مجلـة العلوم
الهندسية والتقنية
مجلة علمية نصف سنوية محكمة - تصدر عن كلية الهندسة وكلية الحاسبات والمعلوماتية – جامعة ذمار

Validating Quality Metrics of State Machine Models

Ammar Osaiweran

Faculty of Computers and Informatics, Thamar University, Thamar, Yemen
ammar.osaiweran@tu.edu.ye

Abstract:
Software metrics are widely used to measure the quality of software and to give an early indication of the efficiency of the development process in industry. There are many well-established frameworks for measuring the quality of source code through metrics, but limited attention has been paid to the quality of software models. In this article, we introduce new metrics that are tailored to measure the quality of models of state machines and then apply the metrics to evaluate the quality of state machine models specified using the Analytical Software Design (ASD) tooling. We discuss how we applied a number of metrics to ASD models in an industrial setting and report about results and lessons learned while collecting these metrics. Furthermore, we recommend some quality limits for each metric and validate them on models developed in a number of real industrial projects. This paper extends [19] by providing a formal and empirical validation of the metrics and their related limits. The results of our work provide a framework to measure the quality of state machine models, developed in ASD, and give a basis for future research on introducing quality metrics for other type of models of which quality metrics are missing.

Keywords: Software engineering, Model-based development, software quality, Model ttransformation, Software development

1. Introduction

The use of model-based techniques in software development processes has been promoted for many years [15,2,7,3]. The aim is to use the models as the main software artifacts in the development process, not only for visualization and communication among developers, but also as means of specification, formal verification, code generation, testing and validation.

In traditional development, source code is the main software artifact. To measure the quality of source code, a number of widely used metrics are utilized, with well-established industrial strength tools and frameworks, such as TICS [17], CodeSonar [4] and VerifySoft [19]. Code metrics are useful means to detect decays and code smells [9] that hinder future evolution and maintenance.

However, these frameworks and tools cannot be applied directly to measure the quality of models. They can measure the generated code, but it is debatable whether this is meaningful. This is because, usually, code generators generate correct and optimal source code tailored to a specific domain and the generated code is often excluded from code analysis tools due to violations and non-adherence to the prescribed coding standards. Therefore, complexity, duplication and other undesired properties must be analysed at the level of models. Since industry is becoming more reliant on software models, there is an urgent need to establish a way for measuring various metrics at the level of models and not at the level of source code.

In our industrial context, we use state machines to design and specify reactive and control aspects of software using a lightweight formal modeling tool called ASD: Suite [18]. The tool allows modeling of state machines in a tabular format. These models can be formally verified and corresponding source code can be generated from these models. Because there are no means to measure the quality of these models, a number of challenging questions are raised. How can we evaluate the quality of this type of state machine models? Are some of the models developed in early projects in our industrial setting overly complex? Which factors contribute to the complexity of models? How can these factors be detected and measured? How can we help engineers to improve the quality of their future models? How can we provide to modelers information on deterioration as their models evolve?

In this paper we provide answers to the above questions by utilizing a number of software metrics that we tailored and adapted for measuring the quality of ASD models. Another challenging research question is how to validate the correctness of these metrics with respect to the perception of software engineers who developed the models. This is because a complex model from a perspective of an engineer may be observed as a simple model by another engineer. To address this challenge, we conducted validation and verification steps to proof the soundness of the metrics using empirical and formal validation steps. The validation also includes the proposed thresholds of the metrics [20].

This paper extends a previously published article [19] with these validation and verification steps of the metrics.

The paper is structured as follows. Section 2 discusses related work on metrics of state machines. Section 3 introduces ASD to the extent needed for this article. In Section 4 a number of well-known software metrics are detailed with the application to ASD models. Section 5 introduces recommended limits of metrics for good quality models. Section 6 details the data collection process of metrics from models and discusses observations during the data analysis. In Section 7 we detail out the verification and validation steps formally and empirically. In Section 8 we conclude our paper highlighting the limitations of our metrics and future work in this regard.

2. Related Work

In previous research at Philips Healthcare [16], guidelines for readability and verifiability of ASD models were introduced. An important guideline is for instance: an ASD tabular model should not include more than 250 rows leading to not more than 3000 lines of generated code. The limitation of this guideline is that it considers only the size of models and generated code while no other complexity factors were addressed. To estimate the reliability of UML state machines, and to identify failure-prone components, a group of authors [12] measured the cyclomatic complexity of UML state machines. They did not measure the CC directly on state machines, but on the control flow graph generated from their software realization. Similarly, other authors focus on assessing the number of tests. For example, in [8] decision diagrams as intermediate artifacts were used to calculate the number of tests for the code of concurrent state machines. In [25] they provide a metric based on complexity rate for each element in the state machine model but they did not provide metrics to evaluate the entire state machine from different aspects. In [26], the authors surveyed quality metrics of requirements in Agile and rapid development but they did not consider metrics and quality of models in their study.

3. ANALYTICAL SOFTWARE DESIGN

This section provides a short introduction of the ASD approach and its toolset, the ASD: Suite [18]. Using the ASD: Suite, models of components and interfaces can be described. Two types of models are distinguished which are both state machines specified by a tabular notation: ASD interface models and ASD design models.

The external behaviour of a component is specified using an interface model which excludes any internal behaviour not seen by client components that use the interface. The interface model is implemented by a design model which typically uses the interfaces of other so-called server components.

An ASD component includes an implemented interface model, a design model, and optional server interface models. Formal verification is established by verifying that calls in design models to interfaces of server components are correct, with respect to contracts of the servers. For this ASD uses CSP/FDR2 [11], [6] for model checking by

exhaustively searching for illegal interactions, deadlocks or livelocks in the behaviour. It is also formally checked whether the behaviour of the design model obeys its implemented interface model.

The ASD tool also provides the modeler with elementary metrics related to the generated state space such as the number of states and transitions and the time required for verification in seconds. Besides formal verification, the ASD: Suite allows code generation to a number of languages (C, C++, C#, Java).

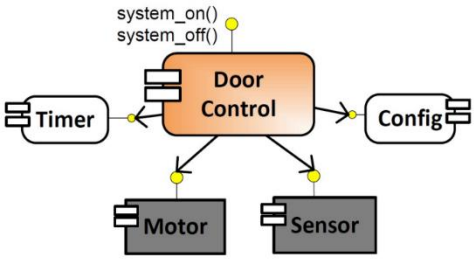

Fig. 1. Example controller system of automatic door

In ASD, a client issues synchronous calls to server components, whereas a server sends asynchronous callbacks to its clients. These callbacks are non-blocking and can be received by a component at any time.

We detail the ASD specification by using a small automatic Door controller example. It consists of a *Door* controller component that controls a *Sensor* and a *Motor* component, see Figure 1. The Controller receives two requests from external clients, namely *systemOn* to start-up the system and *systemOff* to shutdown the system. When the system is *ON*, the controller may receive a callback from the sensor component when there is a detected object. Upon such an event, it issues a command to the motor component to open the door and apply a brake. Then it starts a timer and when it times out the controller issues a command to release the brake to close the door. This example is used to clarify and illustrate the interface model in Section 3.1 and the design model in Section 3.2.

3.1 ASD Interface Models

The interface model is the first artifact that must be specified when creating an ASD component. It describes the external behavior of the component by means of the allowed sequence of calls and callbacks, exchanged with clients. Any internal behavior not visible to clients is abstracted from the interface specification.

Figure 2 depicts the tabular specification of an ASD interface model. The specification lists all implemented interfaces, their events (also called input stimuli), guards or predicates on the events. A sequence of response actions can be specified in the

Actions list such as return values or callbacks to clients, and special actions such as *Illegal* which essentially marks the corresponding event as not allowed in that state.

In Figure 2 the interface specification of the *Door* controller is described. The model contains two states: *Off* and *On*. Any ASD model must be complete in the sense that actions for all input stimuli events must be defined in every state. For example, in row 3 a *systemOn* event is accepted and the component will transit to state *ON* after returning a *voidReply* to *IDoorControlAPI*. In row 4 and 7 of Figure 2 the *Illegal* action is specified denoting that invoking the event is forbidden by clients. Once in the *On* state, the component accepts a *systemOff* request and transits back to the *Off* state. Similarly, Figure 3 depicts the external behavior of the

	Interface	Event	Guard	Actions	State	Target State
1	Off (initial state)					
3	IDoorControlAPI	systemOn		IDoorControlAPI.VoidReply		On
4	IDoorControlAPI	systemOff		Illegal		-
5	On					
7	IDoorControlAPI	systemOn		Illegal		-
8	IDoorControlAPI	systemOff		IDoorControlAPI.VoidReply		Off

Fig. 2. Interface model of door controller

Sensor hardware component, which is strictly alternating between the *Active* and *Inactive* states via the *startSensing* and *stopSensing* events. In row 10, a so-called internal event is specified denoting that something internal in the device can happen, which is in this case a *detectedMovement*. As a consequence, the *detectedObject* callback is sent to the controller and the *Sensor* remains in the *Active* state. Via internal events, the interface abstracts from one or more actions that happen internally in the implementation.

	Interface	Event	Guard	Actions	Stat	Target State
1	Inactive (initial state)					
3	ISensor	startSensing		ISensor.VoidReply		Active
4	ISensor	stopSensing		Illegal		-
6	Active					
8	ISensor	startSensing		Illegal		-
9	ISensor	stopSensing		ISensor.VoidReply		Inactive
10	ISensorINT	detectedMovement		ISensorNI.detectedObject		Active

Fig. 3. Sensor interface model

3.2 ASD Design Models

The ASD design model implements the interface model and extends it with more detailed internal behavior. The model includes calls to other interface models of other components.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	Off (initial state)					
4	IDoorControlAPI	systemOn		IDoorControlAPI.VoidReply; config:IConfigAPI.getConfiguration(>>speed, >>time); sensor:ISensor.startSensing		DoorClose
8	DoorClose					
12	IDoorControlAPI	systemOff		IDoorControlAPI.VoidReply; sensor:ISensor.stopSensing		Off
13	sensor:ISensorNI	detectedObject		motor:IMotorAPI.motorOn(<<speed); timer:ITimerAPI.startTimer(<<time)		DoorOpen
15	DoorOpen					
19	IDoorControlAPI	systemOff		IDoorControlAPI.VoidReply; sensor:ISensor.stopSensing; motor:IMotorAPI.releaseBrake; timer:ITimerAPI.stopTimer		Off
20	sensor:ISensorNI	detectedObject		timer:ITimerAPI.startTimer(<<time)		DoorOpen
21	timer:ITimerNI	timeOut		motor:IMotorAPI.releaseBrake		DoorClose

Fig. 4. Design model of door controller. *Illegal* events are hidden

Figure 4 depicts the design model of the *Door* controller. The specification refines the interface model of Figure 2 with all required internal details and uses the interface models of other components such as the Sensor interface model of Figure 3. For example, row 4 specifies that when the Door component receives a *systemOn* request, it does not only return *voidReply* to the client, as specified in the interface model, but it also calls a configuration component via the *getConfiguration* action and asks the Sensor hardware to start monitoring the surroundings via the *startSensing* action. After that, the controller transits to the *DoorClose* state. Note that, the call to the configuration is supplied with 2 data parameters namely, speed and time. When the call returns, the component stores their values in the local storage parameters of the component using the >> operator, to be retrieved later when needed via << operator. The rest of the specification is self-explanatory.

An example of processing a callback is depicted in row 13 and 21 where the component may receive a *detectedObject* and a *timeOut* callback from the *Sensor* and the *Timer* components respectively

4. TAILORING CODE METRICS FOR ASD MODELS

To measure the quality of ASD models, we tailored a number of metrics that are widely used in industrial practice for measuring the quality of source code like the McCabe and Halstead complexity metrics [13], [10]. In this section we introduce these metrics and discuss how we adapt them to measure ASD design and interface models.

We start by introducing the McCabe cyclomatic complexity metric (CC) and its application to measure complexity of ASD models. Then, we introduce our tailored version of the CC metric along with its application to ASD models. We discuss how both

metrics complement each other and how they provide more insights on the complexity of the models. After that we introduce Halstead metrics detailing how they are adapted to measure ASD models.

4.1 Cyclomatic complexity of ASD models

The cyclomatic complexity (CC) metric provides a quantitative measure on the number of linearly independent paths in source code of a program, represented by a control flow directed graph [13]. At the time the CC metric was developed, the main purpose was to calculate the minimum number of test cases required to test the independent paths of a program. When the CC metric is high it indicates not only that the number of related test cases is high but also that the program itself is hard to read and understand by developers.

To calculate the CC of source code, the program should first be represented as a connected graph. For example, Figure 5 depicts a function foo and its graph representation. The CC of a program can be calculated using the following equation:

$CC = E - N + 1,$

where E denotes the number of edges in the graph and N is the total number of nodes.

The CC of the code presented in Figure 5 is: 5 − 5+1 = 1.

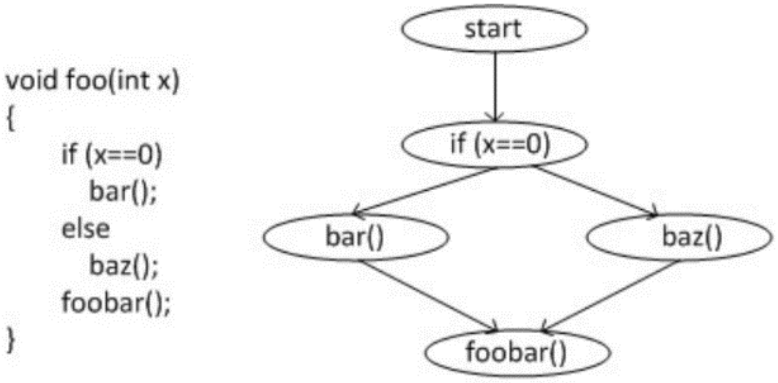

Fig. 5. Code and its graph representation

In a similar way, we can use CC for code as a basis to calculate the CC of ASD models. The tabular notation of ASD models can also be seen as a directed graph that contains edges and nodes. Note that, for ASD components we are mainly concerned with the understandability aspect of ASD components rather than testing effort since model checking replaces testing and guarantees that all paths of a model are exhaustively and fully checked. Testing efforts can be of a concern for non-ASD components since their implementation is handcrafted.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	X (initial state)					
3	IF	a1		IF.VoidReply		Y
4	IF	a2		IF.VoidReply		Y
5	IF	a3		IF.VoidReply		Y
8	Y					
13	IF	a4		IF.VoidReply		Y
14	IF	a5		IF.VoidReply		Y

Fig. 6. An ASD interface model with 2 states and 5 transitions

To illustrate how CC can be collected for ASD models, consider the specification depicted in Figure 6. The specification consists of 2 states namely state *X* and state *Y*. In state *X*, the machine accepts events *a1*, *a2* and *a3* via the *IF* interface and then moves to state *Y*. The machine stays in state Y forever accepting a4 and a5 events.

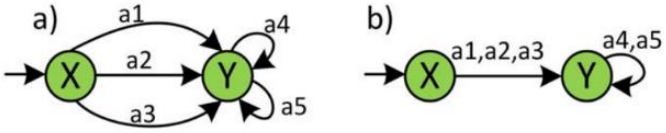

Fig. 7. a) Graphical representation with independent edges for events. b) Graph with unique edges with set of actions

The graphical representation of the ASD state machine is depicted in Figure 7.a. The CC of this model can be calculated as follows:

E = 5, N = 2,

CC = 5 − 2+1=4

Application to the Door models

The CC of the Door interface model depicted in Figure 2 is 1, while the CC of the design model depicted in Figure 4 is 4. The CC of the *Sensor* interface model of Figure 3 is 2.

4.2 Actual (structural) complexity

We tailored the CC metric to collect the so called Actual (or structural) complexity (ACC) of a model. With the ACC metric we group edges between states. If there are multiple edges between certain states, we only count them as one. This means that in ACC any edge may contain one or more events (a set of events) while in CC each edge has only one event. For example, in Figure 7b, it is possible to transit from state X to

state Y via either a1, a2 or a3 events (one transition labeled by a set of events). In state *Y* only *a4* or *a5* events are accepted.

Note that, the ACC metric does not replace CC but it complements it by providing additional insight to complexity. It groups events that have similar transitions and identical effect on a state. The metric gives an indication on how complex and difficult it is for a human to read and to understand the model through navigating and memorizing the history of states. The metric is not concerned with the number of tests required to exercise the state machine. ACC can be calculated using the following equation:

ACC = EU - N + 1,

where EU denotes the total number of unique edges and N is the total number of nodes. For instance, the ACC of the ASD state machine depicted earlier in Figure 6a can be calculated as follows:

EU = 2, N = 2,

ACC = 2 − 2+1=1

Application to the Door models

The ACC of the Door interface model depicted in Figure 2 is 1, while the ACC of the design model depicted in Figure 4 is 4. The ACC of the Sensor interface model of Figure 3 is 2.

4.3 Halstead, LoC and maintainability index

Using Halstead approach, metrics are collected based on counting operators and operands of source code [10]. We introduce these metrics and discuss how we tailored them to ASD models. Furthermore, we show how the lines of code metric and the maintainability index are collected.

We start by introducing Halstead metrics. The metrics measure the cognitive load of a program which is the mental effort used to understand, maintain and develop the program. The higher the load, the more time it takes to design or understand it, and the higher the chances of introducing bugs. Halstead considered programs as implementation of algorithms, consisting of operators and operands. His metrics are designed to measure the complexity of any kind of algorithms regardless of the language in which they are implemented. Halstead metrics use the following basis measures:

and only if there is a sequence $\sigma \in I^*$ distinguishes *M* and *N*, $AM(\sigma) \neq AN(\sigma)$.

- n1: the number of unique operators,
- N1: the number of occurrences of operators,
- n2: the number of unique operands,

- N2: the number of occurrences of operands,
- n = n1 + n2: the model vocabulary,
- N = N1 + N2 the length of the model.

For any ASD model we consider the following to be operands:

- state variables used as guards,
- states of the state machine,
- data variables in events and actions.

We consider the following to be operators:

- events (calls, internal events and stimuli call-backs) and actions (all responses including return values and call-backs),
- operators on state variables such as not, and, or, >, < (value of variable is stored and retrieved), and $ (literal value).

The basic measures are then used to calculate the metrics below:

- Volume: V = N ∗ log2n,
- Difficulty: D = (n1/2) ∗ (N2/n2),
- Effort: E = D ∗ V denotes the effort spent to make the model,
- Time required to understand the model: T = (E/18) (seconds),
- Expected number of Bugs: B = V /3000.

The volume metric V considers the information content of a program as bits. Assuming that humans use binary search when selecting the next operand or operator to write, Halstead interpreted volume as a number of mental comparisons a developer would need to write a program of length N.

Program difficulty D is based on a psychology theory that adding new operators, while reusing the existing operands increases the difficulty to understand an algorithm.

Program effort E measures the mental effort required to implement or comprehend an algorithm. It is measured in elementary mental discriminations. For each mental comparison (and there are V of them), depending on the difficulty, the human mind will perform several elementary mental discriminations. The rate at which a person performs elementary mental discriminations is given by a Stroud number that ranges between 5 and 20 elements per second. Halstead empirically determined that in the calculation of the time T to understand an algorithm this constant should be adjusted to 18.

Finally, the estimated number of bugs B correlates with the volume of the software. The more the size increases, the more the likelihood to introduce bugs. Halstead empirically calculated the estimated number of bugs by a simple division by 3000.

We calculate the lines of code metric based on not only the total number of rows in the model but also the number of actions in the Actions list. Therefore, each action counts as 1 line. For instance, the specification of the Door interface model contains 4 LoC.

The original maintainability index (MI) of source code is calculated based on V, LoC and CC of the source code [5]. It indicates whether it is worth to keep maintaining, modifying and extending a program or to immediately consider refactoring or redesigning it.

Microsoft incorporated the MI in the Microsoft Studio environment. We used the formula of Microsoft to calculate the MI of ASD models. The formula is defined as follows:

$$MI = MAX(0,(171 - 5.2 * \ln(V) - 0.23 * ACC - 16.2 * \ln(LoC)) * 100/171)$$

The formula produces a number between 0 and 100, where 20 or above indicates good and highly maintainable source code.

Application to the Door models

Table 1 lists the volume (V), expected number of bugs (B), difficulty (D) and time (T in seconds) metrics of the three ASD models of the Door system.

Model	*V*	*B*	*D*	*T* (sec)	*LoC*	*MI*
Door interface	33	0.01	2	4	4	76
Door design	236	0.08	16	210	19	55
Sensor interface	56	0.02	4	13	6	70.5

Table 1: METRICS OF DOOR CONTROLLER MODELS

The table is self-explanatory. Notable is the time required to understand the models. The reader of this paper is expected to read and understand the specification of the Door design model in about 210 seconds. All models exhibit a maintainability index of 20 and above, hence they are highly maintainable. The rest of the data provided in the table is self-explanatory.

5. OPTIMAL VALUES AND RECOMMENDED LIMITS OF METRICS

In this section, we propose limits of metrics for good quality interface and design models. The limits were established after carefully analyzing and reviewing over 615 interface and design models built for a large photolithography system, developed by ASML [1]. The limits were proposed after iterative review meetings and alignments with various engineers who owned and developed the models.

Metric	Limit of metric		
	Low	Moderate	High
CC	≤ 30	≤ 50	> 50
ACC	≤ 20	≤ 40	> 40
V	≤ 8000	≤ 14000	> 14000
LoC (IM)	≤ 200	≤ 400	> 400
LoC (DM)	≤ 500	≤ 800	> 800
MI	≤ 10	≤ 20	> 20
VT	≤ 1 min	≤ 5 min	> 5 min

Table 2 OPTIMAL VALUES OF METRICS FOR ASD MODELS

Table 2 lists all metrics and the advised limits in our industrial context. As depicted in the table, the limits of the metrics for interface and design models are similar except for the LoC metric.

In our industrial context, the CC of a module written in C++ should not exceed 10. If source code exhibits a CC between 10 to 40 then the code should be refactored while if it is more than 40 then the code is end-of-life and has to be rewritten again in a simpler way. This CC limit may vary from one organization to another.

The reason that the limits of CC for models are raised compared to the CC for source code is that the metrics are collected at the level of models. We found that the tabular representation of the model raises the abstraction level and increases the understandability of the software artifact compared to source code. Models with a CC less than 30 were easy to understand when reviewing the tabular format of the models.

Similarly, designers were reasonably comfortable reviewing models that exhibit an ACC of less than 20. For the size metric, we used the limit suggested by VerifySoft [19] and observed that models exceeding 8000 are big in size. Finally, the thresholds of MI were chosen as used by Microsoft.

In our industrial context, we recommend that verification time (or waiting time for the model checker during debugging) should not exceed 1 minute. The reason is that we want to prevent that productivity of developers is hindered by the model-checking technology.

Design and modeling are creative processes and having good metrics of a model does not always mean that the underlying design is good. It is possible that certain models exhibit metrics within the accepted limits while mixing the level of abstractions with inappropriate decomposition of components and mixed responsibilities. While metrics can help detecting bad smells and decays in early design phases, additional experts' reviews are still needed to assess the overall design quality.

6. DETAILED DATA ANALYSIS

In this section we detail the application of the proposed metrics and the recommended limits to measure and evaluate the existing ASD models, see Table 3. In order to make the process of data analysis and collection of the models more efficient, we built a tool that automatically extracts the metrics and visualize the results graphically. The tool is compatible with ASD:Suite version 9.2.7. We used the tool to extract metrics from 615 ASD interface and design models, developed in four different projects, within the period of 2008 until the end of 2015.

Metric	Interface Models	Design Models
# of models	348	267
Average CC	18	39.4
Average ACC	4.5	11
Total Volume	204,593	3,533,640
Total LoC	12,580	205,772
Total C++ LoC	55,710	611,724

Table 3 SUMMARY OF STATISTICAL DATA OF DEVELOPED MODELS

Table 3 provides collected metrics data about the models. The total number of interface models is 348 while there are 267 design models. Row 3 and 4 list the average CC and ACC measures for the models. In row 5 the total volume or size of models is depicted. Row 6 lists the total number of lines of code in the models while the last row lists the total number of lines of the generated C++ code excluding blank lines.

Metric	Limit	Interface models	Design models	Percen-tage
CC	≤ 30	299	178	77.56%
	$(30, 50)$	24	26	8.13%
	> 50	25	63	14.31%
ACC	< 20	333	231	91.71%
	$(20, 40)$	7	17	3.9%
	> 40	8	19	4.4%
V	$< 8K$	344	181	85.37%
	$(8K, 14K)$	3	17	3.25%
	$> 14K$	1	69	11.4%
LoC	< 200	338	182	84.55%
	$(200, 400)$	5	14	3.08%
	> 400	5	71	12.36%
VT	< 1 min	348	266	99.84%
	(1 min,5 min)	0	1	0.16%
	> 5 min	0	0	0%

Table 4 ANALYSIS OF METRICS VALUES

We separated ASD interface models from design models and then carefully evaluated them in isolation. After that, we ordered the models according to CC, ACC and volume, to sort the models based on their complexity and size. The purpose of sorting the models is to capture the complex and big models that are present in our archive to refactor and improve these models. The data analysis of these models is summarized in Table 4.

In summary, the analysis revealed that over 22% of the models are relatively complex based on the CC metric and the models should be refactored to reduce

complexity. Considering the ACC metric over 10% of the models should be refactored to simpler models. We discuss the relation between CC and ACC shortly. With respect to size, we considered the volume and LoC metrics. Over 15% of the models are big in size and should be split into smaller models. Similarly, over 15% of the models include many lines of code. Most of these big models exhibit also high complexity metrics; therefore, improving one metric will consequently improve the other metrics.

All models were verified in less than 1 minute except one model which took about 5 minutes from the model checker. This model is also the biggest and the most complex model compared to others. The reason that all models were verified in a short time is that the execution of the components is configured to be single-threaded; therefore, there is no concurrency that leads to the generation of big state spaces.

The data and results of our analysis are communicated to the development teams together with the metric extraction tool to facilitate repeating the experiments. The teams appreciated the work since it helped them uncover hidden complex and big models. A team of one of the projects planned refactoring tasks to gradually improve the quality of complex models. For newly started projects, developers frequently check the metrics of their models to address any issue early during the modeling phase and before final delivery of the models.

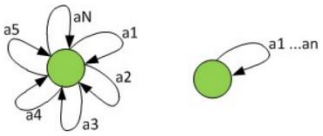

Fig. 8. Representing a stateless machine as a flower-shape (CC) or a mouse ear (ACC)

One observation during the data analysis is that not all models with high CC are really complex to understand. We discuss this observation by comparing CC and ACC of an example specification and discuss how the ACC metrics provided more insight in complexity. Consider Figure 8. At the left of the figure a stateless machine accepts N events. If we set N to 31 (meaning that 31 different events are accepted by the machine) then CC = 31 while ACC = 1. Therefore, from the CC perspective the state machine is considered to be moderate in complexity since it exceeded the complexity limit, we set before as a guideline.

In fact, all models that exhibit a flower-shape behavior are not very complex but they may be rather big because the interface is verbose with many events. These machines are relatively simple to understand since they just consume input events in a single state. This type of models exhibits a relatively very low ACC metric. Correlating CC and ACC can help developers detecting interfaces that include many different events

that have actually the same behavior. In hindsight, it indicates to developers the need to split the interface early and categorize the events into smaller models.

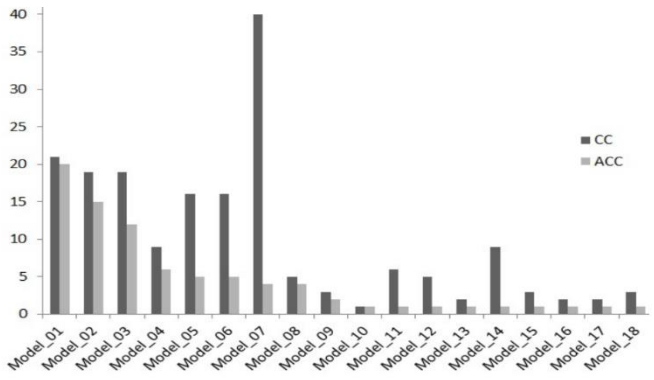

Fig. 9. Complexity of interface models of components sorted by ACC

Figure 9 depicts the CC and ACC of interface models of a number of components in one project. model 07 gives an example of a flower-shaped interface model with high CC and low ACC. By reviewing the contents of the model, we realized that the interface contains many events that should be categorized and split into smaller interface models. Notable are model 05 and model 06 which exhibit similar metrics. After reviewing the models, we found that they are isomorphic in structure (they model 2 physical sensors of the same type with different ids). An action was taken to combine the two models in one and parametrize the ids of the sensors.

We observed that Halstead T and E metrics are very controversial. We found that these metrics provide good estimates for models that are within the recommended size limit of 8000. For some models that exceed this limit the metrics are not very accurate. Empirical experiments are needed to adapt the formula for this type of models.

7. Metric Validation

The validation of a metric is done from two perspectives. The first perspective considers the metric and the views of the engineers and the second considers only the metric. To validate if a metric measures what it intends to, we performed an empirical validation. This validation is described in Section 7.1 which addresses the survey and the statistical framework used to gather and analyze the results. To validate a metric itself, we propose to validate a metric against a set of formal mathematical properties, see Section 7.2. These properties are presented by Elaine Weyuker [21] and she argues that an empirical validation is not sufficient to assess whether a metric is useful.

7.1 Empirical Validation

The view on what is good quality is different per engineer. Specifically, different outcomes have a different meaning for each engineer. For example, for one engineer an outcome of five for the action list size in the response list of ASD tables is fine and for another engineer it is too high. This difference can be caused by the differences in knowledge, design style, experience, and role within the organization, among others. A validation of the metrics can be performed by interviews and review meetings. These meetings take time and effort, since engineers need to reach a consensus, which is needed for a metric to support an engineer during development. Reaching this consensus needs to be done per metric.

Instead of reaching a consensus for each metric, we capture the views of the engineers once and use them to validate multiple metrics (at this moment or in the future). An empirical approach intends to do precisely that. Our approach is inspired by the approach from Jorge Cardoso [21] (Cardoso), which was inspired by M.V. Zelkowitz & D.R. Wallace [23] (Zelkowitz et al.) and D.E. Perry, A.A. Porter, & L.G. Votta [24] (Perry et al.). The approach is divided in six main activities, which are addressed in the following paragraphs. These activities are Research Context, Hypotheses, Study Design, Threats to Validity, Data Analysis and Presentation, and Results.

Research Context The goal of the study is to validate the complexity metrics of ASD models discussed above and start with building up a reference set for future validations.

The specific metrics under consideration are the CC, the ACC, the HC, and the MI. These metrics are presented in a previous article [19] and are reviewed by the engineers within ASML and peers from The Institute of Electrical and Electronics Engineers [19] (IEEE).

Hypotheses Before the study is set up and performed, it is important to know and to state what we intend to evaluate. The hypotheses are statements that represent formally what is under evaluation. We present two hypotheses, an abstract and a concrete hypothesis. The abstract hypothesis indicates in general terms the question we want to answer. The concrete hypothesis is derived from the abstract hypothesis and can be used to test if the hypothesis holds. Our hypotheses do not reference a specific metric, since we validate multiple metrics individually and for each metric a similar hypothesis is tested. Therefore, we use the wording “candidate metric”.

Abstract Hypothesis: The candidate metric gives an indication of the complexity of a model.

Concrete Hypothesis: There is a significant correlation between the candidate metric and the subject’s rating of the complexity of a model.

Study Design After the statement of the hypotheses, the study is set up and designed. The study design is a detailed plan for collecting the data, analyzing it, and testing the hypotheses. The design is as follows:

Variable Selection: Typically, there are two kinds of variables: independent and dependent. The independent variable is the cause of the effect. The dependent variable is dependent on Metrics for independent variables and changes when the independent variables change. In our study, the structure of the model is the independent variable. The dependent variable is the complexity of the model, which varies when the structure of the model changes.

Subject Selection: The subjects in the study are professional engineers from ASML. These engineers are part of different departments and have different roles as well as experience levels. All participants are regular ASD users. Specifically, most participants use ASD on a weekly basis. The population consists of the 22 participants of which there are:

- 9 developers, 14 designers, 3 architects, and 3 testers. [multiple roles possible, 1 unknown]
- 1 undergraduate, 9 graduates, and 8 post-graduates. [4 unknown]
- 5 daily users, 10 weekly users, 4 monthly users, and 2 irregular users. [1 unknown]
- 9 experienced users, 5 not inexperienced nor experienced users, 3 inexperienced users. [5 unknown]
- 10 more experienced, 5 at same level of experience, and 5 less inexperienced participants compared to their direct colleagues. [2 unknown]

Note that, to alleviate threats to validity we excluded any participant who uses ASD in the context other than constructing real production models, such as students or researchers.

Experiment Design: The models under consideration are ASD models randomly selected from a set of over a 1,000 production models based on the measured complexity. The production models were analyzed by computing the CC and ACC metrics. The results of the corpus analysis can be found in [20]. The CC and ACC metrics measure the independent variable of our study. Several intervals were constructed based on the thresholds presented in [19]. All analyzed models were categorized based on these intervals. From each interval, one model was randomly selected to make sure that all variations of measured complexity are under consideration. All selected models were analyzed on the other metrics to verify that there was variation in the outcomes. The total number of selected models is 30.

The dependent variable, the observed complexity, was measured by scores and labels. The models were rated by the participants by providing a score and a label. The scores need to be between 0 and 100, where 0 indicates “no complexity” and 100 “highly complex.” In addition to the scores, the participants gave one of the five labels explicitly indicating the level of perceived complexity, ranging from “no complexity” to “highly complex”.

Most participants rated the models individually in two sessions, one per location. The participants were aware of the setup of the survey and instructed to not share information with each other during the sessions. The participants had unlimited time to rate all the models. A handful of participants was not able to join the sessions and therefore rated the models at their own convenience.

Threats to Validity There are a number of factors that can influence the study and its results or that limit the ability for interpretation. These influences are called threats to validity and the relevant ones are presented in this paragraph

Construction Validity: All the measurements of the dependable variable are subjective and based on the perception of the participants. The participants in this study are familiar with ASD and therefore we think that their ratings reflect their views on complexity

The measurements of the independent variable can also be considered as constructively valid, since they measure the structural interaction between elements of the model, which is in line with complexity theory. Additionally, the metric was reviewed by a number of peers.

The method we used is partially subject to the mono-method bias, bias referring to measures and observations in only one way. The dependent variable is only measured by the views of the engineers. The participants rated the model in two ways, which partially mitigates the mono-method bias. To mitigate it, other sources of complexity aspects can be added in the future, such as number of modifications of the model.

Since the models were selected from the set of production models, participants could fall into the trap of evaluation apprehension. Participants could rate their own models with a lower score than they would do for other models with the same complexity. To mitigate this, participants and models are selected from different departments. For each model the majority of participants is not familiar with it.

Internal validity: Threats to internal validity compromise our confidence in something about the relationship between the dependent and independent variable. The effects of confounding variables, variables that influence the dependent variable but are not in the scope of the study, are typically considered as threats to internal validity. Example effects are the learning effect and the fatigue effect.

The learning and ordering effects are relevant for our study. The learning effect is the effect on the study where participants improve their results or performance because

they are used to the experiment. The ordering effect is the effect where the ratings of models later in the survey are biased, since they are compared against models earlier in the survey. To mitigate both effects, each participant received a random ordering of the models.

The number of models that was selected was based on the assumption that a model could be rated in one to one-and-a-half minutes. This would mean that all models could be rated in 45 minutes to one hour and therefore the effects of fatigue would not occur. During the study we observed that most participants needed more than one hour and some needed even three hours to complete the survey. The random ordering of the models partially mitigates the effects of fatigue, since all models have an equal probability to be subject to these effects. The focus of the survey was on complexity, but the participants were asked to also rate the models on other quality attributes, such as readability and cohesion. The quality attributes more to the right of the survey are more likely to be subject to the effects of fatigue and therefore findings for those attributes should be treated with care.

External Validity: Threats to external validity compromise our confidence in the applicability of the results. We identified two threats to external validity. Namely, the subject selection and the ecological threat. The subject selection limits the ability to generalize the results to other engineers within ASML. The participants were selected based on their familiarity with ASD. Therefore, we cannot generalize the findings for other engineers, who are abundant within ASML.

The other threat to external validity, ecological threat, limits the generalization of the findings to other domains. The selected models are taken from the set of production models and therefore only address the domain of ASML. The findings might apply to other domains, but this needs to be further investigated.

Data Analysis and Presentation The ratings of the participants can be analyzed with two approaches, quantitative and qualitative. Since the participants rated the models with a score of 0 to 100, we selected a quantitative analysis. The labels are used to perform a qualitative analysis of the ratings.

Analysis of the Scores: As mentioned earlier, our goal is to determine if a significant correlation exists between the outcomes of the candidate metric and the subject's rating of the complexity of a model. Since the ratings are distribution-free, the rs is used to determine the correlation. The rs is a non-parametric statistic used to show the relationship between two variables, which are expressed as ranks (the ordinal level of measurement). The coefficient is a measure of the ability of one variable to predict the value of the other. We use the rs to correlate the ratings of a participant to the outcomes of the candidate metrics. To use the rs, a null hypothesis is needed. We used the following null hypothesis:

H0: there is no correlation between the candidate metric and the subject's rating of the complexity of a model.

The probability that the null hypothesis would be erroneously rejected is controlled by the confidence level. We used the confidence level of 95% (indicated by $\alpha = 0.05$). We reject the null hypothesis if $rs > 0.375$. The constant is based on the degrees of freedom of our study, which is 28, and the α.

Participant	CC	ACC	HC	MI	MMI
1	0.709785	0.586756	0.317464	-0.61468	-0.59956
2	0.683487	0.685596	0.51182	-0.64497	-0.6235
3	0.61619	0.785929	0.287049	-0.77592	-0.8004
4	0.773239	0.681581	0.339711	-0.70323	-0.64316
5	0.704907	0.758081	0.4412045	-0.79245	-0.77449
6	-0.64067	-0.49883	-0.52618	0.556161	0.535579
7	-0.63431	-0.64769	-0.51898	0.68237	0.701341
8	0.653787	0.676293	0.43657	-0.75938	-0.74474
9	0.551494	0.62172	0.285049	-0.49374	-0.49956
10	0.685816	0.70492	0.319502	-0.8388	-0.82612
11	0.575714	0.708696	0.412034	-0.57426	-0.59766
12	0.726692	0.803884	0.454024	-0.77267	-0.7984
13	0.141378	0.234391	0.2307	-0.18398	-0.19689
14	0.637281	0.839309	0.422848	-0.72736	-0.77282
15	-0.69719	-0.56736	-0.58594	0.713864	0.696197
16	0.486303	0.485971	0.350697	-0.54873	-0.538
17	0.765341	0.792785	0.476519	-0.78389	-0.75807
Overall	**94.12%**	**94.12%**	**58.82%**	**94.12%**	**94.12%**

Table 5: rs between the participant's rating and the outcomes of the candidate metrics

(Note) Blue cells indicate rejection of the H0 (> 0.375). Candidate metrics are Cyclomatic Complexity [2, 3] (CC), Structural Complexity [3] (ACC), Halstead Complexity [4, 3] (HC), and Maintainability Index [5, 3] (MI).

Analysis of the Labels: The qualitative analysis of the labels is used to determine the thresholds for the candidate metric. Per model we compute the number of participants who rated it with a specific label. In other words, we count how often a specific label is given to a specific model.

If a label receives the absolute majority, the half or more, of the occurrences, then we consider the label as significant. If the number of occurrences is between the uniform share, total number of participants divided by the number of labels, and the absolute majority, then we consider that the label might be significant. Based on the significant

labels, we extract thresholds. These thresholds can be used to assess the complexity of the model and can give a meaning to the outcomes.

Results Table 5 presents the rs between the ratings of the participant and the outcome of the candidate metric. The rs is presented for each participant. The last row presents the percentage of significant correlations. The candidate metrics are the CC, ACC, HC, and MI. The null hypothesis is rejected if the rs is above 0.375. The cell is colored blue for the participants for which the H0 is rejected. Note that for some participants we observed a negative correlation. The reason for this is that their ratings were reversed. For them 100 meant no complexity and 0 highly complex. From the results we can observe that 94.12% of the participant ratings correlates significantly with our candidate metrics, except for the HC. Therefore, we can conclude that the CC, ACC, and MI indeed measure the complexity of a model. Note that for the MI metrics the correlation is a negative correlation.

Model	Occurrences of the labels				
	No complexity	Low complexity	Moderate complexity	High complexity	Overly complex
1	0	6	7	2	3
2	2	12	2	1	2
3	1	2	9	7	0
4	1	12	4	0	2
5	13	2	1	1	1
6	11	6	0	0	2
7	2	11	3	0	3
8	0	2	2	9	6
9	0	7	10	0	2
10	1	3	5	8	2
11	0	1	5	6	7
12	1	6	9	2	1
13	0	11	6	0	2
14	0	5	7	4	3
15	0	5	10	2	2
16	0	1	2	1	15
17	0	1	3	5	10
18	1	3	7	5	3
19	0	2	4	7	6
20	1	10	3	3	2
21	4	8	3	1	3
22	0	2	4	7	6
23	0	9	9	0	1
24	3	6	5	0	5
25	0	6	10	2	1
26	3	10	2	2	1
27	1	4	10	3	1
28	3	7	4	3	2
29	1	2	7	3	6
30	0	7	7	3	2

Table 6: Occurrences of a label for a model. Blue cells indicate significant labels and light blue might be significant labels.

Table 6 presents the number of occurrences of each label given by the participants for each model. The outcomes of the candidate metric are also presented. If a label is significant, absolute majority of the occurrences, the cell is colored blue. If a label might be significant, the number of occurrences is more than the uniform fraction, the cell is colored light blue. From the results we see that for each model at least one label might be significant or is significant. Based on these labels we can extract thresholds. For convenience we order the table based on the outcome of the candidate metric. The extraction of the thresholds is done by eyeballing the table. A statistical approach could be used instead, but we did not investigate possible approaches due to time limitations. Table 7 orders the models based on the outcome of the CC metric. Based on this table we could extract the following (example) thresholds:

- $0 \leq cc < 30$: Low complexity.
- $30 \leq cc < 60$: Moderate complexity.
- $60 \leq cc < \infty$: High complexity

Model	Occurrences of the labels					CC
	No complexity	Low complexity	Moderate complexity	High complexity	Overly complex	
5	13	2	1	1	1	2
6	11	6	0	0	2	8
13	0	11	6	0	2	22
26	3	10	2	2	1	23
4	1	12	4	0	2	26
23	0	9	9	0	1	28
30	0	7	7	3	2	29
7	2	11	3	0	3	30
10	1	3	5	8	2	30
20	1	10	3	3	2	30
15	0	5	10	2	2	34
14	0	5	7	4	3	35
21	4	8	3	1	3	35
28	3	7	4	3	2	36
27	1	4	10	3	1	38
1	0	6	7	2	3	40
12	1	6	9	2	1	40
9	0	7	10	0	2	45
2	2	12	2	1	2	47
11	0	1	5	6	7	53
18	1	3	7	5	3	53
25	0	6	10	2	1	56
24	3	6	5	0	5	57
29	1	2	7	3	6	68
8	0	2	2	9	6	72
3	1	2	9	7	0	87
19	0	2	4	7	6	95
17	0	1	3	5	10	108
22	0	2	4	7	6	173
16	0	1	2	1	15	334

Table 7: Occurrences of a label for a model ordered by the outcome of the CC metric. Blue cells indicate significant labels and light blue might-be-significant labels

7.2 Formal Validation

In addition to the empirical validation, we can also validate a metric against a set of formal mathematically properties. Weyuker proposed a set of nine mathematical properties that a complexity metric should have. Since the focus of our metrics is about complexity, we summarize the properties in the next paragraphs.

Outcome Differences The first property states that there should be programs for which the metric has a different outcome. If for all programs the outcome is the same, it is neither useful nor practical. The reverse is often also not useful, where all items have a different outcome. Essentially, the metric should not be too coarse nor should it be too fine. So, the second property strengthens the first by considering the same outcome only occurs for a limited number of programs. Formally, it states that for an outcome c, there are finitely many programs for which the metric maps them to c. In addition to the second property, the third property states that there are distinct items with the same outcome. The first three properties together make sure a metric is neither too coarse nor too fine.

Implementation Abstraction The first three properties do not consider the fact that programs have semantics and syntax. Property four reflects the fact that we are dealing with syntactic metrics. The intuition behind property four is that even though two programs compute the same function, the details of the implementation determine the complexity. Property four states that there exist behaviorally / semantically the same programs with different outcomes.

Items and Their Components The complexity of a program originates from its implementations and the interactions between the components the implementation is composed of. Property five addresses this aspect, by stating that the complexity of two programs should be less than or equal to the complexity of the composite of the two items. We do not expect the complexity to decrease by composing programs.

On a related line of thought, we would expect that the complexity can increase by composing programs. Specifically, we expect that if there is an interaction between the implementations of two composed programs, that the complexity is different. This is what property six states; there is a program for which the complexity of the composite with one program (Q) is different than for the composite with another program (P), where the programs P and Q have the same complexity.

Property nine is a slight strengthening on properties five and six, in some cases. Property nine indicates that for some programs it holds that the complexity of the composite is larger than the sum of the complexities. The intuition behind property nine is that the interactions between the items contribute to the complexity of the composite.

Implementation Interactions We discussed that the complexity is dependent on the interactions of the implementation. Property seven takes this a bit further and states that the reordering of an implementation leads to a different outcome. A different internal ordering leads to different interactions and therefore a different outcome.

Measure the Syntax The metric is defined on the syntax and not on the semantics. Therefore, property eight states that if we rename a program or parts of its implementation then the outcome remains the same. A concrete example of a metric that satisfies this property is a metric that is robust against the difference in names of variables or functions.

8. CONCLUSIONS AND FUTURE WORK

As industry is rapidly migrating towards model-based development, it is becoming urgent to establish means to measure the quality of models since they form the main software artifact in the modeling paradigm. In this article we proposed a number of metrics for ASD models which are state machines specified in a tabular format. As a result of applying our new metrics, we could measure the quality of these models using the new metrics. This introduces a basis to introduce new metrics for other type of models.

An apparent limitation of our work is that we only considered the structural complexity of models. The added complexity of introducing guards in the specification is not considered. Guards can have a similar complexity effect as introducing states. Finally, the results of this work reveal the importance and need for metrics at the model level. Based on the metric feedback, and subsequent review of the flagged models, interesting patterns and opportunities for model improvement were identified. Moreover, the results reveal that more work is needed to extend the set of metrics making them also less sensitive or biased for certain patterns and aspects.

References

[1] ASML homepage. http://www.asml.com. (Accessed 2024).

[2] F. Badeau and A. Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL, p 334–354. Springer Berlin Heidelberg, 2005.

[3] J.L. Boulanger, F.-X. Fornari, J.-L. Camus, and B. Dion. SCADE: Language and Applications. Wiley-IEEE Press, 1st edition, 2015.

[4] CodeSonar homepage. http://www.grammatech.com. (Accessed 2024).

[5] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. Computer, 27(8):44–49, Aug. 1994.

[6] Formal Systems (Europe) Ltd. FDR2 model checker, 2011. http://www.fsel.com/

[7] J.S. Fitzgerald, P. G. Larsen, and S. Sahara. Vdmtools: advances in support for formal modeling in VDM. SIGPLAN Notices, 43(2):3–11, 2008.

[8] L. Guo, A.S. Vincentelli, and A. Pinto. A complexity metric for concurrent finite state machine based embedded software. In 2013 8th IEEE International SIES, p. 189–195, 2013.

[9] M. Fowler and K. Beck. Refactoring: Improving the Design of Existing Code. Component software series. Addison-Wesley, 1999.

[10] M.H. Halstead. Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA, 1977.

[11] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[12] J. Jurjens and S. Wagner. ¨ Component-Based Development of Dependable Systems with UML, pages 320–344. Springer Berlin Heidelberg, 2005.

[13] T.J. McCabe. A complexity measure. IEEE Trans. Softw. Eng., 2(4):308–320, July 1976.

[14] H. D. Mills. Stepwise refinement and verification in box-structured systems. Computer, 21(6):23–36, June 1988.

[15] A. Osaiweran, M. Schuts, J. Hooman, J.F. Groote, and B. van Rijnsoever. Evaluating the effect of a lightweight formal technique in industry. Int. Jour. on STTT, Springer, 18(1):93–108, 2016.

[16] A. Osaiweran, M. Schuts, J. Hooman, and J. Wesselius. Incorporating formal techniques into industrial practice: An experience report. ENTCS. 295:49–63, May 2013.

[17] Tiobe homepage. http://www.tiobe.com. (Accessed 2024).

[18] Verum homepage. http://www.asd.verum.com. (Accessed 2024). [19] Verifysoft homepage. http://www.verifysoft.com. (Accessed 2024).

[19] A. Osaiweran, J. Marincic, and J. F. Groote, “Assessing the quality of tabular state machines through metrics,” in IEEE International Conference on Software Quality, Reliability and Security. Prague, Czech Republic: IEEE Computer Society, 2017, pp. 426–433.

[20] C. Lambrechts, “Metrics for Control Models in a Model-Driven Engineering Environment”. PDEng thesis, Technische Universiteit Eindhoven, sept 2017.

[21] E. J. Weyuker, “Evaluating software complexity measures,” IEEE Trans. Softw. Eng., vol. 14, no. 9, pp. 1357–1365, Sep. 1988.

[22] J. Cardoso, “Process control-flow complexity metric: An empirical validation,” in Proceedings of the IEEE International Conference on Services Computing, ser. SCC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp.

[23] M. V. Zelkowitz and D. R. Wallace, “Experimental models for validating technology,” Computer, vol. 31, no. 5, pp. 23–31, May 1998.

[24] D. E. Perry, A. A. Porter, and L. G. Votta, “Empirical studies of software engineering: A roadmap,” in Proceedings of the Conference on The Future of Software Engineering, ser. ICSE ’00. New York, NY, USA: ACM, 2000, pp. 345–355.

[25] Masmali, O., Badreddin, O. (2021). Theoretically Validated Complexity Metrics for UML State Machine Models. In: Arai, K., Kapoor, S., Bhatia, R. (eds) Proceedings of the Future Technologies Conference (FTC) 2020, Volume 3. FTC 2020. Advances in Intelligent Systems and Computing, vol 1290. Springer, Cham. https://doi.org/10.1007/978-3-030-63092-8_28

[26] Lidia López, Xavier Burgués, Silverio Martínez-Fernández, Anna Maria Vollmer, Woubshet Behutiye, Pertti Karhapää, Xavier Franch, Pilar Rodríguez, Markku Oivo, Quality measurement in agile and rapid software development: A systematic mapping, Journal of Systems and Software, Volume 186, 2022, 111187, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2021.111187