A Set of Metrics for States and Transitions in UML State Machines

Gefei Zhang
Celonis GmbH
gefei.zhang@pst.ifi.lmu.de

Matthias Hölz
Ludwig-Maximilians-Universität München
matthias.hoelzl@pst.ifi.lmu.de

ABSTRACT

UML state machines are widely used to model software system behaviors. Seemingly intuitive and simple, state machines may actually be rather complex. We present a set of metrics to reflect which model elements actually account for the complexity. Our metrics give a better understanding of the complexity of UML state machines, and may alert the modeler to pay more attention to more complex states and transitions.

Categories and Subject Descriptors
D.2 [SOFTWARE ENGINEERING]; D.2.2 [Design Tools and Techniques]; D.2.3 [Coding Tools and Techniques]; D.3.3 [Language Constructs and Features]

General Terms
Modules and Interfaces, State Diagrams, Object-oriented programming, Language Constructs and Features

Keywords
Decision Module, Requirement, Model, Program, Protocol Contracts

1. INTRODUCTION

UML state machines are widely used to model software system behaviors. They are considered to be simple and intuitive, and therefore “the most popular language for modeling reactive systems” [2]. However, this seeming ease to use disappears rapidly in UML state machines modeling non-trivial behaviors, and may become quite complex instead [9].

In fact, a state machine is simple and intuitive as long as the effect of transitions is kept local: if a transition only deactivates the state it originates from, and only activates the state it leads to, the modeler can visually follow the control flow, and the model is easy to understand.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

In realistic state machines, however, states are in general parallel and contain orthogonal regions. In such state machines it is quite common for a transition to activate not only its target, and to deactivate not only its source, but also states that are optically not directly connected to it. Meanwhile, a state can be activated or deactivated not only by a transition directly connected to it, but also by some optically “distanced” transition. This way there arises some cognitive overhead, the modeler has to carefully study all “hidden” routes of the control flow, and the state machine may get error-prone.

It is therefore important to know how complex a transition or a state is, that is, how many states, optically connected or not, are actually activated or deactivated by a certain transition, and how many transitions, optically connected or not, may actually activate or deactivate a certain state. In this paper, we discuss “hidden” activations and deactivations in detail, and define a set of metrics to indicate how much hidden effect is caused by which model element. This way, we make hidden effects more apparent.

The remainder of the paper is structured as follows: in the following Sect. 2, we give a brief overview of the concrete syntax and informal semantics of UML state machines. In Sect. 3 we discuss the hidden effects in more detail, and define our metrics. Related work is discussed in Sect. 4, before we conclude and outline some future work in Sect. 5.

2. UML STATE MACHINES

A UML state machine provides a model for the behavior of an object or component. Figure 1 shows a state machine modeling (in a highly simplified manner) the behavior of a player during a part of a game. The behavior of the player—a magician—is modeled in the state Play, which contains two concurrent regions and models two different concerns of the magician’s intelligence. The upper region describes the possible movements of the player: she starts in an entrance hall (Hall), from there she can move to a room in which magic crystals are stored (CrystalRoom), and move on to a room containing a Ladder. From this room the player can move back to the hall.

The lower region specifies the magician’s possible behaviors. She may be idle, gathering power for the next fight, Spelling a hex, or Fighting. She may escape from the fight and try to spell another hex, or, if she wins the fight, finish the game.

2.1 Concrete Syntax and Informal Semantics

This example is inspired by [10].
According to the UML specification [5], a UML state machine consists of regions which contain vertices and transitions between vertices. A vertex is either a state, which may show hierarchically contained regions; or a pseudo state regulating how transitions are compound in execution. Transitions are triggered by events and describe, by leaving and entering states, the possible state changes of the state machine. The events are drawn from an event pool associated with the state machine, which receives events from its own or from other state machines.

A state is simple, if it contains no regions (such as Hall in Fig. 1); a state is composite, if it contains at least one region; a composite state is said to be orthogonal if it contains more than one region, visually separated by dashed lines (such as Play). A region may also contain state and other vertices. A state, if not on the top-level itself, must be contained in exactly one region. A composite state and all the states directly or recursively contained in it thus build a tree.

Transitions are triggered by events (toCrystalRoom, fight). Completion transitions (not shown in this paper) are triggered by an implicit completion event emitted when a state completes all its internal activities. Events may be deferred (not shown), that is, put back into the event pool if they are not to be handled currently. By executing a transition its source state is left and its target state entered; transitions may also be declared to be internal (not shown), thus skipping the activation-deactivation scheme. An initial pseudo state, depicted as a filled circle, represents the starting point of the execution of a region. A final state, depicted as a circle with a filled circle inside, represents the completion of its containing region; if all top-level regions of a state machine are completed then the state machine terminates. For simplicity, we omit the other pseudo state kinds entry and exit points, fork and join, shallow and deep history, junction, choice, and terminate. Except join, they all can be simulated using other model elements, see [8]; join requires a slight extension of the methods presented in this paper.

At run time, states get activated and deactivated as a consequence of transitions being fired. The active states at a stable step in the execution of the state machine form the active state configuration. Active state configurations are hierarchical: when a composite state is active, then exactly one state in each of its regions is also active; when a substate of a composite state is active, so is the containing state too. The execution of the state machine can be viewed as different active state configurations getting active or inactive upon the state machine receiving events. Note that for any given region, at most one direct substate of the region can be active at any given time, because a state configuration can contain at most one direct substate of the region.

For example, an execution trace, given in terms of active state configurations, of the state machine in Fig. 1 might be (Play, Hall, Idle), (Play, Hall, Spelling), (Play, Hall, Fighting), followed by the final state, which terminates the execution.

2.2 Control Flow Fanning in and out

The complexity of UML state machines arises mainly from two causes. One is that the language is low-level, providing only if-then-else and goto like constructs (see [8]). In this sense, modeling with state machines is like programming byte code, where the programmer has to construct every program structure himself.

Another cause is that in a state machine, the control flow may be fanning in and fanning out all the time. Recall that if a composite state is active, then exactly one state in each of its regions is also active, and that the deactivation of a composite state also deactivates all its substates. Therefore, the number of currently active states is varying all the time. To understand what a state machine is supposed to do, the reader of the machine has to be keeping track of all the active states, and always be aware of which transition may cause which state to be active or inactive. This is often a non-trivial task.

For instance, in Fig. 1, the transition leaving the initial pseudo state activates not only Hall in the upper region, but also Idle in the lower region; the transition leaving Fighting deactivates not only this state, but also the state in the upper region which is currently active. Considering that in general, this kind of "hidden" (de-)activations may be recursive, it may actually cause significant complexity. In the following section, we will give our metrics to quantify this complexity.

3. METRICS

3.1 Notation

The abstract syntax of UML state machines we consider in this paper is shown in Fig. 2. UML allows many syntactic variations that complicates any static analysis of state machines. Therefore, we require some minor restrictions in addition to the UML Specification [5]:

1. A composite state may contain at most one region $r$ without an initial vertex; $r$ must contain directly or recursively a state which is the target of a transition
contains an initial vertex, we call the transition leaving this initial vertex the initial transition of \( r \), and refer to it as \( \text{intr}(r) \).

For example, in Fig. 1 the initial transition of the lower region is the one leading into the Idle state; the upper region does not have an initial transition.

**Definition 2 (Source Structured Transitions).** A transition \( t \) is called source structured, referred to as \( \text{struc}_{\text{src}}(t) \), if its source is a direct subvertex of the LCR of its source and its target.

Formally, \( \text{struc}_{\text{src}}(t) \) if \( \text{src}(t) \in \text{subvertex}(\text{LCR}(\text{src}(t), \text{tgt}(t))) \).

**Definition 3 (Target Structured Transitions).** A transition \( t \) is called target structured, referred to as \( \text{struc}_{\text{tgt}}(t) \), if its target is a direct subvertex of the LCR of its source and its target.

Formally, \( \text{struc}_{\text{tgt}}(t) \) if \( \text{tgt}(t) \in \text{subvertex}(\text{LCR}(\text{src}(t), \text{tgt}(t))) \).

Intuitively, a transition “goes through” the border of a composite state if it is source or target unstructured. Obviously, a transition may be both source and target structured, and does not need to be any. In Fig. 1, all transitions are target structured except for the one leading into state Hall from outside the Play state.

**Definition 4 (Container State in region).** Given a state \( s \) and a region \( r \), the container state of \( s \) in region \( r \) is the state \( x \) which contains (directly or recursively) \( s \) and is a direct substate of \( r \). More formally, \( \text{Csr}(r, s) = \text{subvertex}(r) \cap \{ x \mid s \in \text{substate}^+(x) \} \).

Obviously, if \( s \notin \text{substate}^+(r) \), there is exactly one element in \( \text{Csr}(r, s) \), otherwise \( \text{Csr}(r, s) \) is empty. We therefore refer to this single element as \( \text{Csr}(r, s) \).

In Fig. 1 no state has a container state in either the upper or lower region. The state Play is container state for Hall, CrystalRoom, Ladder, Idle, Spelling and Fighting in the region enclosing the whole state machine.

### 3.2 Metrics Regarding State Activation

According to the UML Specification [5], a state \( s \) can be activated in one of the following ways:

1. a transition \( t \) with \( \text{tgt}(t) = s \) is fired (transition spell to state Spelling) in Fig. 1,
2. a substate \( x \) of \( s \) is activated by a transition \( t \) where \( \text{src}(t) \notin \text{substate}^+(s) \) (Dungeon, when toLadder fires),
3. \( s \) is the target of an “initial transition” in a region, contained in composite state \( S \), and transition \( t \) with \( \text{tgt}(t) = S \) is fired (in Fig. 4 when the transition to Play is fired),
4. \( s \) is the target of an “initial transition” in a region, contained in composite state \( S \), when a state \( x \) in one of the neighbor regions of \( s \) gets activated by a target-unstructured transition \( t \) with \( \text{tgt}(t) = x \) (Idle in Fig. 4 when toLadder fires).

More formally, let \( A^T(s) \) be the set of transitions that may make state \( s \) active, we have

\[
A^T(s) = T^T(s) \cup BT(s)
\]
where

$$T^{in}(s) = \{ t \mid tgt(t) = s \lor\$$

$$\text{src}(t) \notin \text{substate}^{+}(s) \land tgt(t) \in \text{substate}^{+}(s) \land t \neq \text{intr}($$

$$\text{container}(s))\}$$

and

$$BT(s) = \begin{cases} \emptyset & \text{if } s \text{ is not target of the initial transition} \\
\cup_{t' \in IC(s)} S^{tr}(s') & \text{where } LCA(s,s') \notin LCR(s,s') \\
& \text{otherwise} \end{cases}$$

$$IC(s) = \begin{cases} \{ s \} & \text{if } s \text{ is not } tgt(\text{intr}($$

$$\text{container}(s))) \\
\{ s \} \cup IC(\text{state}($$

$$\text{container}(s))) & \text{otherwise} \end{cases}$$

$$T^{in}(s)$$ covers the first two cases, whereas $$BT(s)$$ deals with the other ones.

With these premises, we define the metric of Number of Activating Transitions of a state as the cardinality of the set of transitions that may activate it:

**Definition 5 (Number of Activating Transitions).** Given a state $$s$$, its Number of Activating Transitions is

$$NATr(s) = \#A^{tr}(s)$$

In Fig. 1 all states contained in Play are simple, therefore the number of activating transitions for each state is not surprising, e.g., for Hall it is 2, for CrystalRoom it is 1. In the lower region, the number of activating transitions for Idle is 1, the one for Spelling is 2 and the one for Fighting is 1.

Given a transition $$t$$, the set $$Act(t)$$ of states which may be activated by $$t$$ is as follows:

1. If $$t$$ is target structured and its target is a simple state, then it only activates its target.

2. If $$t$$ is target structured and its target is composite, then it activates all initial states directly contained in one of the regions of its target, and this activation continues recursively until simple states are reached.

3. If $$t$$ is not target structured, the chain of activations starts with the container state $$S$$ of its target in the region that contains both its source and target. This is the topmost state that can become active, since any state containing both source and target of $$t$$ has to be already active before $$t$$ can fire, and is not activated by $$t$$. If $$S$$ is itself the target of $$t$$, then, as in the previous case, all initial states recursively contained in $$S$$ are activated. If, however, $$S$$ is not target of $$t$$, then $$S$$ must contain the target state $$s = tgt(t)$$. In this case, $$t$$ activates all regions of $$S$$ that are not in the “path” to $$s$$ in the usual way, whereas in regions that are on the way to $$s$$ it activates these states through which it passes, independently of whether they are targets of initial transitions or not.

$$Act(t) = \begin{cases} \{ tgt(t) \} & \text{if } \text{struc}_{gt}(t) \land \text{simple}(tgt(t)) \\
\{ tgt(t) \} \cup \cup_{r \in \text{region}(tgt(t))} Act(\text{intr}(r)) & \text{if } \text{struc}_{gt}(t) \land \neg \text{simple}(tgt(t)) \\
Act_{s}(t, \text{Csr}(LCR(\text{src}(t), tgt(t))), tgt(t)) & \text{if } \neg \text{struc}_{gt}(t) \end{cases}$$

where

$$Act_{s}(t, S) = \begin{cases} \{ S \} \cup \cup_{r \in \text{region}(S), r \neq t'} Act(\text{intr}(r)) & \text{if } tgt(t) = S \\
\cup Act_{s}(t, \text{Csr}(r', tgt(t))) & \text{otherwise} \end{cases}$$

$$\text{where } r' \in \text{region}(S) \land tgt(t) \in \text{substate}^{+}(r') \text{ if } tgt(t) \neq S$$

With these premises, we define the metric Number of Activated States of a transition as the cardinality of the set of the states that may be activated by the transition:

**Definition 6 (Number of Activated States).** Given a transition $$t$$, its Number of Activated States is

$$NAS(t) = \#Act(t)$$

Applying this metric to the example given in Fig. 1, we get some interesting results. For example, let $$t$$ be the transition from the initial to Hall, then we have $$NAS(t) = 3$$, reflecting the fact that not only the obvious Hall is activated when $$t$$ is fired, but also Idle and Play. In this sense, $$t$$ is obviously more complex than the transition from Hall to CrystalRoom, which only has a $$NAS$$ of 1.

### 3.3 Metrics Regarding State Deactivation

According to the UML Specification [5], a state $$s$$ can be deactivated in one of the following ways:

1. a transition $$t$$ is activated,

2. a transition $$t$$ is activated, $$\text{src}(t) = S$$, where $$S$$ is a state containing $$s$$,

3. a transition $$t$$ is activated, $$\text{src}(t) = s'$$, where $$s'$$ is in one of the neighbor regions of $$s$$ and $$tgt(t)$$ is in a region containing $$s$$.

More formally, let $$D^{tr}(s)$$ be the set of transitions that may deactivate state $$s$$, we have

$$D^{tr}(s) = \cup_{S \in \text{superstate}^{+}(s)} T^{out}(S) \cup AT(s)$$

$$\text{where } AT(s) = \cup \{ t \mid tgt(t) \notin LCA(\text{src}(t), s) \}$$

With these premises, we define the metric of Number of Deactivating Transitions of a state as the cardinality of the set of transitions that may deactivate the state:

**Definition 7 (Number of Deactivating Transitions).** Given a state $$s$$, its Number of Deactivating Transitions is

$$NATr(s) = \#D^{tr}(s)$$

Moreover, given a transition $$t$$, the set $$Dct(t)$$ of states which may be deactivated by $$t$$ is as follows:

1. A source-structured transition from a simple state deactivates only its source state.

2. A source-structured transition $$t$$ from a composite state $$S$$ deactivates $$S$$ and, potentially, all of its substates. More precisely, $$t$$ deactivates exactly one state in each of the active regions recursively contained in $$S$$. Since any of these states may be deactivated by $$S$$ we count the number of substates in $$S$$. 
3. For a source-unstructured transition, the same considerations apply for states in regions not on the “path” of the transition; on the way from the source to the target of $t$ only these states through which $t$ passes may be deactivated.

$$Dct(t) = \begin{cases} \{\text{src}(t)\} & \text{if } \text{struc}_{\text{src}}(t) \land \text{simple}(\text{src}(t)) \\ \{\text{src}(t)\} \cup \bigcup_{r \in R} \{x \mid x \in \text{substate}^+(r)\} & \text{if } \text{struc}_{\text{src}}(t) \land \neg\text{simple}(\text{tgt}(t)) \\ Dct_\text{src}(t, S) & \text{if } \neg\text{struc}_{\text{src}}(t) \end{cases}$$

where

$$Dct_\text{src}(t, S) = \begin{cases} \{S\} \cup \text{substate}^+(S) & \text{if } \text{src}(t) = S \\ \bigcup_{r \in R, r \neq t} \text{substate}^+(r) \cup Dct_\text{src}(t, \text{Csr}(r', \text{src}(t))) & \text{if } \text{src}(t) \neq S \end{cases}$$

With these premises, we define the metric $\text{Number of Deactivated States}$ of a transition as the cardinality of the set of states that may be deactivated by the transition:

**Definition 8** (Number of Deactivated States). Given a transition $t$, its Number of Deactivated States is

$$\text{NDS}(t) = \#Dct(t)$$

Let $t$ be the transition from Fighting to the final state, it has a high NDS of 5, because not only Fighting, but also Hall, CrystalRoom, Ladder and Play will get inactive once $t$ is fired.

### 3.4 Example

While the metrics presented in this paper represent only a rough estimate of the complexity of various states and transitions in a state machine, we believe that they could serve a useful purpose in alerting modelers to unexpected features of their models. For example, Fig. 4 contains a slightly larger model of the game used as example, that contains several potential mistakes by the modeler that can be identified using the metrics presented in this paper. The game now consists of two areas, a laboratory (Lab) in which the wizard may rest (in state Idle) or brew potions (in state BrewPotion) and the multi-room dungeon (Dungeon) in which fights take place. The wizard enters the level from the hall of the dungeon, and can use the ladder of the dungeon to escape to her lab and later return to the dungeon. In addition, she can take a potion before fighting (TakePotion) which will increase the power of her next spell tenfold (PowerSpelling and PowerFighting).

The first potential mistake in this state machine is that the wizard can escape to her laboratory whenever she is in the ladder room, even during a fight. This is not immediately obvious from the lower region of the state machine which describes the behavior of the wizard. When looking at the number of deactivating transitions for the states in the lower region, however, it becomes immediately obvious that these states can not only be left by the locally visible transitions, but also by transitions from other regions.

A slightly more sophisticated analysis can show that it is possible for the wizard to obtain arbitrarily large power ups: State PowerFighting post-dominates [1] states TakePotion and PowerSpelling, i.e., each path from TakePotion or PowerSpelling to a final state goes through PowerFighting. In this case, it is likely that the modeler assumes that the exit action of PowerFighting will always be executed after one of the post-dominated states is reached. By computing the number of deactivating transitions for the post-dominated states and comparing them to the locally visible outgoing transitions it becomes again possible to determine that the model has a high probability of containing a mistake.

### 4. RELATED WORK

Complexity metrics of state machines have been recognized as useful indicators [7]. Hierarchical states are considered in [3; 4]. These approaches first transform UML state machines into flat, non-hierarchical state transition systems, and then calculate the complexity of the state transition system. In comparison, our approach is based on UML elements. The calculation is therefore easier, and it is also more clear which elements actually cause the complexity.

Cyclomatic complexity [6] is a very widely-used metric of state-based systems. Like the approaches cited above, cyclomatic complexity is also only applicable to flat state transition systems, and is therefore, in the domain of UML, not as direct as our approach.

The determination of the transitions activating or deactivating a certain state is also an essential technique for weaving aspect-oriented state machines [10].

### 5. CONCLUSIONS AND FUTURE WORK

We have discussed in detail the activation and deactivation of (hierarchical) composite states in UML state machines, and, based on this discussion, defined four metrics to reflect the complexity of transitions leading to or leaving composite states. Our metrics give a better understanding of the complexity of UML state machines than traditional metrics. They also show where the modeler or reader of UML state machines must pay attention.

We plan to validate our metrics in more realistic models, and also to implement support for the metrics in modeling tools.

### Acknowledgement

This work has been partially sponsored by the EU project ASCENS, 257414.

### References


Figure 4: Two-room game example


