*Моделирование и анализ информационных систем.* Т. 24, №6 (2017), с. 691–703
*Modeling and Analysis of Information Systems.* Vol. 24, No 6 (2017), pp. 691–703

**DOI:** 10.18255/1818-1015-2017-6-691-703

UDC 519.686.2

# Towards Measuring the Abstractness of State Machines based on Mutation Testing

**Thomas Baar**

*Received October 30, 2017*

**Abstract.** The notation of state machines is widely adopted as a formalism to describe the behaviour of systems. Usually, multiple state machine models can be developed for the very same software system. Some of these models might turn out to be equivalent, but, in many cases, different state machines describing the same system also differ in their level of abstraction.

In this paper, we present an approach to actually *measure the abstractness level* of state machines w.r.t. a given implemented software system. A state machine is considered to be less abstract when it is conceptionally closer to the implemented system. In our approach, this distance between state machine and implementation is measured by applying coverage criteria known from software mutation testing.

Abstractness of state machines can be considered as a new metric. As for other metrics as well, a known value for the abstractness of a given state machine allows to assess its quality in terms of a simple number. In model-based software development projects, the abstract metric can help to prevent model degradation since it can actually measure the semantic distance from the behavioural specification of a system in form of a state machine to the current implementation of the system.

In contrast to other metrics for state machines, the abstractness cannot be statically computed based on the state machine's structure, but requires to execute both state machine and corresponding system implementation.

The article is published in the author's wording.

**Keywords:** model-based software development, metric, state machine, mutation testing

**For citation:** Thomas Baar, "Towards Measuring the Abstractness of State Machines based on Mutation Testing", *Modeling and Analysis of Information Systems*, **24**:6 (2017), 691–703.

**On the author:**
Thomas Baar, orcid.org/0000-0002-8443-1558, PhD,
University of Applied Sciences (Hochschule für Technik und Wirtschaft (HTW) Berlin)
Wilhelminenhofstrasse 75 A, D-12459, Berlin, Germany, e-mail: thomas.baar@htw-berlin.de

## 1. Introduction

The notation of *state machines* [7, 8] is widely used in industry to specify the behaviour of software systems and is supported by numerous tools [15, 8, 9, 2]. Though the notation comes in different flavours, the core concepts *states*, *events*, *transitions*, *actions*, *state variables*, and *guards* are ubiquitous. Though there are some tools like Yakindu [9] available, which offer the generation of code into a target programming language such as C, C++, Java, etc., we will assume in this paper the (still common) situation that the implementation of the system has been implemented independently from the behaviour specification in form of a state machine.

Once both the specification and the implementation of the system has been finalized, the natural question arises whether the actual behaviour of the system is actually reflected by the specification given as a state machine.

It is tempting to tackle this *correctness question* by using formal methods to actually prove that the implementation behaves according to the specification. Abadi and Lamport show in [1], that for each correct implementation there exists a formally definable refinement mapping. However, Abadi and Lamport assume the implementation also be given in form of a state machine. Their refinement mapping connects the two artefacts *specification* and *implementation*, which both have been written in the same formalism of state machine.

Our situation is quite different. While the specification is again given in form of a state machine, the implementation can be written in *any* programming language. Even worse, we do not rely on having any formal semantics of the programming language available, but just the compiler/interpreter allowing us to execute the implementation. Thus, we actually consider the implementation as a gray box, i.e. a combination of white box and black box in the following sense: The white box characteristic is due to the fact that we can execute the implementation and, moreover, we are able to stop the execution at any time in order to inspect the internal state. For example, the internal state might be given by the values of global variables together with the set of all existing objects including the values of their attribute in case of object-orientation has been used as a programming paradigm. The black box characteristic is due to the fact that we do not have a formal semantics of the used implementation language at hand. As a consequence, there is no chance to extract and to analyze any further artefacts such as control-flow graph from the implementation.

Due to this gray box characteristic, we cannot formally prove that the actual behaviour of the implemented system is correctly described by the state machine specification. What we can do is to test the correct realization in concrete scenarios. We can stimulate the implemented system with events and then check, whether the implemented system changes its internal state the same way the state machine specification has prescribed. However, this requires to define a formal correspondence between the states of the state machine (together with the current values of the state variables) to the internal states of the implementation. This correspondence between the state spaces of state machine and implementation is done in terms of formal predicates and is called *bridge* in our approach (cmp. Section 4.).

The second question we are interested in is how precisely the state machine reflects the behaviour of the implementation. In this paper we propose a novel metric to measure the *abstractness* of a state machine w.r.t. a given implementation. While the motivation for this metric originated from assessing the efforts students have made to re-model an existing application (see Section 2.), this measurement is also helpful to detect *model degradation*. A well-known problem of model-based software development is that models tend to degrade over time: if modeling artefacts are not maintained properly while the system implementation evolves, they become less and less valuable since they do not reflect any longer structure and/or behaviour of the implementation [16].

Technically, the abstractness is measured using the technique of mutation testing.

For a predefined list of input events, both the state machine and the implementation are executed in parallel. After each event has been processed, it is checked whether both systems are in comparable states (this is checked using the bridge-predicates, which relate both state spaces). For measuring the abstractness, the parallel executions of implementation and state machine are repeated for the chosen list of input events, but now the implementation has been mutated, i.e. at some point in the implementation a statement has been changed (for example, an operator '+' has been changed to '-'). If the trace of the manipulated implementation can still be mapped correctly to the trace of the state machine, then this is a sign that the state machine does not specify the behaviour of the original implementation *precisely* (since a change in the implementation is not detected) and thus, this model is considered to be rather abstract. In the opposite case of diverged traces, we have a witness that the state machine prescribes the behaviour of the original implementation precisely, thus the state machine is considered to be rather detailed.

The remaining of the paper is organized as follows. Section 2 presents a motivating example, which already shows some pitfalls when applying the state machine notation for the specification of real-world software. Section 3 introduces the notation of state machine formally, while Section 4 formally defines the bridge for connecting the state space of state machine and implementation. The core of our approach, i.e. the technique to measure the abstractness of state machines w.r.t. a given implementation, is detailed in Section 5 In Section 6 we review relevant literature while Section 7 concludes the paper.

## 2. Motivating Example

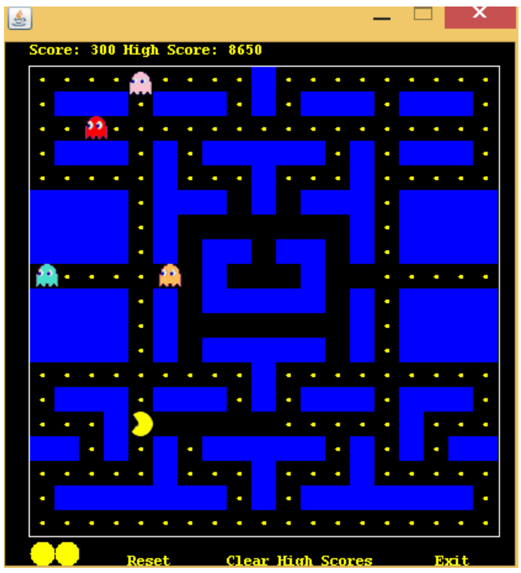


Fig 1. Screenshot Pac-Man

Suppose, you had to describe the behaviour of the classic computer game Pac-Man[1].

[1]For a detailed description see https://en.wikipedia.org/wiki/Pac-Man

Using informal language, you might solve this task be referring to Figure 1, which shows a screenshot from a open-source implementation in Java[2]. You might continue by saying that the user can move the Pac-Man via keyboard or joystick through the labyrinth and its task is to eat as many dots as possible while avoiding any collision with the enemies (the four ghosts). The Pac-Man has three lifes and a life ends by colliding with a ghost. The game is over when either the Pac-Man lost his last life or when it has eaten sufficiently many dots.

Suppose someone urges you now to use instead of informal language a modeling notation such as the ones bundled by the Unified Modeling Language (UML) [12]. Still, your model should be easily understood by a software engineer, so that she does not need to look into the implementation code to understand the rules of the game.

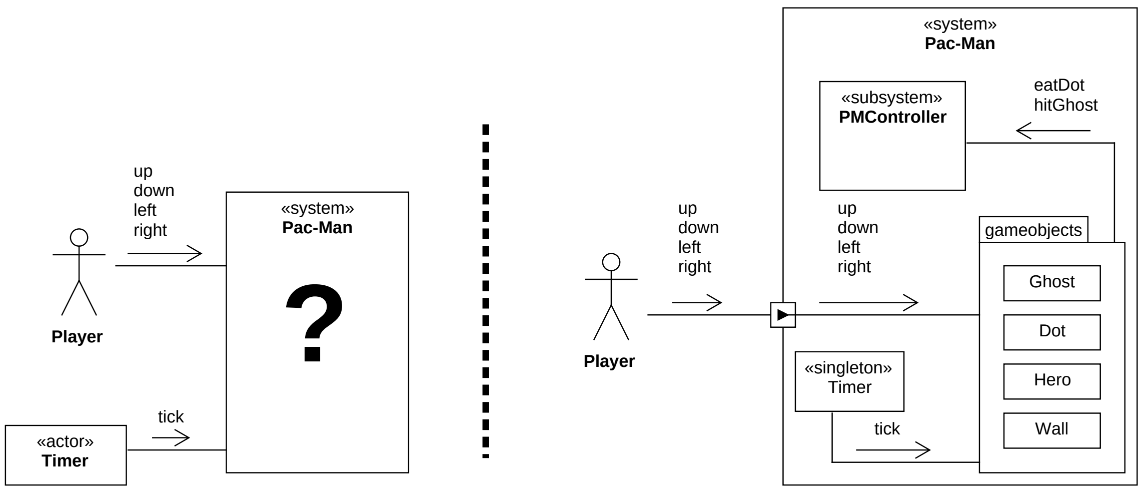


Fig 2. Naive(left) and elaborated(right) environment model – Determining the events for the state machine

Surely, the classic state machine notation seems the right formalism to be used, but before you can start you have to determine the external events to be taken into account.

Environment models as shown in Figure 2 can help a lot in this regard. You might start with a basic version shown at the left side and observe, that the system receives as input events from `Player` the four possible directions for moving Pac-Man. Furthermore, since the ghosts change their position independently of the player's input, it is quite obvious to model also an external Timer sending events `tick` to the system.

Unfortunately, it turns out to be impossible to develop a state machine which is purely based on these five events. A compelling argument is that the state machine has to reflect that Pac-Man has three lifes at the beginning and loses a life whenever it collides with a ghost. But how can the state machine detect such a collision? For this, the state machine had to keep track of the position of both Pac-Man and all ghosts.

To solve this problem you might restructure the Environment model as done at the right side of Figure 2. We split the system into a subsystem `PMController` and other

[2]The Java source code is available from `https://github.com/dtschust/javapacman`.

components for the individual game objects and the timer (whether `Timer` is an internal or external component remains a matter of taste). The original events from the player are now forwarded to the game objects. Also the event `tick` is forwared to them. Once the game objects detect a collision or that a dot was eaten by the Pac-Man, they issue new events `eatDot` and `hitGhost`, which are sent to `PMController`.

Having found now the right level of abstraction for the input events, a compact state machine reflecting the rules of the Pac-Man game can be developed quite easily, as shown in Figure 3, right side[3].

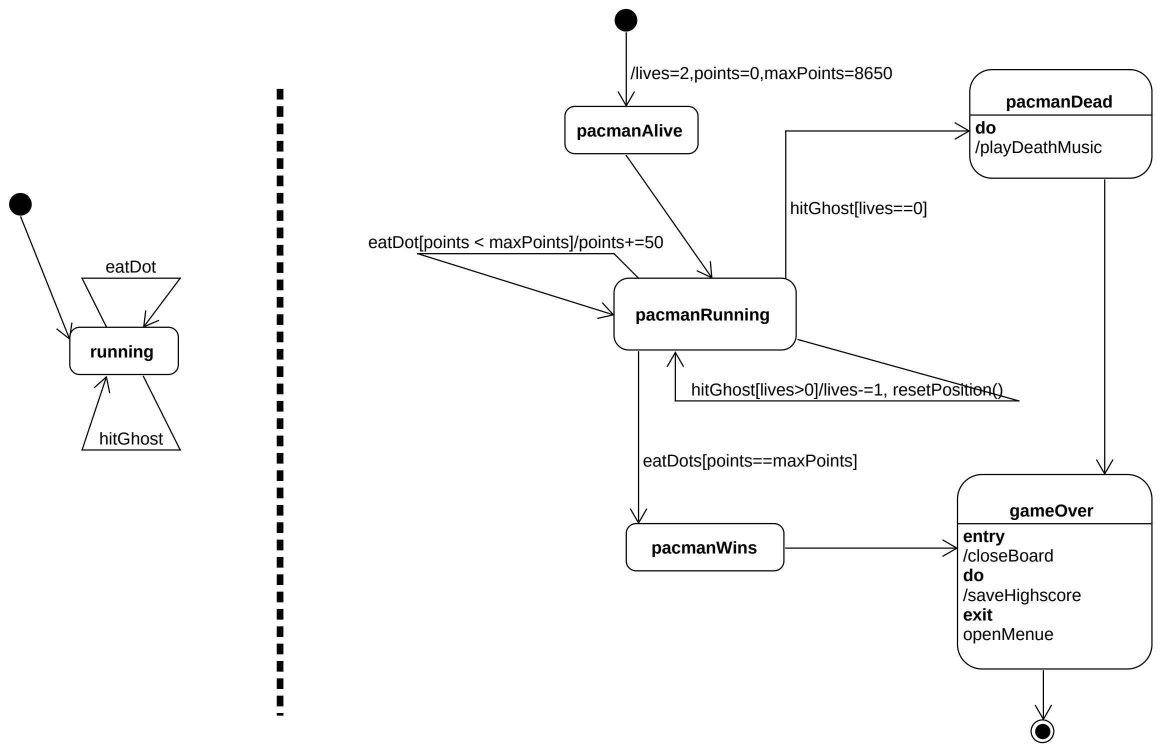


Fig 3. Useless(left) and elaborated(right) state machine

This one picture (state machine) has the same value as a lot of words. It reveals immediately how the Pac-Man game is organized and how the user can win the game. However, to decide whether this state machine actually describes the given implementation correctly, is rather challenging. Note that the implementation can only be stimulated with the four external events from the user, but that the state machine refers to the events `eatDot` and `hitGhost`, which are generated internally. When testing on the correctness of the implementation, it is crucial to find such event sequences that let the implementation generate these internal events.

A striking example, why one could be interested in measuring the abstractness of state machines w.r.t. an implementation is given in Figure 3, left side. One could argue that this state machine also describes the given implementation and it is pretty obvious,

[3]This state model is based on a student solution submitted by Fiona Brömer, Thorsten Vaterrodt, and Josefine Keller.

that the description is correct (if we do not require that after the game has been finished, the state machine must result in a final state). However, this state machine is useless since it correctly describes any other implementation dealing with the events `eatDot` and `hitGhost` as well.

In order to have a formal criterion on how we can distinguish useful from useless state machines, the metric for abstractness of a state machine is developed in this paper.

## 3. Background: State Machines

The state machine notation comes in many different variants and some of the advanced concepts such as *hierarchical state*, *parallel state*, *history state*, *state variable*, *time-triggered event*, *spontaneous transition*, *action*, *entry-/exit-action* are not supported by every tool.

In the remainder of this section, we define the version of state machines used in this paper[4]. The definition is done both in terms of syntax and semantics.

### 3.1. Syntax

We use a rather basic version of state machines. Only the concepts *state*, *start state* (always unique), *event*, *transition*, *guard*, *state variable*, and *action* (in form of parallel assignments of arithmetic expressions to state variables) are supported.

**Definition 1** (State Machine)**.** *A state machine $SM$ is a tuple $(S, Ev, V, init, trans)$ where*

- *$S$, $Ev$, $V$ are pairwise disjoint, non-empty sets of states, events, and (integer) variables, respectively*
- *$init \in (S \times Bind)$ denotes the initial state and the initial binding of variables. Here, $Bind = \{b | \, b : V \to \mathbb{Z} \cup \{\bot\}\}$ denotes the set of all possible bindings of variables to an integer value or to undefined ($\bot$)*
- *$trans \subseteq S \times S \times Ev \times Exp_{bool} \times (V \nrightarrow Exp_{int})$ denotes the set of transitions. For $(s_{pre}, s_{post}, e, g, asgmnt) \in trans$ we call $s_{pre}$ the pre-state, $s_{post}$ the post-state, $e$ the event, $g$ the guard, and $asgmnt$ the assignment of the transition*

*$Exp = Exp_{bool} \cup Exp_{int}$ denotes the set of all arithmetic/boolean expressions over the set of variables $V$. The expressions of $Exp$ are built using the usual arithmetic and boolean operators and must obey the usual type rules. By definition, all variables from $V$ are of type $int$.*

As an example, we explore the state machine $SM_{stack}$ describing the behaviour of a stack as defined in Fig. 4. Here the variable $num$ encodes the number of elements currently residing on the stack. Fig. 4 shows two equivalent definitions of $SM_{stack}$. In

[4]Tool support for the described version of state machine is available in form of the toolset SSMA (Simple State Machine Analyzer) [2]. Deduction-based analysis techniques implemented by SSMA are discussed in detail in [3].

$SM_{stack} = (S, Ev, V, init, trans)$ with
- $S = \{e, ne\}$
- $Ev = \{push, pop\}$
- $V = \{num\}$
- $init = (e, \{num \mapsto 0\})$
- $trans = \{$

$(e, ne, push, true, \{num \mapsto num + 1\},$
$(ne, ne, push, true, \{num \mapsto num + 1\},$
$(ne, ne, pop, num > 1, \{num \mapsto num - 1\},$
$(ne, e, pop, num == 1, \{num \mapsto num - 1\}$
$\}$

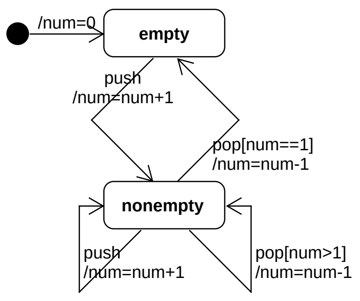


Fig 4. Example for State Machine: Stack

the left part the definition is done in a mathematical style following the definition Def. 1 whereas in the right part, a graphical version using the well-established graphical concrete syntax for state machines known from the Unified Modeling Language (UML) is used[5].

## 3.2. Semantics

The semantics of a state machine is given by describing how a state machine changes its state and the value of variables upon receiving a sequence of events. The resulting behaviour is called a *trace*.

**Definition 2** (Trace). *Let SM = (S, Ev, V, init, trans) be a state machine and inp =* $(e_1, e_2, e_3, \dots, e_n)$ *with* $e_i \in Ev$ *for* $i \in \{1, \dots, n\}$ *a finite sequence of (input) events. Let* $eval : Exp \times Bind \rightarrow \{true, false, \bot\}$ *be the usual evaluation function of expressions under a given variable binding. The evaluation w.r.t. undefinedness is strict. Then, a trace is a sequence of execution states* $(es_0, es_1, \dots, es_n)$ *where each execution state is of form* $es_i = (s_i, b_i) \in S \times Bind$ *and each of the following constraints is satisfied:*

1. $es_0 = (s_0, b_0) = init$
2. $es_{i+1} = es_i$ *if the event is rejected since there is no outgoing transition from the pre-state* $s_i$ *for the event* $e_{i+1}$*. Formally:* $\{t \mid t = (s_{pre}, s_{post}, e, g, asgmnt) \in trans\, and\, s_{pre} = s_i\, and\, e = e_{i+1}\} = \emptyset$
3. $es_{i+1} = es_i$ *if the transition cannot fire due to guard evaluation: For all transitions which are outgoing from* $s_i$ *and which are annotated with event* $e_{i+1}$*, the evaluation of the guard does not yield true. Formally:* $\{t \mid t = (s_{pre}, s_{post}, e, g, asgmnt) \in trans\, and\, s_{pre} = s_i\, and\, e = e_{i+1}\, and\, eval(g, b_i) = true\} = \emptyset$
4. *Transition is fired and bindings for variables are updated: There exists a transition* $t = (s_i, s_{i+1}, e_{i+1}, g, asgmnt) \in trans$ *with* $eval(g, b_i) = true$*. The new binding* $b_{i+1}$ *is obtained by* $b_{i+1} = b_i \leftarrow asgmnt$ *as the update of the previous binding* $b_i$ *according to the transition's assignment asgmnt*

[5]Mathematical and graphical definition differ solely in the chosen names for states. Thanks to the space efficiency of the graphical notation, we can afford here longer and more expressive state names, e.g. nonempty vs. ne.

**Example:** For the above stack example $SM_{stack}$ and for the input event sequence *inp = (pop, push, push)*, the sequence of execution states (($e$, *num* $\mapsto$ 0), ($e$, *num* $\mapsto$ 0),($ne$, *num* $\mapsto$ 1),($ne$, *num* $\mapsto$ 2)) would be a trace. Every other sequence of execution states would be not a trace for the chosen input sequence *inp*.

## 4. Bridging State Machines and Implementations

Due to the semantics as defined in the previous section, state machines can be actually used as a graphical notation to program. Tools such as Yakindu [9] offer the generation of implementation code for a given state machine. However, a feature missed in many state machine tools (including Yakindu) is the possibility to set the edited state machine in relation to a given implementation.

Recall that an (object-oriented) implementation is executed by invoking methods on objects (or, rarely, classes) and that the system state is determined by the set of currently existing objects and the values for their attributes. In contrast, the execution of state machines is determined by the list of incoming events. Moreover, the execution state of a state machine consists of the currently active state and the current binding of state variables to values.

We relate the state space of both implementation and state machine by defining a mapping as follows, the mapping is called a *bridge*.

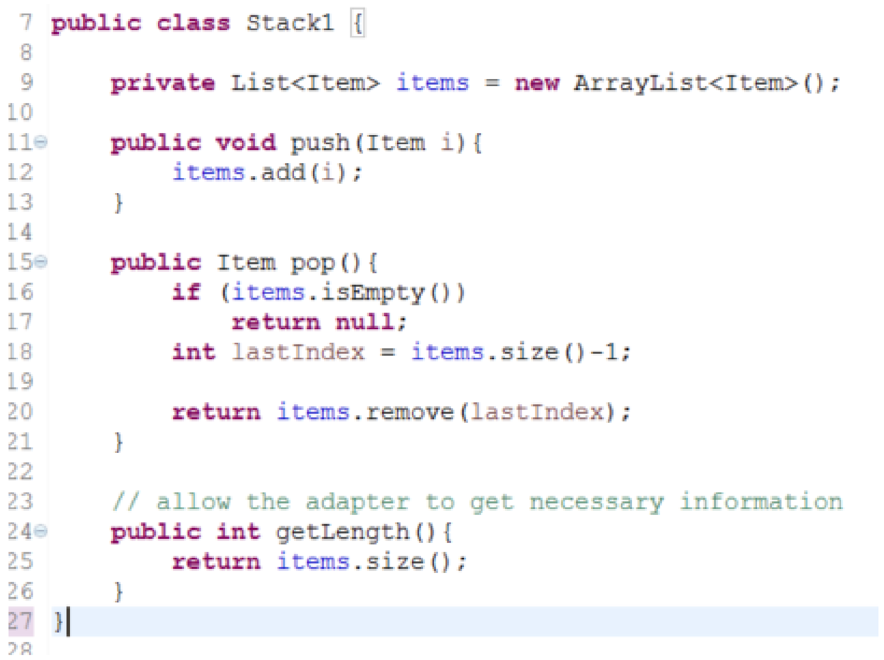
**Definition 3** (Bridge)**.** *Let $SM = (S, Ev, V, init, trans)$ be a state machine, $Impl$ an implementation, $M_{Impl}$ the set of its methods, and $S_{Impl}$ the set of all possible states the implementation can reach.*

*Then, a bridge $B(SM, Impl)$ for $SM$ and $Impl$ is a tuple $(Q, map_{ev}, Pred)$ consisting of*

- *a set $Q$ of queries $q_i : S_{Impl} \to \mathbb{Z} \cup \{\perp\}$*
- *a partial mapping $map_{ev} : M_{Impl} \nrightarrow Ev$ from methods to events*
- *a set $Pred \subseteq Exp_{bool}$ of predicates. The Boolean expressions are build with the usual boolean and arithmetic operators over variables from $V$, but quantifiers ($\forall, \exists$) are not allowed to occur. Boolean expressions can, in addition, contain subexpressions $q_i$ for accessing the current state of the implementation and the atomic predicate $inState(s)$ where $s \in S$*

**Definition 4** (Valid Bridge)**.** *Let $SM$ be a state machine, $Impl$ an implementation, and $B(SM, Impl) = (Q, map_{ev}, Pred)$ a bridge. Let $es$ be an execution state of $SM$ and $ies$ an implementation state of $Impl$.*

*We call $B(SM, Impl)$ a* valid bridge *w.r.t. the pair $(es, ies)$, iff all predicates $p \in Pred$ are evaluated to true in $(es, ies)$. The evaluation of $p$ in $(es, ies)$ is defined via structural induction on the terms occurring in $p$ as usual. The subterm $q_i$ is evaluated to $q_i(ies)$. The subterm $inState(s)$ is evaluated to true, if and only if $es$ is of form $(s, b)$ for an arbitrary binding $b$.*



```
public class Stack1 {

    private List<Item> items = new ArrayList<Item>();

    public void push(Item i){
        items.add(i);
    }

    public Item pop(){
        if (items.isEmpty())
            return null;
        int lastIndex = items.size()-1;

        return items.remove(lastIndex);
    }

    // allow the adapter to get necessary information
    public int getLength(){
        return items.size();
    }
}
```

Fig 5. Java-Implementation of Stack

**Example:** We consider a possible implementation in Java of the stack as shown in Fig. 5. A possible bridge between the state machine defined in Fig. 4 and the implementation from Fig. 5 could look as follows:

- $Q = \{c\_s\}$ where $c\_s$ evaluates in a concrete implementation state to the return value of the invocation of the method getLength()
- $map_{ev} = \{push() \mapsto push, pop() \mapsto pop\}$
- $Pred = \{inState(empty)\, implies\, c\_s = 0,$
  $inState(nonempty)\, implies\, c\_s > 0\}$

Informally speaking, the bridge maps the method calls $push()/pop()$ to the corresponding events. The two predicates actually relate the execution states of the state machine with those of the implementation and stipulate, whenever the state machine is in state *empty*, the implementation is in a state where $getLength()$ returns zero. Furthermore, if the state machine is in state *nonempty*, the invocation of $getLength()$ yields a number greater than zero.

**Definition 5** (Valid Abstraction Trace). *Let $SM$ be a state machine, $Impl$ an implementation and $B(SM, Impl)$ a bridge for them.*

*We call the trace of the state machine $(es_0, \ldots, es_n)$ induced by an event sequence $(e_1, \ldots, e_n)$ a* valid abstraction trace *iff for the corresponding implementation trace $(ies_0, \ldots, ies_n)$ the following holds: For each state pair $(es_i, ies_i)$ for $i \in \{0, \ldots, n\}$ the bridge $B(SM, Impl)$ is actually a valid bridge.*

The notion *valid abstraction trace* witnesses that the state machine has specified the behaviour of an implementation correctly for a given list of input events.

In the remainder of the paper, we call a state machine a *presumably valid abstraction* w.r.t. a given implementation, if we cannot find any sequence of events, for which the induced trace of the state machine is not a valid abstraction trace.

## 5. Measuring the Abstractness of State Machines

After we have clarified (i) syntax and semantics of the basic version of state machines used in this paper and (ii) how a state machine trace can relate to a trace of an implementation, we now present the core of our approach to measure the abstractness of a state machine.

The basic idea can be stated as follows: Let a state machine, an implementation and a bridge be given. Due to successful tests on many input event sequences we are convinced that the state machine is a presumably valid abstraction of the implementation w.r.t. the bridge.

In order to measure the abstractness of the state machine, we repeat the same tests but we use now a slightly changed implementation, a so-called *mutation* of the original implementation. If the test fails now, this means that the state machine is detailed in the sense that it is not an abstraction of the changed implementation. If the test is still successful, the opposite is true: the state machine is abstract enough to ignore the change in the implementation (at least for the current input sequence).

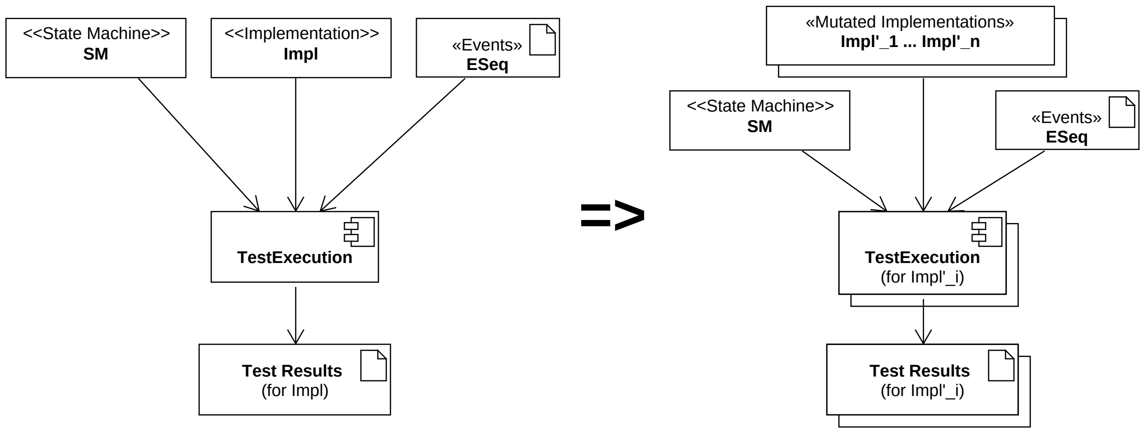


Fig 6. Repeating test on mutated implementations

Figure 6 illustrates this approach. The left part shows the parallel execution of state machine $SM$ and implementation $Impl$ for the given input event sequence $ESeq$. Since we assume that the state machine is a presumably valid abstraction, the test must have been successful, because otherwise we had a counterexample for a valid abstraction trace.

In the right part, Figure 6 shows the test executions on the mutated implementations $Impl'_1 \dots Impl'_n$. For each mutation $Impl'_i$, the test is rerun for the same sequence of input events $ESeq$. If the test fails, this is a sign that the state machine $SM$ was not more abstract than the original implementation $Impl$. If the test succeeds, the state machine was more abstract, since the current test is also a valid abstraction trace w.r.t. the mutated implementation $Impl'_i$. This brings us to the central definition of this paper:

**Definition 6** (Abstractness)**.** *Let a state machine $SM$, an implementation $Impl$ and a bridge $B = (SM, Impl)$ be given. Let $SM$ be a presumably valid abstraction of $Impl$ w.r.t. $B$. Let $Mut(Impl)$ be a fixed set of mutations of $Impl$, let $n = \#Mut(Impl)$ be the number of the considered mutations.*

*The abstractness of $SM$ for a given input event sequence $ESeq$ (denoted by $abstr_{ESeq}(SM)$) is a number between 0 and 1 and computed as*

$$\boxed{abstr_{ESeq}(SM) = \frac{k}{n}}$$

*where $k$ is the number of succeeding tests when running the test for input sequence $ESeq$ on all mutated implementations from $Mut(Impl)$.*

A mutation of implementation $Impl$ is obtained by applying a mutation operator on $Impl$. Mutation operators have been thorougly studied in mutation testing [10]. For Java implementations, mutation operators have been made generally available by the Java compiler in terms of command line options, but one can also use dedicated frameworks such as PIT [14] to mutate an existing implementation. Note that mutating a given implementation is a repeatable transformation. When applied on the same location within the implementation, a mutation operator will always produce the same mutated implementation. The number of mutants ($n = \#Mut(Impl)$) for a given implementation $Impl$ depends on i) the length and complexity of $Impl$ and ii) the set of selected mutation operators.

Modern mutation frameworks such as PIT [14] allow to control on which locations within the implementation code the mutation operators should be applied. For the Pac-Man example given in Section 2. it would desirable to allow mutations only in the implementation classes controlling the behaviour of the game objects and to avoid such mutations, for example, in GUI classes.

## 6. Related Work

Measuring the quality of modeling artefacts is traditionally done by metrics. Zhang and Hölzl propose in [18] a set of metrics for measuring the complexity of UML state machines. They apply criteria known from traditional object-oriented programming metrics [6] such as FanIn/FanOut to assess the complexity of state machines. Furthermore, they propose the refactoring of complex constructs such as Junction by a set of states and transitions before measuring the complexity to make the measured values comparable. In contrast to our approach, the metric is computed statically and does not take system executions into account.

Model-based testing (MBT) [17, 5, 4] is a widely recognized approach to take models as input for testing an executable system implementation. Compared to traditional pre-/post-state specifications as used by unit tests, the test specification is much more compact. Tools supporting MBT such as ParTeG [13] or SpecExplorer [11] support state machines (or similar notations) as a description of the expected system behaviour. From this input model, most MBT-tools generate traditional unit tests, which are then executed. If all tests succeed, then the implemented system hehaves (presumably) as specified by the state machine. MBT tools provide the ability to define a bridge between state machine and implementation. They use elaborated techniques to generate the 'right' test cases by analysing the implementation code, what makes it a white-box approach.

## 7. Conclusion

This paper presented a novel approach to measure the abstractness of state machines. For this measurement, beside the state machine, a correct implementation and a bridge from the state machine to the implementation is needed. The abstractness is a value between 0 and 1 and encodes the 'semantic distance' between state machine and implementation.

Our approach is highly flexible since it allows to combine any state machine with any possible implementation thanks to the flexible bridging mechanism. To realize a bridge, the user has to implement an adapter allowing the state machine to access the internal state of the implementation at runtime. A second adapter realizes the mapping of input events for the state machines to method calls (or any other signals) of the implementation.

## References

[1] Martin Abadi and Leslie Lamport, "The existence of refinement mappings", *Theoretical Computer Science*, **82** (1991), 253–284.

[2] Thomas Baar, "SSMA – Simple State Machine Analyzer", https://github.com/thomasbaar/simplesma.

[3] Thomas Baar, "Verification Support for a State-Transition-DSL Defined with Xtext", *Proceedings of the 10th International Andrei Ershov Informatics Conference, PSI*, Lecture Notes in Computer Science, **9609**, Springer, 2015, 50–60.

[4] Robert V. Binder, Bruno Legeard, Anne Kramer, "Model-based testing: where does it stand?", *Communications of the ACM*, **58** (2015), 52–56.

[5] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, Alexander Pretschner, *Model-Based Testing of Reactive Systems, Advanced Lectures*, Lecture Notes in Computer Science, **3472**, Springer, 2005.

[6] Shyam R. Chidamber, Chris F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, **20**:6 (1994), 476–493.

[7] David Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, **8**:3 (1987), 231–274.

[8] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, Mark B. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems", *IEEE Transactions on Software Engineering*, **16**:4 (1990), 403–414.

[9] Itemis, "Yakindu", http://statecharts.org/.

[10] Yue Jia, Mark Harman, "An analysis and survey of the development of mutation testing", *IEEE Transactions on Software Engineering*, **37**:5 (2011), 649–678.

[11] Microsoft, "SpecExplorer", https://msdn.microsoft.com/en-us/library/ee620411.aspx.

[12] Object Management Group, "Unified Modeling Language (UML), version 2.5", http://www.omg.org/spec/UML/2.5/.

[13] Stephan Weißleder, "Partition Test Generator (ParTeG)", http://parteg.sourceforge.net/.

[14] Pitest, "PIT Mutation Testing", http://pitest.org/.

[15] QuantumLeaps, “QM $^{\text{TM}}$”, http://www.state-machine.com/qm/.

[16] Bernhard Schätz, *Model-Based Development of Software Systems: From Models to Tools*, Technical University Munich, 2009.

[17] Mark Utting and Bruno Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann, 2007.

[18] Gefei Zhang and Matthias Hölzl, “A set of metrics for states and transitions in UML state machines”, *Proceedings of the 2014 Workshop on Behaviour Modelling – Foundations and Applications*, ACM, 2014.

---

**Баар Т.**, "К вопросу об измерении уровня абстракции диаграмм состояний на основе тестирования мутаций", *Моделирование и анализ информационных систем*, **24**:6 (2017), 691–703.

**DOI:** 10.18255/1818-1015-2017-6-691-703

**Аннотация.** Система обозначений диаграмм состояний (state machines) широко применяется в качестве формального средства описания поведения систем. Обычно для одной и той же программной системы можно создать много разных диаграмм состояний. Некоторые из этих моделей могут оказаться эквивалентными, но во многих случаях разные диаграммы состояний описывают одну и ту же систему на разных уровнях абстракции. В этой статье мы предлагаем подход, позволяющий провести фактическое измерение уровня абстракции диаграмм состояний по отношению к заданной реализации программной системы. Диаграмма состояний считается тем менее абстрактной, чем ближе она концептуально к реализованной системе. Согласно нашему подходу эта отдаленность диаграммы состояний от реализации системы измеряется путем применения критерия покрытия, используемого для тестирования мутации программного обеспечения. Уровень абстракции диаграмм состояний можно рассматривать как новый вид метрики. Что касается других метрик, то знание значения уровня абстракции заданной диаграммы состояний дает возможность оценить ее качество в числовых терминах. В тех проектах по разработке программного обеспечения, которые начинаются с построения модели, метрика абстракции может помочь избежать деградации моделей, поскольку она позволяет измерить фактическое отдаление спецификации поведения системы, представленной в виде диаграммы состояний, от текущей реализации системы. В отличие от прочих метрик для диаграмм состояний уровень абстракции нельзя вычислить статически, основываясь лишь на структуре самой диаграммы; для этого нужно сравнивать выполнения диаграмм состояний и соответствующую реализацию системы. Статья публикуется в авторской редакции.

**Ключевые слова:** разработка программного обеспечения на основе моделей, метрика, диаграмма состояний, тестирование мутаций

**Об авторе:**
Баар Томас, orcid.org/0000-0002-8443-1558, профессор,
Берлинская Высшая школа техники и экономики,
Wilhelminenhofstrasse 75 A, D-12459, Berlin, Germany,
e-mail: thomas.baar@htw-berlin.de