2017 IEEE International Conference on Software Quality, Reliability and Security

Assessing the Quality of Tabular State Machines through Metrics

Ammar Osaiweran
ASML Netherlands B.V.,
Veldhoven, The Netherlands
ammar.osaiweran@asml.com

Jelena Marincic
ASML Netherlands B.V.,
Veldhoven, The Netherlands
jelena.marincic@asml.com

Jan Friso Groote
Eindhoven University of Technology,
Eindhoven, The Netherlands
j.f.groote@tue.nl

***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 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 industrial projects.**

I. 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 analyzed 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. This article is structured as follows. Section II discusses related work on metrics of state machines. Section III introduces ASD to the extent needed for this article. In Section IV a number of well-known software metrics are detailed with the application to ASD models. Section V introduces recommended limits of metrics for good quality models. Section VI details the data collection process of metrics from models and discusses observations during the data analysis. In Section VII we conclude our paper highlighting the limitations of our metrics and future work in this regard.

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.52

II. 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.

III. 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 behavior of a component is specified using an interface model which excludes any internal behavior 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 behavior. It is also formally checked whether the behavior 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).

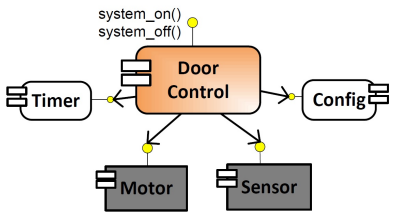

Fig. 1. Example controller system of automatic door

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 timesout 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 III-A and the design model in Section III-B.

A. 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

	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

the *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

B. 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.

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 $\gg$ operator, to be retrieved later when needed via $\ll$ 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.

IV. 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.

A. 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:

	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

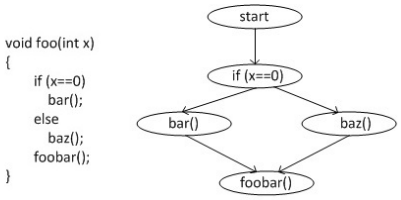

Fig. 5. Code and its graph representation

$$CC = E - N + 1,$$

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: $5 - 5 + 1 = 1$.

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.

To illustrate how *CC* can be collected for ASD models, consider the specification depicted in Figure 6. The specification consists of 2 states namely

	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

state X and state Y. In state X, the machine accepts events a_1, a_2 and a_3 via the *IF* interface and then moves to state Y. The machine stays in state Y forever accepting a_4 and a_5 events.

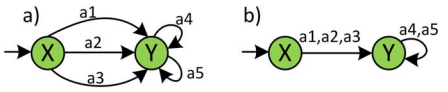

Fig. 7. a) Graphical representation with independent edges for events. b) Graph with unique edges with set of actions

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.

B. 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 a_1, a_2 or a_3 events (one transition labeled by a set of events). In state Y only a_4 or a_5 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:

$$ACC = E_U - N + 1,$$

where E_U 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:

$$E_U = 2, N = 2,$$
$$ACC = 2 - 2 + 1 = 1.$$

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.

C. 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:

- n_1: the number of unique operators,
- N_1: the number of occurrences of operators,
- n_2: the number of unique operands,
- N_2: the number of occurrences of operands,
- $n = n_1 + n_2$: the model vocabulary,
- $N = N_1 + N_2$ 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 callbacks) and actions (all responses including return values and callbacks),
- operators on state variables such as not, and, or, $>$, $<$, $==$, $\leq$, $\geq$, +, $-$, and *otherwise* (a keyword denoting the else part of a guard),
- operators on data variables such as $\gg$, $\ll$, $><$ (value of variable is stored and retrieved), and \$ (literal value).

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

- Volume: $V = N * log_2 n$,
- Difficulty: $D = (n_1/2) * (N_2/n_2)$,
- 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:

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

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 I 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 I
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.

V. 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 II
OPTIMAL VALUES OF METRICS FOR ASD MODELS

Table II 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.

VI. 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 III. 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 III
SUMMARY OF STATISTICAL DATA OF DEVELOPED MODELS

Table III 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	Percentage
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 IV
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 IV.

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.

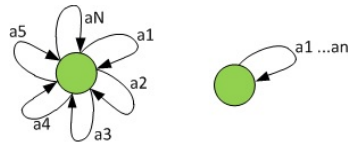

Fig. 8. Representing a stateless machine as a flower-shape (*CC*) or a mouse ear (*ACC*)

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 exhibit a relatively very low *ACC* metric. Correlating *CC* and *ACC* can help developers detecting interfaces

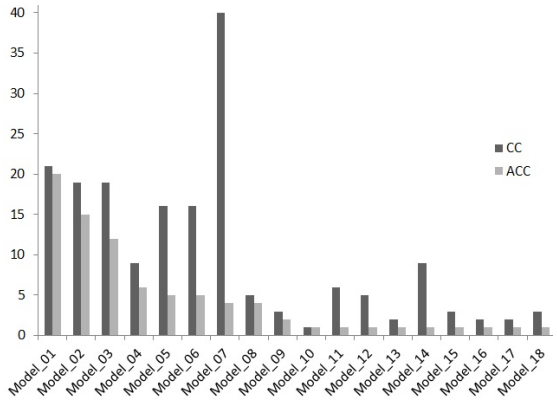

Fig. 9. Complexity of interface models of components sorted by *ACC*

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.

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.

VII. 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.

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 2017).

[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 2017).

[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. Jürjens 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 2017).

[18] Verum homepage. http://www.asd.verum.com. (Accessed 2017).

[19] Verifysoft homepage. http://www.verifysoft.com. (Accessed 2017).