# Painting Flowers: Reasons for Using Single-State State Machines in Model-Driven Engineering

Nan Yang
Pieter Cuijpers
Eindhoven University of Technology
The Netherlands

Ramon Schiffelers
ASML, Eindhoven University of Technology
The Netherlands

Johan Lukkien
Alexander Serebrenik
Eindhoven University of Technology
The Netherlands

## Abstract

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 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.

We study 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 employ the sequential explanatory strategy to study 1500 state machines from 26 components at ASML, a leading company in manufacturing lithography machines from the semiconductor industry. We observe that 25 out of 26 components contain SSSMs, making up 25.3% of the model base. To understand the reasons for this extensive usage we conduct a series of interviews followed by a grounded theory building. The results suggest that SSSMs are used to interface with the existing code, to deal with tool limitations, to facilitate maintenance and to ease verification. Based on our results, we provide implications to modeling tool builders. Furthermore, we formulate two hypotheses about the effectiveness of SSSMs as well as the impacts of SSSMs on development, maintenance and verification.

## CCS Concepts

• **Software and its engineering → Model-driven software engineering**; *Design patterns*; *Maintaining software.*

## Keywords

Model-driven engineering, single-state state machines

**ACM Reference Format:**
Nan Yang, Pieter Cuijpers, Ramon Schiffelers, Johan Lukkien, and Alexander Serebrenik. 2020. Painting Flowers: Reasons for Using Single-State State Machines in Model-Driven Engineering. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3379597.3387452

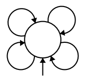
**Figure 1: A flower model (SSSM). The circle represents the single state and the arrows going from and to the same state represent the transitions. The incoming arrow indicates the initial state.**

## 1 INTRODUCTION

Models play a central role in model-driven engineering (MDE) [65]. 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 [38]. 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 [3, 18, 49, 52]. One of the recommendations commonly repeated both in books [3, 17, 18] 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 [3]. Single-state state machines (SSSMs)—affectionately known as "flowers" due to their graphical representation shown in Figure 1—violate this recommendation, yet they are known to have been used, *e.g.,* as models of decision making in conversational agents [34], and in the supervisory control of discrete event systems [15]. 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 [12, 46, 60].

We believe that understanding why a widespread recommendation is not followed in practice is the first step towards improvement of modeling tools and practice. To understand the use of SSSMs in practice, we conduct an exploratory case study at ASML, the leading manufacturer of lithography machines. We employ the sequential explanatory strategy [22]. We first mine 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 report our quantitative findings to software architects and interview them to understand *why they opt for SSSMs* (**RQ3**) and *what advantages and disadvantages of SSSMs they perceive* (**RQ4**).

We observe that SSSMs make up 25.3% of the models considered. These SSSMs are often used with other models as *design patterns* to

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

achieve developers' goals. We identify five such design patterns that are repeatedly used in multiple components. The used SSSMs and design patterns provide industrial evidence on how developers deal with existing code base and tool limitations that are the common problems in MDE adoption [38].

Given ASML has a large portion of its code base developed with the traditional 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 reveal that SSSMs pass verification easily, which is considered as both an advantage and a disadvantage by developers. Based on the analysis of the built theory we compose a set of suggestions for tool builders and researchers.

## 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 [33, 47].

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 [62]. 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 (Figure 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*.

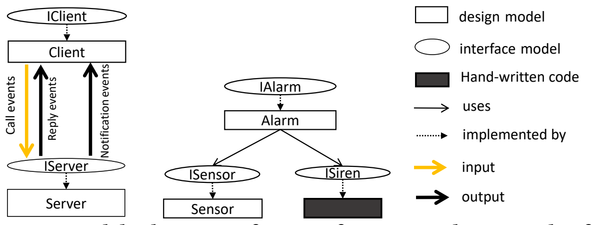

**Figure 2: Model relations. Left: type of events. Right: example of an ASD module . I*** stands for an IM.**

#### 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 verification can be summarized into two steps. First, ASD verifies whether each DM correctly uses the interfaces of its servers. The role of IM in this check is therefore to provide the verification tool with the order the events that should be called. For our alarm module example, ASD checks whether DM *Alarm* calls events in the order specified in IMs *ISensor* and *ISiren*. Second, ASD verifies whether the DM of a component, together with the interfaces of its servers correctly refines the IM of this component under Failures-Divergence Refinement (FDR) [1]. For our alarm module example, it 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.

## 3 CASE STUDY DESIGN

To get a deeper understanding of the use of SSSMs in embedded systems industry, we conducted an exploratory case study. Case study is an empirical method aimed at investigating contemporary phenomena in a context [51, 71].

### 3.1 Industrial Context

We follow the recommendation of Runeson and Höst and intentionally select a case of analysis to serve the study purpose [51]. We conduct our exploratory case study at ASML. The company uses the commercial state machine modeling tool-chain Analytical Software Design (ASD) developed by Verum [62], 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.

### 3.2 Methods

We employed sequential explanatory strategy which consists of a quantitative phase and a qualitative phase [22]. 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 [68], the meaning is determined by use. Thus we exploit structural dependencies (cf. [6, 20]) 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 [27, 35, 45].

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 [54]. 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 6.1.

## 4 PREVALENCE ANALYSIS (RQ1)

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

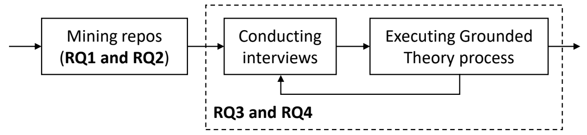

**Figure 3: Overview of our research methods**

### 4.1 Data Analysis

We analyse 1500 ASD models corresponding to components A–Z. We first convert each model into an Ecore model [23] 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) [42] to measure the number of states $\#state$ and the number of state variables $\#sv$. An SSSM is a model with $\#state = 1$ and $\#sv = 0$.

### 4.2 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.

## 5 WHAT ROLE PLAY THE 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.

### 5.1 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 [42] 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. [56]) including tokenization based on common naming conventions such as

**Table 1:** Overview, prevalence of SSSM and frequency of the identified terms for the selected state machine based projects. "-" indicates that the percentage cannot be computed as the component does not include DMs.

| **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 |

**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 |

under_scores, camelCase and PascalCase [55], stemming [67] and removal of stop words and digits using the NLTK package [58]. 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*.

## 5.2 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

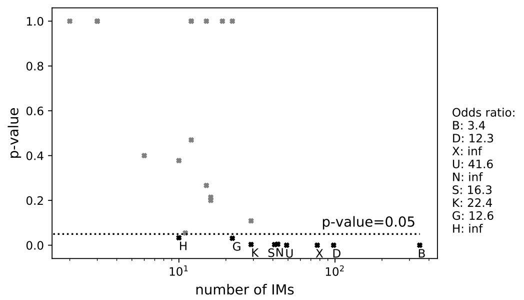

**Figure 4: $p$-values of the Fisher's exact test vs. number of IMs: the null hypothesis is more likely to be rejected for components with more IMs and the odds ratio for each rejected case is larger than one.**

only component B has disconnected models, we exclude *disc* from the statistical analysis. For each component, we construct a $2 \times 2$ 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 [25] rather than a more common $\chi^2$ test: indeed, many components have few IMs and the normal approximation used by $\chi^2$ 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*),

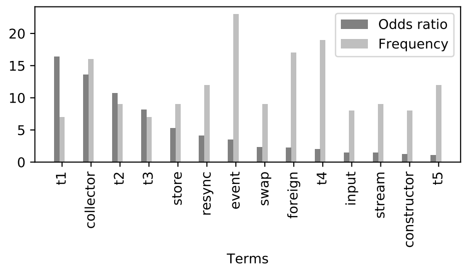

**Figure 5: Frequency and odds ratio of terms that belong to *Shared&OR>1&Frequent* for component B**

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", "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 Figure 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 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". Table 3 summarize 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. 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.

## 6 INTERVIEW (RQ3 AND RQ4)

### 6.1 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 [11], 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 summarises the process we follow.

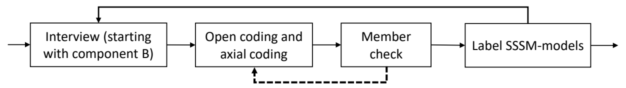
Interview (starting with component B)
Open coding and axial coding
Member check
Label SSSM-models

**Figure 6: Steps in the qualitative phase**

**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 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 refamiliarise 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 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

**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 | barrier | data | error | servic | access | sequenc | measur | t1,...,t14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **#SSSM-IMs** | 16 | 9 | 12 | 23 | 9 | 26 | 8 | 9 | 8 | 10 | 34 | 17 | 8 | 22 | 8 | 10 | 141 |

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 [11] 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 Figure 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.

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

*6.2.1 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 Figure 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.

*6.2.2 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 Figure 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* 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

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

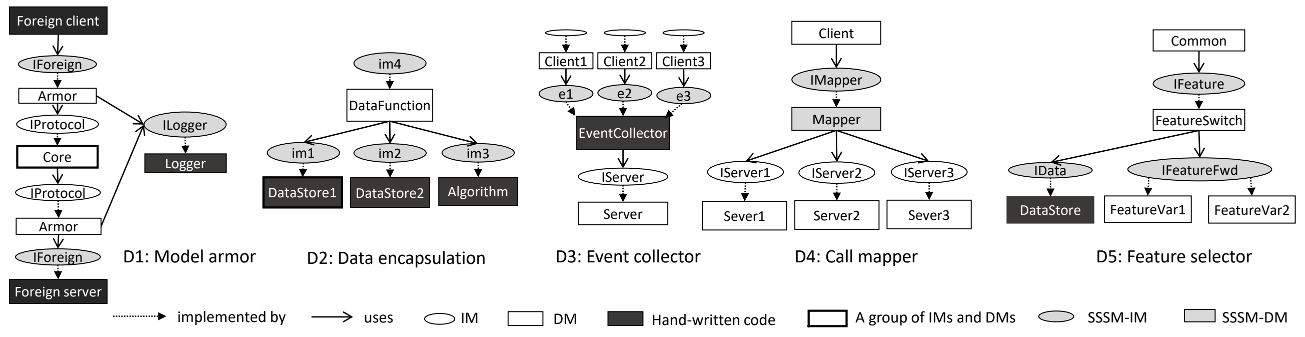

**Figure 7: Identified design patterns D1,...,D5**

**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). We refer the design patterns that involve a set of models to D1,...,D5 as shown in Figure 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.

| **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 |

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 Figure 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*.

*6.2.3 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 Figure 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 [13]. D5 in Figure 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.

*6.2.4 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*.

## 6.3 (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.

## 7 DISCUSSION AND IMPLICATION

We start by relating our findings to literature (Section 7.1), followed by a series of actionable implications for the tool builders (Section 7.2) as well as the researchers (Section 7.3). Finally, we position our work in a broader software engineering context (Section 7.4).

### 7.1 Discussion

We found that developers introduce armoring to interface model-based components with code-based components. This concurs with one of challenges that has been reported to hinder MDE adoption in companies [32, 40, 43, 53]: using MDE together with the existing code base. We provide a *concrete* industrial example illustrating the challenges while interfacing state-machine-based models with the existing code base and showing how developers deal with them.

Moreover, 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. This observation reflects another frequently discussed concern about MDE: the trade-off between general-purpose modeling languages and domain-specific ones [61]. 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 [19].

Apart from dealing with tool limitations, developers invest effort in easing 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 also found two cases where SSSM-IMs are used as documentation for team communication, which is the traditional use case of models [14].

When it comes to the merit and demerit of SSSMs, our interviewees juxtapose the ease of modeling activity and the verification adequacy. As discussed by Chaudron et al. [14], 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 [36]. Our results imply a similar trade-off 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.

Design patterns summarised in Table 1 suggest that developers tend to build reusable designs that can benefit later projects. This observation is consistent with earlier findings on MDE adoption [30] and software engineering practice in general [5].

### 7.2 Implications for Tool Builders

The ease of use and the maturity of tools are identified as the crucial factors for the adoption of MDE [14, 24, 30, 31, 38, 43, 44, 48, 64, 65]. Our work calls for improving modeling tools, and in particular improving the support of integration of models and code-based components. The need to integrate models with the existing code base [31, 38, 64] and to integrate models from different domains [57, 59] 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 [28]. One of the discussed design patterns is the GoF design pattern *Delegation* [26] 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 (Figure 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.

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 6.2.2). 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++ [9].

Finally, we have observed that developers attempt to implement global constants with SSSMs (Section 6.2.2). 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 [8, 16]. Such modeling tools as Uppaal [10] 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 [21]. 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 would be adopting hybrid solutions [21, 69] that translate models from one tool to another, to meet wider verification needs.

### 7.3 Implications for Researchers

As befitting an exploratory case study [51], we propose two 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 6.2 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 [26] have shown that design patterns do not always achieve the claimed advantages [4, 72]. 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 6.3). 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 [14, 50].

Beyond the specific hypotheses, 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 [7, 66, 70], specification mining [37, 39] and process mining [2, 29, 63].

### 7.4 Related Work

Similarly to the growing literature on how and why software developers (do not) follow recommendations or best practices [12, 46, 60], we observe that developers are often forced to deviate from them, e.g., when the desired functionality is solely available in discouraged Eclipse interfaces [12] or when SSSMs are the only way to implement data-dependent behavior in ASD. Compared to the studies of models in industrial practice [38, 41, 43, 65], we provide both quantitative and qualitative analysis of a specific phenomenon in state machine modeling, which results in a set of actionable implications for the tool builders and researchers.

## 8 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 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.

Threats to **external validity** concern the generalizability of our conclusions beyond the studied context. We studied 26 model-based components. We are aware that we limited our study to 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 evidences. 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.

## 9 CONCLUSIONS

We investigated the use of single-state state machines in industrial practice. We employed a sequential explanatory strategy which consists of an analysis of 1500 state machines from 26 components of an embedded system and a series of interviews followed by a grounded theory building. SSSMs make up 25.3% of the model base and are used to interface with the existing code base, deal with tool limitations, facilitate maintenance and ease verification.

Based on our results, we provided a set of implications to tool builders and researchers. Modeling tools should provide a (semi-)automatic way to interface model-based software with existing code base developed with the traditional software engineering. Moreover, modeling tools should reduce the needs of injecting code between models by providing sufficient built-in libraries. Furthermore, we formulated two hypotheses about the effectiveness of SSSM-related design patterns as well as the impacts of SSSMs on development, maintenance and verification.

## 10 ACKNOWLEDGEMENTS

This research was partially supported by ASML Netherlands B.V. and Eindhoven University of Technology, carried out as part of the IMPULS II project. We are very grateful to the ASML software developers and architects who participated in our study and shared their knowledge.

## References

[1] 2014. FDR homepage. http://www.fsel.com.

[2] Wil M.P. van der Aalst. 2011. *Process mining: discovery, conformance and enhancement of business processes.* Vol. 2. Springer.

[3] Scott W. Ambler. 2005. *The elements of UML™2.0 style.* Cambridge University Press.

[4] Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Sofia Charalampidou, and Paris Avgeriou. 2015. The effect of gof design patterns on stability: A case study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 781–802.

[5] Apostolos Ampatzoglou, Apostolos Kritikos, George Kakarontzas, and Ioannis Stamelos. 2011. An empirical investigation on the reusability of design patterns and software packages. *Journal of Systems and Software* 84, 12 (2011), 2265–2283.

[6] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. 1998. Design pattern recovery in object-oriented software. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242).* IEEE, 153–160.

[7] Kousar Aslam, Yaping Luo, Ramon R. H. Schiffelers, and Mark G.J. van den Brand. 2018. Interface protocol inference to aid understanding legacy software components.. In *MODELS Workshops.* 6–11.

[8] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 50.

[9] David M. Beazley. 1996. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++.. In *Tcl/Tk Workshop.* 43.

[10] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. 2006. Uppaal 4.0. (2006).

[11] Eli Buchbinder. 2011. Beyond checking: Experiences of the validation interview. *Qualitative Social Work* 10, 1 (2011), 106–122.

[12] John Businge, Alexander Serebrenik, and Mark G.J. van den Brand. 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, 37–46.

[13] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014), 3–23.

[14] Michel R. V. Chaudron, Werner Heijstek, and Ariadi Nugroho. 2012. How effective is UML modeling? *Software & Systems Modeling* 11, 4 (2012), 571–580.

[15] Yi-Liang Chen and Stéphane Lafortune. 1995. Modular Supervisory Control with Priorities for Discrete Event Systems. In *34th Conference on Decision and Control.* IEEE, 409–415.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2001. Progress on the state explosion problem in model checking. In *Informatics.* Springer, 176–194.

[17] Javier de San Pedro and Jordi Cortadella. 2016. Mining structured petri nets for the visualization of process behavior. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, Sascha Ossowski (Ed.). ACM, 839–846.

[18] Alan Dennis, Barabara Haley Wixom, and David Tegarden. 2009. *Systems Analysis and Design UML Version 2.0.* Wiley.

[19] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 35, 6 (2000), 26–36.

[20] Jing Dong, Dushyant S Lad, and Yajing Zhao. 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, 371–380.

[21] Richard Doornbos, Jozef Hooman, and Bernard van Vlimmeren. 2012. Complementary verification of embedded software using ASD and Uppaal. In *2012 International Conference on Innovations in Information Technology (IIT).* IEEE, 60–65.

[22] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering.* Springer, 285–311.

[23] Ecore. [n.d.]. https://www.eclipse.org/modeling/emf/.

[24] Kleinner Farias, Alessandro Garcia, Jon Whittle, and Carlos Lucena. 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, 639–655.

[25] Ronald A. Fisher. 1922. On the interpretation of $\chi$ 2 from contingency tables, and the calculation of P. *Journal of the Royal Statistical Society* 85, 1 (1922), 87–94.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming.* Springer, 406–431.

[27] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. 2013. A comparative analysis of software architecture recovery techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press, 486–496.

[28] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. 2015. Integration of handwritten and generated object-oriented code. In *International Conference on Model-Driven Engineering and Software Development.* Springer, 112–132.

[29] Monika Gupta, Atri Mandal, Gargi Dasgupta, and Alexander Serebrenik. 2018. Runtime Monitoring in Continuous Deployment by Differencing Execution Behavior Model. In *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings (Lecture Notes in Computer Science)*, Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu (Eds.), Vol. 11236. Springer, 812–827. https://doi.org/10.1007/978-3-030-03596-9_58

[30] John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming* 89 (2014), 144–161.

[31] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *Proceedings of the 33rd international conference on software engineering.* ACM, 471–480.

[32] Rodi Jolak, Truong Ho-Quang, Michel R.V. Chaudron, and Ramon R. H. Schiffelers. 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.* 213–223.

[33] Young Gon Kim, Hyoung Seok Hong, Doo-Hwan Bae, and Sung Deok Cha. 1999. Test cases generation from UML state diagrams. *IEE Proceedings-Software* 146, 4 (1999), 187–192.

[34] Fredrik Kronlid. 2006. Turn Taking for Artificial Conversational Agents. In *Cooperative Information Agents X, 10th International Workshop (Lecture Notes in Computer Science)*, Matthias Klusch, Michael Rovatsos, and Terry R. Payne (Eds.), Vol. 4149. Springer, 81–95.

[35] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49, 3 (2007), 230–243.

[36] Christian F.J. Lange, Michel R.V. Chaudron, and Johan Muskens. 2006. In practice: UML software architecture and design description. *IEEE software* 23, 2 (2006), 40–46.

[37] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 81–92.

[38] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. 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, 166–182.

[39] David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. 2011. *Mining software specifications: methodologies and applications.* CRC Press.

[40] Anthony MacDonald, Danny Russell, and Brenton Atchison. 2005. Model-driven development within a legacy system: an industry experience report. In *Australian Software Engineering Conference.* IEEE, 14–22.

[41] Josh Mengerink, Alexander Serebrenik, Ramon R. H. Schiffelers, and Mark G. J. van den Brand. 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, IWSM-Mensura 2017, Gothenburg, Sweden, October 25 - 27, 2017*, Miroslaw Staron and Wilhelm Meding (Eds.). ACM, 116–121. https://doi.org/10.1145/3143434.3143442

[42] Josh G. M. Mengerink, Alexander Serebrenik, Ramon R. H. Schiffelers, and Mark G. J. van den Brand. 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, 116–121.

[43] Parastoo Mohagheghi and Vegard Dehlen. 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, 432–443.

[44] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A. Fernandez. 2013. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering* 18, 1 (2013), 89–116.

[45] Arif Nurwidyantoro, Truong Ho-Quang, and Michel R. V. Chaudron. 2019. Automated classification of class role-stereotypes via machine learning. In *Proceedings of the Evaluation and Assessment on Software Engineering.* ACM, 79–88.

[46] Fabio Palomba, Damian Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. 2018. Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells? *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2883603

[47] Alexandre Petrenko, Sergiy Boroday, and Roland Groz. 2004. Confirming configurations in EFSM testing. *IEEE Transactions on Software engineering* 30, 1 (2004), 29–42.

[48] Parsa Pourali and Joanne M Atlee. 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, 224–234.

[49] Steffen Prochnow. 2008. *Efficient development of complex statecharts.* Ph.D. Dissertation. Christian-Albrechts Universität Kiel.

[50] Adithya Raghuraman, Truong Ho-Quang, Michel R. V. Chaudron, Alexander Serebrenik, and Bogdan Vasilescu. 2019. Does UML modeling associate with lower defect proneness?: a preliminary empirical investigation. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 101–104. https://doi.org/10.1109/MSR.2019.00024

[51] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131.

[52] Gunnar Schaefer. 2006. *Statechart style checking–automated semantic robustness analysis of Statecharts.* Ph.D. Dissertation. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik.

[53] Miroslaw Staron. 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, 57–72.

[54] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).* IEEE, 120–131.

[55] syntok. 2014. https://github.com/fnl/syntok.

[56] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. 2014. Static test case prioritization using topic models. *Empirical Software Engineering* 19, 1 (2014), 182–212. https://doi.org/10.1007/s10664-012-9219-7

[57] Juha-Pekka Tolvanen and Steven Kelly. 2010. Integrating models with domain-specific modeling languages. In *Proceedings of the 10th Workshop on Domain-specific Modeling.* ACM, 10.

[58] Natural Language Tookkit. 2014. https://www.nltk.org/.

[59] Weslley Torres, Mark G.J. van den Brand, and Alexander Serebrenik. 2019. Model management tools for models of different domains: a systematic literature review. In *2019 IEEE International Systems Conference (SysCon).* IEEE, 1–8.

[60] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Trans. Software Eng.* 43, 11 (2017), 1063–1088. https://doi.org/10.1109/TSE.2017.2653105

[61] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. 2008. Challenges in model-driven software engineering. In *International Conference on Model Driven Engineering Languages and Systems.* Springer, 35–47.

[62] Verum. 2014. http://www.verum.com.

[63] Jan Martijn E. M. van der Werf, Boudewijn F. van Dongen, Cor A. J. Hurkens, and Alexander Serebrenik. 2009. Process Discovery using Integer Linear Programming. *Fundam. Inform.* 94, 3-4 (2009), 387–412. https://doi.org/10.3233/FI-2009-136

[64] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. 2013. Industrial adoption of model-driven engineering: Are the tools really the problem?. In *International Conference on Model Driven Engineering Languages and Systems.* Springer, 1–17.

[65] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. 2014. The State of Practice in Model-Driven Engineering. *IEEE Software* 31, 3 (2014), 79–85.

[66] Rick Wieman, Maurício Finavaro Aniche, Willem Lobbezoo, Sicco Verwer, and Arie van Deursen. 2017. An experience report on applying passive learning in a large-scale payment company. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 564–573.

[67] Andrew Wiese, Valerie Ho, and Emily Hill. 2011. A comparison of stemmers on source code identifiers for software search. In *2011 27th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 496–499.

[68] Ludwig Wittgenstein. 2009. *Philosophical investigations.* John Wiley & Sons.

[69] Jiansheng Xing, Bart D. Theelen, Rom Langerak, Jaco van de Pol, Jan Tretmans, and Jeroen P.M. Voeten. 2010. From POOSL to UPPAAL: Transformation and quantitative analysis. In *2010 10th International Conference on Application of Concurrency to System Design.* IEEE, 47–56.

[70] Nan Yang, Kousar Aslam, Ramon Schiffelers, Leonard Lensink, Dennis Hendriks, Loek Cleophas, and Alexander Serebrenik. 2019. Improving model inference in industry by combining active and passive learning. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 253–263.

[71] Robert K. Yin. 2003. Design and methods. *Case study research* 3 (2003).

[72] Cheng Zhang and David Budgen. 2011. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering* 38, 5 (2011), 1213–1231.