Assessing the quality of tabular state machines through metrics

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Assessing the quality of tabular state machines through metrics

Ammar Osaiweran, Jelena Marincic, Jan Friso Groote

17/01

ISSN 0926-4515

editors: prof.dr. P.M.E. De Bra
prof.dr.ir. J.J. van Wijk

Reports are available at:
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Author&level=1 and
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Year&Level=1

Computer Science Reports 17-01
Eindhoven, February 2017

Assessing the quality of tabular state machines through metrics

Ammar Osaiweran[1], Jelena Marincic[1], Jan Friso Groote[2]

[1]ASML Netherlands B.V., Veldhoven, The Netherlands
[2]Eindhoven University of Technology, Eindhoven, The Netherlands
ammar.osaiweran@asml.com, *jelena.marincic@asml.com*, *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 when collecting these metrics. Furthermore, we recommend some quality limits for each metric and validate them on models developed in a number of industrial projects.

1 Introduction

The use of model-based techniques in software development processes has been promoted for many years [24, 3, 4, 5, 1, 12, 6]. The aim is to use the models as a main software artifact in the development process, not only for visualization and communication among developers but also as an important means of specification, formal verification, code generation, testing and validation.

The premise is that, by modeling, engineers will focus more on the core software functionality rather than the implementation details. As a crucial part of the modeling paradigm, the code is often automatically generated, implementing the specification of the source model. This automatic construction of source code gives real-world value to the behavior specified in the model. Usually, the transformation to code is hidden from modelers; it is just one more command to execute or a button to click before compilation. Furthermore, for some modeling frameworks, behavioral correctness of models is established by automatic formal verification of which related formal specification is also hidden from end users. Visible to users is only the verification results or counterexamples guiding users when certain properties are violated.

The shift from traditional coding towards the model-based development paradigm is becoming very popular and attractive in industry. The reason is that it results in a notable increase of quality and reduction of time to market. Implementation details that support the execution of the core functionality is taken care by the code generator, reducing the time and overhead for error-prone manual implementation, facilitating automatic verification, and increasing overall productivity [23].

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 [27], CodeSonar [9], SourceMonitor [26] and VerifySoft [28]. Code metrics are useful means to detect decays in architectures and code smells [13] 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 and 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. The behavior of these machines is described using a lightweight formal modeling tool called ASD:Suite. The tool allows specification of state machines in a tabular format. These specifications can be formally verified and corresponding source code can be generated from these verified models [29].

Using ASD:Suite, we can create models but how can we ensure their quality. Because there is 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? Will we find complex and big models in the software archive? 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. We discuss a number of observations raised when analyzing metrics of various models. Based on our empirical results, we propose a number of practical thresholds for various metrics. Note that our work is applied to models developed in real industrial projects and real products that are shipped to the market, and not to simple case studies or prototypes.

This article is structured as follows. Section 2 discusses related work on model-based development and 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 conclude our paper highlighting the limitations and future work in this regard.

2 Related work

In a previous research at Philips Healthcare [25], 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. Furthermore, there was no automatic means to calculate the metrics at the level of models.

Most recently, a number of metrics such as cyclomatic complexity (CC) [21], Halstead complexity [17] and Zage complexity [31, 30] were applied to SCADE models. The purpose is to establish whether metrics for traditional source code can be used to assess complexity of SCADE model and to detect unavoidable complexity.

To estimate the reliability of UML state machines, and to identify failure-prone components, a group of authors [20] 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 [15] decision diagrams as intermediate artefacts were used to calculate the number of tests for the code of concurrent state machines.

For automatically generated state-machines that contains a large number of states, and that have abstraction levels flattened, the work of [16] proposes a complexity metric to assist in generating hierarchical state-machines from a flat state-machine. A technique for search-based clustering of related states to identify potential superstates is used and then the *CC* of each cluster is evaluated for a proper choice of super-states.

3 Analytical Software Design

This section provides a short introduction of the ASD approach and its toolset, the ASD:Suite [7, 19]. ASD is an approach used for building formally verified, component-based systems through the application of formal methods into industrial practice. ASD combines the Box Structure Development Method [22] and the Communicating Sequential Processes (CSP) formalism [18], and uses Failures-divergence refinement FDR2 [11] as a model checker tool for formal verification.

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*. These models are specified following the Sequence-Based Specification technique, to force consistent and complete specifications [8].

The external behavior (or contract) 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.

In ASD we distinguish between two types of components: ASD components and foreign components. An ASD component includes an implemented interface model, a design model, and optional server interface models. A foreign component has only an interface model of which implementation is constructed manually.

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*. The model checker tool exhaustively searches for illegal interactions, deadlocks or livelocks in the specification. It is also formally checked whether the behaviour of the design model obeys its implemented interface model. Verification starts automatically with the click of a button. In case an error is detected in the

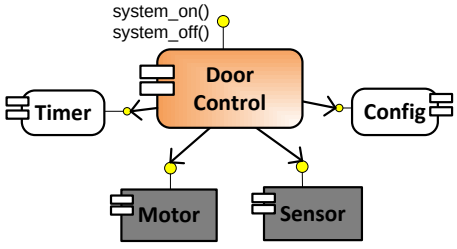

Figure 1: example controller system of automatic door

models, the modeler receives a counterexample visualized in a sequence diagram, nicely traceable to the original specification of the model. Besides formal verification, the ASD:Suite allows code generation from design and interface models to a number of languages (C, C++, C#, Java).

In ASD, communication between client and server components is asymmetric, using synchronous calls and asynchronous callbacks. A client issues synchronous calls to server components, whereas a server sends callbacks to its clients. Callbacks are stored in a First-In-First-out (FIFO) callback queue. These callbacks are non-blocking and can be received by a component at any time.

Note that in ASD:Suite a designer can configure an ASD component to be multi-threaded or single-threaded. Using the multi-threaded option any ASD queue will run in its own thread causing potential thread-switching and interleaving of actions. In our industrial context we always use the single-threaded option which means that actions are executed until completion without any interleaving with other actions of the same or other ASD components.

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 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 formal contract of the component by means of the allowed sequence of calls and callbacks, exchanged with clients. Any internal behaviour 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. 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 *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. Or conversely, it is an abstract event that can or must occur which therefore acts as an obligation for any component that implements that interface.

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
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		

Figure 2: interface model of door controller

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
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		

Figure 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 design model is used to specify how the provided interface model is implemented by mapping it to all required (or used) interface models. This means that the design model may include 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

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
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		

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

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. Careful attention and thorough review of the data is needed because checking actual content of the data is excluded from formal verification in ASD. The rest of the specification is self-explanatory.

An example of processing a callback that is stored in the ASD queue 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 McCabe and Halstead complexity metrics [21, 17]. In this section we introduce these metrics and discuss how we adapt them to measure ASD design and interface models.

We start by introducing McCabe cyclomatic complexity metric (CC) and its application to measure complexity of ASD models. Then, we introduce our tailored version of the CC metric and also 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. Finally, we present metrics related to formal verification generated by the model checker of ASD:Suite.

4.1 Cyclomatic complexity of ASD models

The cyclomatic complexity (CC) metric provides a quantitative measure on the number of linearly independent paths in a program source code, represented by a control flow directed graph [21]. 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.

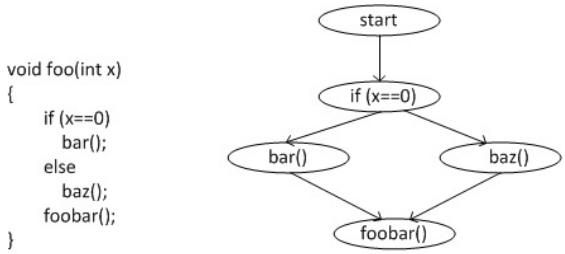

Figure 5: code and its graph representation

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:

$$\mathrm{CC} = \mathrm{E} - \mathrm{N} + 1,$$

where E denotes the number of edges in the graph and N is the total number of nodes. Clearly 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 ASD foreign 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 state X and state Y. In state X, the

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
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		

Figure 6: An ASD interface model with 2 states and 5 transitions

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.

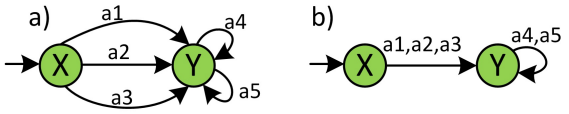

Figure 7: a) Graphical representation with independent edges for events. b) Graphical representation with unique edges with set of actions

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

$$\mathrm{E} = 5, \mathrm{N} = 2,$$
$$\mathrm{CC} = 5 - 2 + 1 = 4$$

Application to the Door models
The *CC* of the *Door* interface model depicted in Figure 2 is 1, while *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 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:

$$\mathrm{ACC} = \mathrm{E_U} - \mathrm{N} + 1,$$

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:

$$\mathrm{E_U} = 2, \mathrm{N} = 2,$$
$$\mathrm{ACC} = 2 - 2 + 1 = 1.$$

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 1.

4.3 Halstead metrics, LoC and maintainability index

Using Halstead approach, metrics are collected based on counting operators and operands of source code [17]. We introduce these metrics and discuss how we tailored them to ASD models. Furthermore, we show how the lines of code metric is collected for ASD models. Another metric called the maintainability index can be derived based on Halstead metrics, the lines of code and CC metrics. We show how this metric is calculated for ASD models.

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 total number of occurrences of operators,
- n_2: the number of unique operands,
- N_2: the total number of occurrences of operands,
- $n = n_1 + n_2$ which indicates the model vocabulary,
- $N = N_1 + N_2$ which denotes 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.

Furthermore, 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*, $>$, $<$, $==$, $\leqslant$, $\geqslant$, $+$, $-$, and *otherwise* (a keyword denotes the else part of a guard),
- operators on data variables such as $\gg$, $\ll$, $><$ (value of variable is stored and retrieved), and \$ (literal value a programming language allows).

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 of source code is calculated based on volume, LoC and *CC* of source code [10]. It indicates whether it is worth to keep maintaining, modifying and extending a program or to immediately consider refactoring or redesigning it. *MI* should be above 85 or not less than 65 in the worst case. The Maintainability Index is defined as follows:

$$MI = 171 - 5.2 * \ln(V) - 0.23 * CC - 16.2 * \ln(\text{LOC})$$

Microsoft incorporated the metrics in Microsoft Studio environment with a slight modification to the above formula:

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

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 metrics of the three ASD models of the *Door* system. 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.

Model	Volume	Bugs	Difficulty	Time (s)	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

4.4 Metrics for formal verification overhead

ASD uses model checking for formal verification of interface models and design models. The model checking tool produces statistical information about the state space that captures all possible execution scenarios of a model (or a group of communicating models).

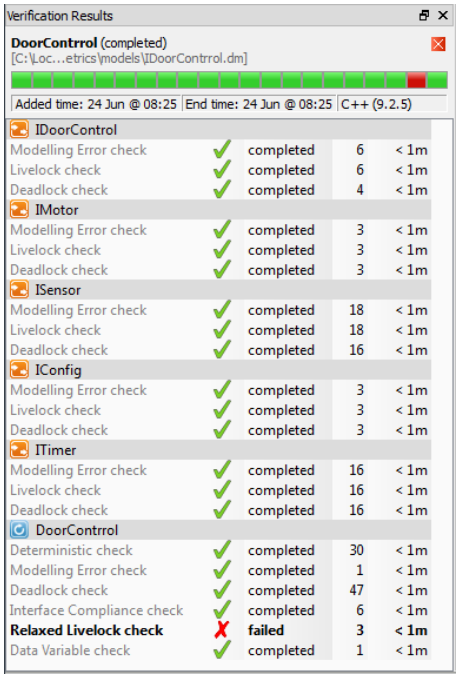

Figure 8: List of models and verification metrics (states and verification time).

Figure 8 depicts a screenshot of the results of the formal verification of ASD:Suite. It includes the design model of the Door controller and its used interface models. A green color indicates success of the formal check while red indicates a failing result.

As can be seen from the figure, the number of generated states of the design model for the deadlock check is 47 and the time required for all listed checks to complete is less than a minute. These metrics can also be obtained from a file generated by the ASD:Suite when the verification check is accomplished.

The deadlock check for the door design model is marked by a green tick sign indicating that the design model is deadlock-free for all possible execution paths.

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

Metric	Type of model					
	Interface Model			Design Model		
	Low	Moderate	High	Low	Moderate	High
CC	$\leqslant 30$	$\leqslant 50$	> 50	$\leqslant 30$	$\leqslant 50$	> 50
ACC	$\leqslant 20$	$\leqslant 40$	> 40	$\leqslant 20$	$\leqslant 40$	> 40
Volume	$\leqslant 8000$	$\leqslant 14000$	> 14000	$\leqslant 8000$	$\leqslant 14000$	> 14000
LoC	$\leqslant 200$	$\leqslant 400$	> 400	$\leqslant 500$	$\leqslant 800$	> 800
MI	$\leqslant 10$	$\leqslant 20$	> 20	$\leqslant 10$	$\leqslant 20$	> 20
VT	$\leqslant 1$ min	$\leqslant 5$ min	> 5 min	$\leqslant 1$ min	$\leqslant 5$ min	> 5 min

Table 2: Optimal values of metrics for ASD models

built for a large photolithography system, developed by ASML [2]. The limits were established after iterative review meetings and alignments with various engineers who owned and developed the models.

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

Note that 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 limit of *CC* for models is 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, we were reasonably comfortable reviewing models that exhibit an *ACC* of less than 20. For the size metric, we used the limit suggested by VerifySoft [28] 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. We want to avoid that a developer fixes an error in the specification and waits for a long time before the model checker succeeds or detects another error (and the behaviour repeats itself causing undesired long waits reducing the productivity of the designer). More important is that this limit is set to also prevent designers from making overly complex specification because they are safe with verification of model-checking.

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. Human creativity is still needed to judge whether a design is conceptually acceptable while metrics can help detecting bad smells and decays in the architecture very early.

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	Percentage
CC	$\leqslant 30$	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%
Volume	$< 8K$	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 although controlled empirical validation of the metrics are planned for future. 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 quality of their models to address any issue early during the modeling phase and before final delivery of the models.

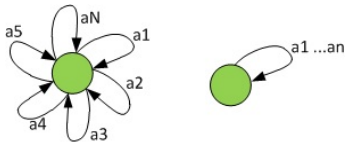

Figure 9: Representing a stateless machine as a flower-shape (CC) or a mouse ear (ACC)

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 on complexity. Consider Figure 9. 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 CC perspective the state machines is considered to be complex since it exceeded the complexity limit we set before as a guideline.

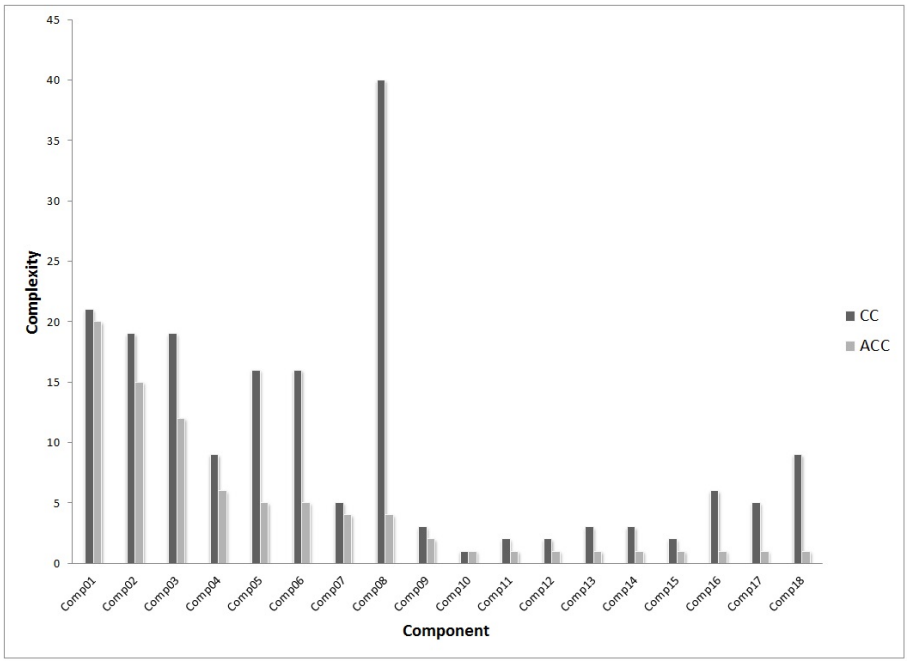

Figure 10: Complexity of interface models of components sorted by ACC

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 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 10 depicts the CC and ACC of interface models of a number of components in one project. $Comp08$ in the figure 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 $Comp05$ and $Comp06$ which exhibit similar metrics. After reviewing the models we found that they are exact copies (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 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 to measure the quality of 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. In fact, guards can have a similar effect in complexity as introducing states. For some developers, specifications with guards are relatively more complex to understand than specifications without guards. Future empirical evaluation is needed to validate this observation.

The metrics and the limits proposed in this article are constructed based on consensus and alignment with the majority of ASD designers through a number of meetings and interviews. The designers applied the metrics and the limits to their own developed models. As a future work we are planning to validate the metrics and the limits by executing controlled empirical experiments with a set of models selected from different projects.

For further validation we want to answer a number of questions like: is it always the case that any model big in size is complex (and vice versa)? Which metric contributes more to the number of bugs in the field? Size or complexity? How is McCabe's CC metric correlated to Halstead's difficulty metric? Shall we pay more attention to one of them or both? How can we re-calibrate the expected number of bugs of Halstead given that models are formally verified? Another interesting direction is to correlate these metrics to software quality attributes such as extensibility, scalability, testability and verifiability.

Another future direction is to detect similarities in the models caused by duplicating guards or responses events in the actions list. As highlighted previously in the paper, we accidentally detected clones between models by observing the plots of complexity. In the future we are investigating other systematic means to detect clones between models (part of a model is included in another model). Furthermore, modularity metrics will be introduced to indicate the degree of coupling and cohesion among the models.

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.

Acknowledgment
We would like to thank Sven Weber for his constructive and valuable comments to the article.

References

[1] J.R. Abrial. *The B tool (Abstract)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[2] ASML homepage. `http://www.asml.com`. (Accessed 2017).

[3] 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, Berlin, Heidelberg, 2005.

[4] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. *Météor: A Successful Application of B in a Large Project*, p 369–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[5] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of asms in software design. In *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verit à, Switzerland, March 19-24, 2000, Proceedings*, p 361–366, 2000.

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

[7] G.H. Broadfoot and P.J. Broadfoot. Academia and industry meet: some experiences of formal methods in practice. In *Tenth Asia-Pacific Software Engineering Conference, 2003.*, p 49–58, Dec 2003.

[8] J.M. Carter and J.H. Poore. Sequence-based specification of feedback control systems in simulink®. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 332–345, Riverton, NJ, USA, 2007. IBM Corp.

[9] CodeSonar homepage. `http://www.grammatech.com`. (Accessed 2017).

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

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

[12] 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.

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

[14] J.F. Groote and M. R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[15] L. Guo, A.S. Vincentelli, and A. Pinto. A complexity metric for concurrent finite state machine based embedded software. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 189–195, June 2013.

[16] M. Hall, P. McMinn, and N. Walkinshaw. Superstate identification for state machines using search-based clustering. In *Genetic and Evolutionary Computation Conference, Proceedings, Portland, Oregon, USA, July 7-11, 2010*, pages 1381–1388, 2010.

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

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

[19] P.J. Hopcroft and G.H. Broadfoot. Combining the box structure development method and csp for software development. *Electron. Notes Theor. Comput. Sci.*, 128(6):127–144, May 2005.

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

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

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

[23] A. Osaiweran. Formal development of control software in the medical systems domain. Eindhoven : Technische Universiteit Eindhoven, 2012.

[24] A. Osaiweran, M. Schuts, J. Hooman, J.F. Groote, and B. van Rijnsoever. Evaluating the effect of a lightweight formal technique in industry. *International Journal on Software Tools for Technology Transfer*, 18(1):93–108, 2016.

[25] A. Osaiweran, M. Schuts, J. Hooman, and J. Wesselius. Incorporating formal techniques into industrial practice: An experience report. *Electron. Notes Theor. Comput. Sci.*, 295:49–63, May 2013.

[26] SourceMonitor homepage. `http://www.campwoodsw.com/sourcemonitor.html`. (Accessed 2017).

[27] Tiobe homepage. `http://www.tiobe.com`. (Accessed 2017).

[28] VerifySoft homepage. `http://www.asd.verum.com`. (Accessed 2017).

[29] Verum homepage. `http://www.verifysoft.com`. (Accessed 2017).

[30] W.M. Zage and D.M. Zage. Evaluating design metrics on large-scale software. *IEEE Softw.*, 10(4):75–81, July 1993.

[31] W.M. Zage, D.M. Zage, M. Bhargava, and D.J. Gaumer. Design and code metrics through a diana-based tool. In *Proceedings of the 11th Ada-Europe International Conference on Ada: Moving Towards 2000*, pages 60–71, London, UK, UK, 1992. Springer-Verlag.

Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2012):

12/01	S. Cranen	Model checking the FlexRay startup phase
12/02	U. Khadim and P.J.L. Cuijpers	Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP
12/03	M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher	Revised budget allocations for fixed-priority-scheduled periodic resources
12/04	Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever	Experience Report on Designing and Developing Control Components using Formal Methods
12/05	Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse	A cure for stuttering parity games
12/06	A.P. van der Meer	CIF MSOS type system
12/07	Dirk Fahland and Robert Prüfer	Data and Abstraction for Scenario-Based Modeling with Petri Nets
12/08	Luc Engelen and Anton Wijs	Checking Property Preservation of Refining Transformations for Model-Driven Development
12/09	M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte	Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version –
12/10	Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien	Efficient reprogramming of sensor networks using incremental updates and data compression
12/11	John Businge, Alexander Serebrenik and Mark van den Brand	Survival of Eclipse Third-party Plug-ins
12/12	Jeroen J.A. Keiren and Martijn D. Klabbers	Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2
12/13	Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever	Evaluating the Effect of Formal Techniques in Industry
12/14	Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman	Incorporating Formal Techniques into Industrial Practice
13/01	S. Cranen, M.W. Gazda, J.W. Wesselink and T.A.C. Willemse	Abstraction in Parameterised Boolean Equation Systems
13/02	Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse	Decomposability in Formal Conformance Testing
13/03	D. Bera, K.M. van Hee and N. Sidorova	Discrete Timed Petri nets
13/04	A. Kota Gopalakrishna, T. Ozcelebi, A. Liotta and J.J. Lukkien	Relevance as a Metric for Evaluating Machine Learning Algorithms
13/05	T. Ozcelebi, A. Weffers-Albu and J.J. Lukkien	Proceedings of the 2012 Workshop on Ambient Intelligence Infrastructures (WAmIi)
13/06	Lotfi ben Othmane, Pelin Angin, Harold Weffers and Bharat Bhargava	Extending the Agile Development Process to Develop Acceptably Secure Software
13/07	R.H. Mak	Resource-aware Life Cycle Models for Service-oriented Applications managed by a Component Framework
13/08	Mark van den Brand and Jan Friso Groote	Software Engineering: Redundancy is Key
13/09	P.J.L. Cuijpers	Prefix Orders as a General Model of Dynamics

14/01	Jan Friso Groote, Remco van der Hofstad and Matthias Raffelsieper	On the Random Structure of Behavioural Transition Systems
14/02	Maurice H. ter Beek and Erik P. de Vink	Using mCRL2 for the analysis of software product lines
14/03	Frank Peeters, Ion Barosan, Tao Yue and Alexander Serebrenik	A Modeling Environment Supporting the Co-evolution of User Requirements and Design
14/04	Jan Friso Groote and Hans Zantema	A probabilistic analysis of the Game of the Goose
14/05	Hrishikesh Salunkhe, Orlando Moreira and Kees van Berkel	Buffer Allocation for Real-Time Streaming on a Multi-Processor without Back-Pressure
14/06	D. Bera, K.M. van Hee and H. Nijmeijer	Relationship between Simulink and Petri nets
14/07	Reinder J. Bril and Jinkyu Lee	CRTS 2014 - Proceedings of the 7th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems
14/08	Fatih Turkmen, Jerry den Hartog, Silvio Ranise and Nicola Zannone	Analysis of XACML Policies with SMT
14/09	Ana-Maria Șutîi, Tom Verhoeff and M.G.J. van den Brand	Ontologies in domain specific languages – A systematic literature review
14/10	M. Stolikj, T.M.M. Meyfroyt, P.J.L. Cuijpers and J.J. Lukkien	Improving the Performance of Trickle-Based Data Dissemination in Low-Power Networks
15/01	Önder Babur, Tom Verhoeff and Mark van den Brand	Multiphysics and Multiscale Software Frameworks: An Annotated Bibliography
15/02	Various	Proceedings of the First International Workshop on Investigating Dataflow In Embedded computing Architectures (IDEA 2015)
15/03	Hrishikesh Salunkhe, Alok Lele, Orlando Moreira and Kees van Berkel	Buffer Allocation for Realtime Streaming Applications Running on a Multi-processor without Back-pressure
15/04	J.G.M. Mengerink, R.R.H. Schiffelers, A. Serebrenik, M.G.J. van den Brand	Evolution Specification Evaluation in Industrial MDSE Ecosystems
15/05	Sarmen Keshishzadeh and Jan Friso Groote	Exact Real Arithmetic with Pertubation Analysis and Proof of Correctness
15/06	Jan Friso Groote and Anton Wijs	An $O(m \log n)$ Algorithm for Stuttering Equivalence and Branching Bisimulation
17/01	Ammar Osaiweran, Jelena Marincic Jan Friso Groote	Assessing the quality of tabular state machines through metrics