Deursen A (2017) An experience report on applying passive learning in a large-scale payment company, IEEE, ICSME
Wiese A, Ho V, Hill E (2011) A comparison of stemmers on source code identifiers for software search. In: 2011 27Th IEEE international conference on software maintenance (ICSM). IEEE, pp 496–499
Wittgenstein L (2009) Philosophical investigations. Wiley, Hoboken
Xing J, Theelen BD, Langerak R, van dePolJ, Tretmans J, Voeten JP (2010) From poosl to uppaal: Transformation and quantitative analysis. In: 2010 10th International conference on application of concurrency to system design. IEEE, pp 47–56
Yang N, Aslam K, Schiffelers R, Lensink L, Hendriks D, Cleophas L, Serebrenik A (2019) Improving model inference in industry by combining active and passive learning, IEEE
Yang N, Cuijpers P, Schiffelers R, Lukkien J, Serebrenik A (2020) Painting flowers: Reasons for using single-state state machines in model-driven engineering. In: 17th International conference on mining software repositories
Yin RK (1994) Case study research: Design and methods, applied social research. Methods Ser 5
Zhang C, Budgen D (2011) What do we know about the effectiveness of software design patterns? IEEE Trans Softw Eng 38(5):1213–1231

Nan Yang is currently pursuing the Ph.D. degree with the Interconnected Resource-Aware Intelligent Systems (IRIS) Research Cluster at Eindhoven University of Technology in the Netherlands. She is conducting empirical studies to understand software engineering practice for high-tech systems, aimed at proposing software analytics tools for addressing the complexity of such systems. Her research interests include reverse-engineering, log analysis, model-driven engineering, and open source software ecosystems.

Pieter Cuijpers is an Assistant Professor of Quantitative Formal Modelling and Analysis of Cyber-Physical Systems at Eindhoven University of Technology in the Netherlands, and visiting Associate Professor at Aalborg University in Denmark. His current research and educational efforts focus on development of suitable syntactic and semantic models, as well as suitable modeling and analysis methods, aimed at real-time properties of communication standards used in industrial and in-vehicle networking, and computer-aided proof checking of those properties.

Ramon Schiffelers is a Senior Software Architect at ASML, world's leading provider of lithography systems for the semiconductor industry, and an Assistant Professor of Model Driven Software Engineering at Eindhoven University of Technology. His research focuses on theory, methods and tools towards cost effective, industrial scale model-driven system/software engineering. Ramon is positioned at the interface between scientific knowledge and its application in industry. Next to innovative products, this resulted in long-term collaborative research between ASML and academia.

Johan Lukkien is full professor in System Architecture and Networking since 2008. His research interests are in embedded software systems, and in particular in their architecture, evolution and performance. He has been involved in numerous national and international projects on networked systems: software intensive and with resource and timing constraints. Aims of this research include understanding and improving the design trajectory as well as the evolution of such systems. Johan Lukkien is an IEEE senior member.



Alexander Serebrenik is a Full Professor of Social Software Engineering at Eindhoven University of Technology. His research goal is to facilitate evolution of software by taking into account social aspects of software development. He has co-authored a book "Evolving Software Systems" (Springer Verlag, 2014), and more than 100 scientific papers and articles. He has won several distinguished paper and distinguished review awards.

Affiliations

Nan Yang[1] · Pieter Cuijpers[1] · Ramon Schiffelers[1,2] · Johan Lukkien[1] · Alexander Serebrenik[1]

Pieter Cuijpers
p.j.l.cuijpers@tue.nl

Ramon Schiffelers
R.R.H.Schiffelers@tue.nl

Johan Lukkien
j.j.lukkien@tue.nl

Alexander Serebrenik
a.serebrenik@tue.nl

1 Eindhoven University of Technology, Eindhoven, The Netherlands

2 ASML, Veldhoven, The Netherlands