A Set of Metrics of Non-locality Complexity in UML State Machines

Gefei Zhang[1(✉)] and Matthias M. Hölzl[2]

[1] Celonis GmbH, München, Germany
gefei.zhang@pst.ifi.lmu.de
[2] Ludwig-Maximilians-Universität München, München, Germany
matthias.hoelzl@pst.ifi.lmu.de

Abstract. One of the barriers to widespread adoption of behavior modeling languages lies in the complexity of the models. We show in the context of UML state machines how non-locality, i.e., the information for the current behavior of a model being spread over several model elements instead of being locally available, may make seemingly intuitive and simple models rather complex and error-prone. We present a set of metrics to measure the complexity of UML state machines arising from different kinds of non-locality. Our metrics give a better understanding of the complexity of UML state machines, and may alert the modeler to pay more attention to pitfalls in *apparently simple* UML state machines.

1 Introduction

In Software Engineering research, Model-Driven Engineering (MDE) has been recognized as "a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively" [6]. However, in practice, MDE has not yet experienced broad acceptance. The reasons are manifold [7]. In our opinion, one important reason that has not yet been widely discussed in the literature is that behavioral models are, despite the best intention of the modeler, sometimes counter-intuitive and hard to comprehend.

UML state machines are widely used to model software system behaviors and have been described as "the most popular language for modeling reactive systems" [2]. In literature about software or behavior modeling, UML state machines are generally considered to be simple and intuitive. However, these intuitions can easily be misleading when UML state machines are used to model non-trivial behaviors [13].

In general, a state machine is only 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.

In many realistic state machines, however, states are 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

© Springer International Publishing Switzerland 2015
E. Roubtsova et al. (Eds.): BM-FA 2009-2014, LNCS 6368, pp. 59–81, 2015.
DOI: 10.1007/978-3-319-21912-7_3

states that are visually not directly connected to it. Conversely, a state can be activated or deactivated not only by a transition directly connected to it, but also by visually “remote” transitions. Moreover, transitions and states in different regions may depend on each other in complex ways. In these situations the modeler has to carefully study all information carried by remote model elements to avoid introducing errors into the state machine.

It is therefore important to study this kind of complexity, which we call non-locality complexity. In this paper, we discuss in detail “hidden” activations and deactivations as well as cross-region dependencies, and we define a set of metrics to measure the non-locality complexity caused by states or transitions, that is, how many states—visually connected or not—are actually activated or deactivated by a certain transition, how many transitions—visually connected or not—may actually activate or deactivate a certain state, how many transitions—in different regions—are fired in one execution step, and how many states—in different regions—may prevent each other from getting active. These metrics quantify various non-local effects of a state machine’s behavior and may alert developers to potential pitfalls.

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, and show how non-locality complexity may arise. In Sect. 3 we discuss the non-local effects in more detail, define our metrics, and give some application samples of how the metrics may indicate possible flaws in a state machine. 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.[1] 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 conduct. 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 activities. 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

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*,

[1] This example is inspired by [14].

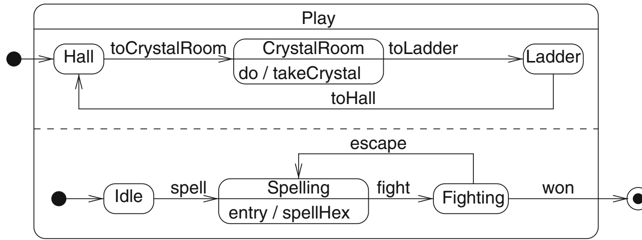


Fig. 1. Example: UML state machine

which may show hierarchically contained regions; or a *pseudo state* regulating how transitions are compounded 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 states and other vertices. A state, if not on the top-level itself, must be contained in exactly one region. To simplify the notation in later sections we assume in this paper that the top-level state is enclosed in a region `top`, so that each state is contained in exactly one region. A composite state and all the states directly or recursively contained in it thus build a tree. A state may have an *entry*, a *do*, and an *exit* action, which are executed before, whilst, and after the state is active, respectively.

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. A transition may have an *effect*, which is an action to be executed when the transition if fired. Very briefly speaking, the result of firing a transition is that the source state of the transition (and potentially other states, see Sect. 3.3) is left, its target state (and potentially other states, see Sect. 3.2) entered, and the entry and exit actions are executed; for more details, see below. 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 *junction* pseudo state, depicted as a dot, chains transitions together. 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, choice, and terminate. These vertices, except for joins, can be simulated using states and transitions only, see [10]; joins require 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.

When an event in the event pool is processed, first the maximum set of enabled conflict-free transitions is calculated. We refer to transitions where the source state is active, the trigger is the current event, and the guard is evaluated to `true` as *enabled* transitions. At runtime, if there are several enabled transitions where the source states are in different regions, then all the transitions are in general contained in the set, and should be fired in one execution step.[2] A special case, however, is when the source of one of the enabled transitions is contained in the source of another enabled transition. In this case, only the transition with the *innermost* source state is contained in the maximum set of conflict-free enabled transitions. If there are several enabled transitions whose sources are in the same region, then they all have the same source state, since in one region there is at most one active state (here: exactly one). In this case, we say the enabled transitions (from the same source) are *conflicting*, and only one of them, chosen non-deterministically, is contained in the maximum set of conflict-free enabled transitions.

The transitions in the maximum set of conflict-free enabled transitions are then fired in one execution step. More specifically, first the source states, and other states to be deactivated (see Sect. 3.3), are deactivated, then their exit actions are executed, then the effects of the transitions are executed, then the entry actions of the target states, and other states to be activated (see Sect. 3.2), are executed, and finally these states are activated. For the entry and exit actions, the UML Specification [5] imposes a partial order of execution: for two states s and S, if s is directly or recursively contained in S, then the entry action of S is executed before that of s when the states are activated, and the exit action of s is executed before that of S when the states are deactivated. In the other cases, where s and S are in different regions and are not contained in each other, no special order of execution is specified. In the metrics we are going to define in this paper, we do not make use of this partial order. In the future, more precise metrics can be defined if this partial order is also taken into account.

Since the maximum set of enabled transitions is determined *before* the execution of the entry actions, the transitions' effects, and the exit actions, these actions have no effect on the choice of transitions to be fired.

In our example of Fig. 1, an execution trace, given in terms of active state configurations the state machine, might be (`Play`, `Hall`, `Idle`), (`Play`, `Hall`,

[2] Strictly using the terms defined in the UML Specification, this is a special case of the *run-to-completion* step. For simplicity, we do not use the concept run-to-completion in this paper. Since we ignore most of the pseudo states, see Sect. 3.1, a run-to-completion event is actually very similar to our execution step.

Spelling), (Play, Hall, Fighting), followed by the final state, which terminates the execution trace.

2.2 Non-locality Complexity

While the simplest UML state machines may be intuitively comprehensible, the complexity increases rapidly when the behavior under modeling gets more involved [13]. There are several reasons for this increase in complexity. One is that the language is low-level, providing only *if-then-else* and *goto* like constructs (see [10]). In this sense, modeling with state machines is similar to programming in assembly language, where the programmer has to implement every program structure without the benefit of abstractions built into the language.

The metrics we propose in this paper are, however, mostly concerned with another cause for complexity in state machines: the control flow in a region is often determined not only by information that is available "locally", i.e., stored in currently active states or transitions just before or after being fired. Instead, relevant information is "hidden" in model elements that are not directly connected with active states and transitions. Recall, for instance, that when a transition t is fired, not only is its target s activated, but, if s is a substate in a composite state S not containing the source of t, so is the state S (if a substate is active, then so is its containing state, too). Therefore, in each region of S exactly one of the contained states, although not connected directly with s or t, is also activated. Similarly, when a transition t is fired, not only is its source s deactivated, so are all states S containing s but not the target of t, and hence all states that are directly or recursively contained in S, although they are not connected directly with s or t.

As an example, consider 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 this kind of "remote" activation and deactivation may be recursive, it may cause significant potential for misinterpretation of the state machine's actual behavior by the modeler.

Moreover, different, parallel regions of a state are not executed independently of each other. Instead, there often exist cross-region dependencies within a state. For instance, enabled transitions with the same trigger and source states in different regions are normally fired in the same execution step—unless something (like a guard in a nested region) prevents some of these transitions from firing. To understand what a state machine is supposed to do, the reader of the machine has thus to keep track of a lot of non-local information—a guard that prevents a transition from firing does not have to be guarding the transition it inhabits or, for that matter, be anywhere close to this transition in the visual representation of the state machine.

We will present more examples of this kind of complexity, which we call *non-locality complexity*, later on. In the following, we define metrics to measure non-locality complexity.

3 Metrics

We first review the metamodel of UML state machines, define auxiliary functions, and then define metrics to capture the non-locality complexity.

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 complicate 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 t, and t's source is not contained in r.[3]
2. The state machine must not contain junctions. All junctions, except the ones following an initial vertex, should be removed by the (semantics-preserving) transformation shown in Fig. 3, which essentially merges each pair of junction-connected transitions into one transition, with the conjuncture of the two original guards as new guard. For simplicity, we do not consider state machines with junctions following an initial vertex in this paper.

The first constraint applies only to states with multiple regions and serves to clarify their semantics: in the case of multiple regions the behavior of a state machine that does not satisfy this restriction is not clear. The second constraint actually slightly restricts the range of possible state machines; it would be possible to lift this restriction by a straightforward extension of our metrics.

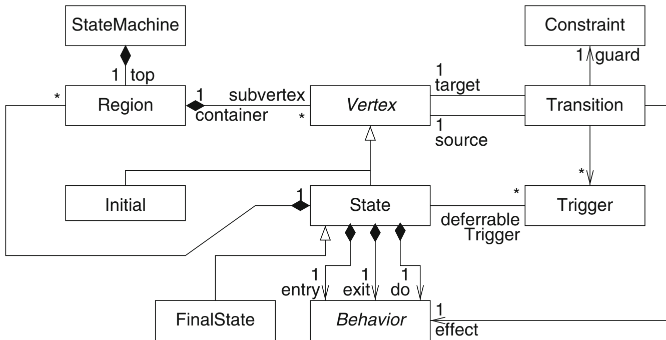
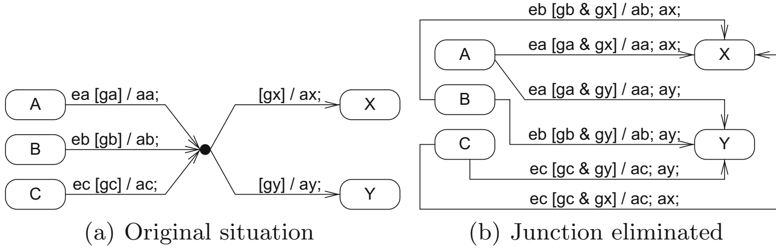


Fig. 2. Metamodel: UML state machines (simplified)

Given states s_1 and s_2, we write $\mathsf{LCR}(s_1, s_2)$ for the least region containing both s_1 and s_2, and $\mathsf{LCA}(s_1, s_2)$ for the least state containing both s_1 and

[3] In other words, t is a *target-unstructured* transition, see below, Definition 3.



(a) Original situation (b) Junction eliminated

Fig. 3. Removing junctions

s_2. We suppose that on the top-level, a state machine consists in a region (called `top`), which contains the top-level vertices of the state machine, see Sect. 2.1. Therefore, for any s_1 and s_2, $\mathsf{LCR}(s_1, s_2)$ is well-defined. If there is no state containing both s_1 and s_2 (i.e., $\mathsf{LCR}(s_1, s_2) = \texttt{top}$), we write $\mathsf{LCA}(s_1, s_2) = \bot$. In Fig. 1, $\mathsf{LCR}(\texttt{Spelling}, \texttt{Fighting})$ is the lower region of `Play` and $\mathsf{LCA}(\texttt{Spelling}, \texttt{Fighting}) = \texttt{Play}$.

Given region r, we write $\mathsf{substate}^+(r)$ to represent all states (directly or recursively) contained in r. For a state s, we write $\mathsf{substate}^+(s)$ to represent $\bigcup_{r \in \mathsf{region}(s)} \mathsf{substate}^+(r)$ and write $\mathsf{substate}^*(s)$ to represent $\mathsf{substate}^+(s) \cup \{s\}$. We write $S \in \mathsf{superstate}^*(s)$ if $s \in \mathsf{substate}^*(S)$. We write $\mathsf{simple}(s)$ if state s is simple. Given states S and $s \in \mathsf{substate}^+(S)$, we write $s \in S$, otherwise we write $s \notin S$.

We refer to property p of object o as $p(o)$. If the name of a property is not given explicitly, we follow the common UML convention and use, independently of the multiplicity, the lower-cased name of its type as the property name. For example, we write $\mathsf{state}(r)$ for the state associated with a region r according to UML metamodel given in Fig. 2.

In the following, we define some more auxiliary notations.

Definition 1 (Initial Transition). *If a region r contains an initial vertex, we call the transition leaving this initial vertex the* initial transition *of r, and refer to it as $\mathsf{intr}(r)$.*

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 $\mathsf{struc}_{\mathsf{source}}(t)$, if its source is a direct subvertex of the LCR of its source and its target. More formally, $\mathsf{struc}_{\mathsf{source}}(t)$ is* ***true*** *if $\mathsf{source}(t) \in \mathsf{subvertex}(\mathsf{LCR}(\mathsf{source}(t), \mathsf{target}(t)))$.*

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

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 either. 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,* $\mathsf{Csr}(r, s) = \mathsf{subvertex}(r) \cap \{x \mid s \in \mathsf{substate}^+(x)\}$. *Obviously, if* $s \in \mathsf{substate}^+(r)$*, there is exactly one element in* $\mathsf{Csr}(r, s)$*, otherwise* $\mathsf{Csr}(r, s)$ *is empty. We therefore refer to this single element as* $\mathsf{Csr}(r, s)$*.*

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

The activation and deactivation of states in UML state machines is relatively complex, because concurrent and nested regions may be involved. We therefore introduce in Fig. 4 a slightly more complicated variant of the game in Fig. 1. 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 (`takeBrew`) which will increase the power of her next spell tenfold (`PowerSpelling` and `PowerFighting`). In Sect. 3.5 we will use the metrics defined in this paper to show that this model has several possible modeling mistakes; for now we are only interested in the activation and deactivation of states by transitions.

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

1. a transition t with $\mathsf{target}(t) = s$ is fired (transition `spell` to state `Spelling`) in Fig. 1,
2. a substate x of s is activated by a transition t where $\mathsf{source}(t) \notin \mathsf{substate}^*(s)$ (`Play` when the "initial transition" from the outer region to `Dungeon` is fired; `Dungeon`, when the transition `toLadder` from state `Idle` in `Lab` is fired),
3. s is the target of an "initial transition" in a region, contained in composite state S, and transition t with $\mathsf{target}(t) = S$ is fired (`Hall` and `Idle` inside `Dungeon` in Fig. 4 when the "initial transition" to `Dungeon` 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 $\mathsf{target}(t) = x$ (state `Idle` in the lower region of `Dungeon` in Fig. 4 when `toLadder` from the state `Idle` in `Lab` is fired).

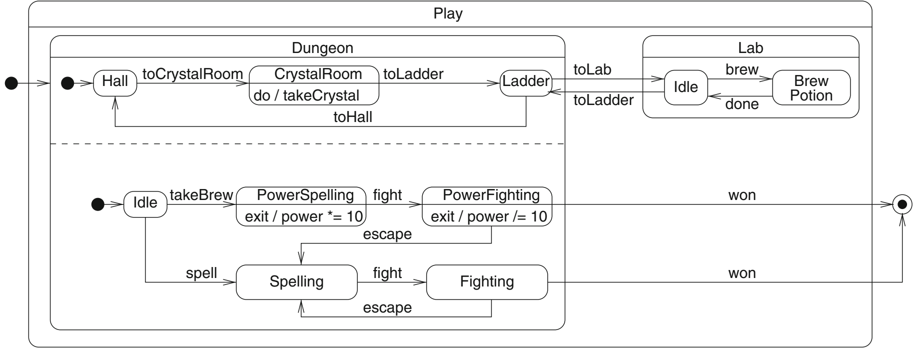


Fig. 4. Two-room game example

To capture this notion precisely we first define a function orth that returns the set of all states contained in regions orthogonal to the one containing s, i.e., all states either directly in a region orthogonal to the region containing s or recursively contained in a state in such a region:

$$\begin{aligned} \mathsf{superstate}(s) &= \mathsf{state}(\mathsf{container}(s)) \\ \mathsf{orth}(s) &= \{s' \in \mathsf{substate}^+(\mathsf{superstate}(s)) \mid \mathsf{LCA}(s,s') \in \mathsf{LCR}(s,s')\} \end{aligned}$$

We can now define the set of transitions that may make state s active, $A^{tr}(s)$ more precisely. $A^{tr}(s)$ is the least fixed point of the equations

$$A^{tr}(s) = T^{in}(s) \cup BT(s) \cup PT(s)$$

where

$$T^{in}(s) = \{t \mid \mathsf{target}(t) = s \vee (\mathsf{source}(t) \notin \mathsf{substate}^*(s) \wedge \mathsf{target}(t) \in \mathsf{substate}^+(s))\}$$

$$BT(s) = \begin{cases} \emptyset & \text{if } s \text{ is not target of an initial transition} \\ \bigcup_{s' \in IC(s) \setminus \{s\}} A^{tr}(s') & \text{otherwise} \end{cases}$$

$$IC(s) = \begin{cases} \{s\} & \text{if } s \text{ is not } \mathsf{target}(\mathsf{intr}(\mathsf{container}(s))) \\ \{s\} \cup IC(\mathsf{superstate}(s)) & \text{otherwise} \end{cases}$$

$$PT(s) = \begin{cases} \emptyset & \text{if } s \text{ is not target of an initial transition} \\ \left\{t \in \bigcup_{s' \in \mathsf{orth}(s)} A^{tr}(s') \mid \mathsf{source}(t) \notin \mathsf{orth}(s)\right\} & \text{otherwise} \end{cases}$$

$T^{in}(s)$ covers the first two cases, whereas $BT(s)$ deals with the third and $PT(s)$ captures the fourth one. A^{tr} is defined as fixed point to cover cases like the ones depicted in Fig. 5: Since both B and D are initial states in their respective regions, $A^{tr}(B)$ has to take into account $A^{tr}(D)$ when determining $PT(B)$, but $A^{tr}(D)$ in turn relies on $A^{tr}(B)$. In Fig. 5, the transition from `A` to `B` and the transition from `C` to `D` both activate states `B` and `D`.

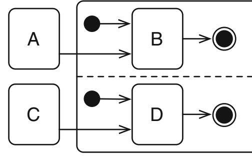


Fig. 5. State machine that illustrates the need for fixed points in A^{tr}

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 $s = \mathsf{target}(t)$. 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.

$$Act(t) = \begin{cases} \{\mathsf{target}(t)\} \\ \qquad \text{if } \mathsf{struc}_{\mathsf{target}}(t) \wedge \mathsf{simple}(\mathsf{target}(t)) \\ \{\mathsf{target}(t)\} \cup \bigcup_{r \in \mathsf{region}(\mathsf{target}(t))} Act(\mathsf{intr}(r)) \\ \qquad \text{if } \mathsf{struc}_{\mathsf{target}}(t) \wedge \neg \mathsf{simple}(\mathsf{target}(t)) \\ Act_s(t, \mathsf{Csr}(\mathsf{LCR}(\mathsf{source}(t), \mathsf{target}(t)), \mathsf{target}(t))) \\ \qquad \text{if } \neg \mathsf{struc}_{\mathsf{target}}(t) \end{cases}$$

where

$$Act_s(t, S) = \begin{cases} \{S\} \cup \bigcup_{r \in \mathsf{region}(S)} Act(\mathsf{intr}(r)) \text{if } \mathsf{target}(t) = S \\ \bigcup_{r \in \mathsf{region}(S), r \neq r'} Act(\mathsf{intr}(r)) \cup Act_s(t, \mathsf{Csr}(r', \mathsf{target}(t))) \\ \qquad \text{where } r' \in \mathsf{region}(S) \wedge \mathsf{target}(t) \in \mathsf{substate}^+(r') \\ \qquad \text{if } \mathsf{target}(t) \neq S \end{cases}$$

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, $\mathsf{source}(t) = s$,
2. a transition t is activated, $\mathsf{source}(t) = S$, where S is a state containing s,
3. a transition t is activated, $\mathsf{source}(t) = s'$, where s' is in one of the neighbor regions of s and $\mathsf{target}(t)$ is in a region containing s.

An example for the first kind of deactivation is the transition from `Fighting` to `Spelling` in Fig. 4. A transition from `Dungeon` to `Lab` would be an example for the second case. The third kind of deactivation happens, e.g., for state `Fighting` when the transition from `Ladder` to `Idle` is taken.

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

$$D^{tr}(s) = \bigcup_{S \in \mathsf{superstate}^*(s)} T^{out}(S) \cup AT(s)$$

where

$$AT(s) = \bigcup \{t \mid \mathsf{target}(t) \notin \mathsf{LCA}(\mathsf{source}(t), s)\}$$

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 Deactiviting Transitions). *Given a state s, its* Number of Deactivating Transitions *is*

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

Deactivation is simpler than activation since no "cascading deactivations" may happen: A transition will deactivate all states in regions contained in its source and all states in regions "between" its source and target, but it may not trigger additional deactivations in regions inside its target state as is the case for activations. In Fig. 4, state `Fighting` has 3 deactivating transitions: from `Fighting` to `Spelling`, from `Fighting` to the final state, and from `Ladder` (in the upper region) to `Idle`.

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.

$$Dct(t) = \begin{cases} \{\mathsf{source}(t)\} \\ \qquad \text{if } \mathsf{struc}_{\mathsf{source}}(t) \wedge \mathsf{simple}(\mathsf{source}(t)) \\ \{\mathsf{source}(t)\} \cup \bigcup_{r \in R} \{x \mid x \in \mathsf{substate}^+(r)\} \\ \qquad \text{if } \mathsf{struc}_{\mathsf{source}}(t) \wedge \neg\mathsf{simple}(\mathsf{target}(t)) \\ Dct_s(t, S) \\ \qquad \text{if } \neg\mathsf{struc}_{\mathsf{source}}(t) \end{cases}$$

where

$$Dct_s(t,S) = \begin{cases} \{S\} \cup \mathsf{substate}^*(S) \\ \qquad \text{if } \mathsf{source}(t) = S \\ \bigcup_{r \in R, r \neq r'} \mathsf{substate}^*(r) \cup Dct_s(t, \mathsf{Csr}(r', \mathsf{source}(t))) \\ \qquad \text{where } r' \in \mathsf{region}(s) \wedge \mathsf{source}(t) \in \mathsf{substate}^+(r') \\ \qquad \text{if } \mathsf{source}(t) \neq S \end{cases}$$

With these premises, we define the metric *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*

$$NDS(t) = \#Dct(t)$$

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 Metrics Regarding Cross-Region Dependency

Cross-region dependencies may have two forms: some "actions" (transitions being fired, states being activated or deactivated) in different regions are carried out simultaneously, while others may prevent each other from being carried out. These dependencies crosscut model elements across several regions, their comprehension requires careful study of the state machine.

In the following, we use another extension of our computer game to illustrate cross-region dependencies, see Fig. 6. In this variant of the game, the navigation region contains another state `CrystalPedestal` that can be reached from the `CrystalRoom` upon the event `investigate` whenever the player is near a pedestal. The `Idle` state in the region responsible for the player's activities has been refined by a state machine that describes the various activities the player can perform while she is idle: she can `Wander` around the room; while wandering the player may `trip` and enter into a `Curious` state in which she has the possibility to `investigate` the reason for tripping, or to `ignore` the incident and continue wandering. If the player does investigate while she is in the `Curious` state two courses of action may unfold: Either there was no particular reason for tripping, in which case the investigation deducts some health points (not shown in the state machine) and the player continues to wander, or the player tripped over a hidden trap door (guard `TD`) in which case she discovers a trap door leading to fame and fortune, and thus immediately wins the level without having to fight. Whenever the player is in the `Idle` state she can `investigate` her surroundings and thereby gather new information about the room or objects nearby.

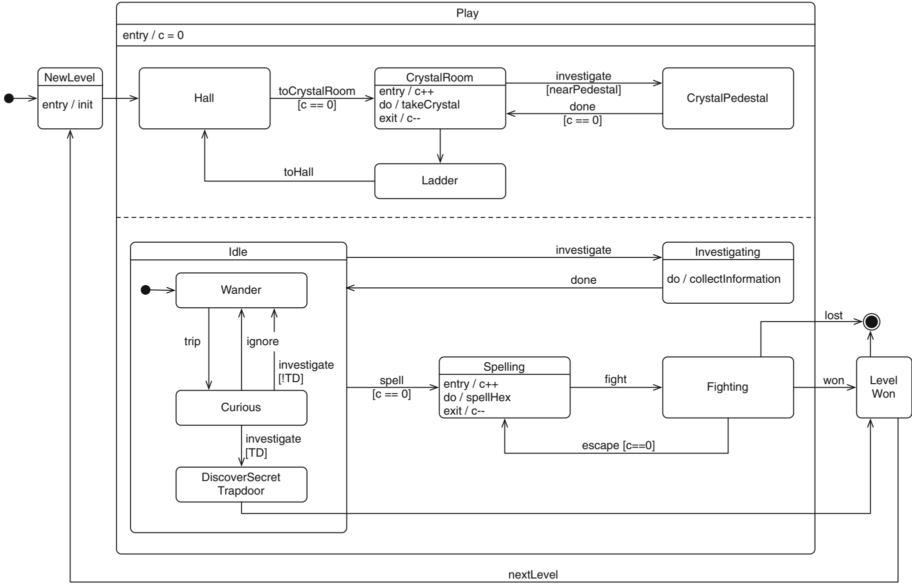


Fig. 6. Game with cross-region dependencies

In this version of the game we have used the `investigate` event on three different transitions: to change from the `CrystalRoom` to the `CrystalPedestal` with the guard `nearPedestal`, to change from `Idle` to `Investigating` unconditionally, and from `Wander` to `Curious` (both substates of `Idle`) when the guard `TD` holds. Whenever the transition inside the `Idle` state is enabled, the transition from `Idle` to `Investigating` cannot fire. This shows that submachines are not compositional, i.e., the addition of a submachine to a state may invalidate invariants of the containing machine. It is often convenient to model in this way. For example, if the `investigate` event is mapped to a physical event generated by controller hardware, the state machine in Fig. 6 represents the different results of pressing this button in various game situations in a very concise manner. However, the "stealing" of events by interior transitions is also a source of possible mistakes, in particular when working with modeling tools that allow modelers to hide nested states. Therefore we consider it useful to investigate metrics that can draw attention to these kinds of potential problems. In the following we will therefore propose some measures for the influence exerted by states or transitions on the execution of other regions of the state machine.

Simultaneous Initial Transitions. One simple case of transitions being fired in one execution step is that when a composite state S is activated, either via a transition $t, \mathsf{target}(t) = S$, or via a transition $t', \mathsf{target}(t') \in \mathsf{substate}^+(S)$, then the initial transitions of the regions of S are also fired. Details were described in Sect. 3.2.

Given an initial transition t, for each state in the set $Act(t) \setminus \{\mathsf{target}(t)\}$, there is another transition which is fired in the same execution step as t. The complexity measure for this situation is therefore given by the cardinality $NAS(t) - 1$, see page 11.

Bonded Transitions. Another case of transitions being fired in one execution step is that when transition t_1 is fired, another transition t_2 from a parallel region is also fired in the same execution step if all the four following conditions are satisfied:

1. the source states are both active,
2. $\mathsf{guard}(t_1)$ and $\mathsf{guard}(t_2)$ both hold,
3. both transitions have the same trigger,
4. the source states are in different, parallel regions.

To capture situations where these conditions may hold we define the set of transitions that are bonded to a given transition.

Definition 9 (Bonded Transitions). *Given a transition t, the set of transitions that are* bonded *to t is $\overline{\mathcal{B}}(t) = \{t' \mid \mathsf{Can}(t, t')\}$, where the predicate* Can *is defined as* $\mathsf{Can}(t, t') ::= \mathsf{trigger}(t) = \mathsf{trigger}(t') \wedge \mathsf{LCA}(\mathsf{source}(t), \mathsf{source}(t')) \in \mathsf{substate}^{+}(\mathsf{LCR}(\mathsf{source}(t), \mathsf{source}(t')))$.

The definition states that two transitions t and t' are *bonded*, written as $\mathsf{Can}(t, t')$, when the last two of the four conditions above are satisfied. The first part of the predicate Can reflects condition 3, and the second part reflects condition 4. However, the converse is not true: The information to determine if the other two conditions hold is only available at runtime. Since this paper is concerned with static analysis, we ignore these two conditions, and *over-estimate* the run-time value instead.

The set $\overline{\mathcal{B}}(t)$ may also contain transitions that are not fired when t is fired: suppose $t', t'' \in \overline{\mathcal{B}}(t), \mathsf{source}(t'') \in \mathsf{substate}^{+}(\mathsf{source}(t'))$, and at some time during the state machine's execution, $\mathsf{source}(t'')$ (and therefore $\mathsf{source}(t')$) is active. If the current event is $\mathsf{trigger}(t) = \mathsf{trigger}(t') = \mathsf{trigger}(t'')$, then, according to the UML Specification [5], t'' is fired while t' is not, since the source of t'' is a substate of t'. For example, when `Curious` (and thus `Idle`) is active, since one of the conditions `[TD]` and `[!TD]` is always `true`, the transition from `Idle` to `Investigating` in Fig. 6 will not be fired, even if the current event is `investigate`.

In the next definition, we restrict the set to contain only transitions that we know will definitely be fired when t is fired. The flip side of the definition is that it may ignore those transitions t' such that $\mathsf{Can}(t, t')$ holds but there is some $t'', \mathsf{source}(t'') \in \mathsf{substate}^{+}(\mathsf{source}(t')) \wedge \mathsf{Can}(t, t'')$.

Definition 10 (Strictly Bonded Transitions). *Given a transition t, the set of transitions that are* strictly bonded *to t is* $\mathcal{B}(t) = \{t' \mid \mathsf{Can}(t, t') \wedge \nexists t'' \cdot [\mathsf{Can}(t, t'') \wedge \mathsf{source}(t'') \in \mathsf{substate}^{+}(\mathsf{source}(t'))]\}$

These two relations are the most precise values we can compute by static analysis, since in general only by actually executing the state machine is it possible to find out whether $\mathsf{source}(t'')$ will be active whenever $\mathsf{source}(t')$ is active.

The relations are not symmetric, i.e., if $t' \in \overline{\mathcal{B}}(t)$, generally it does not hold that $t \in \overline{\mathcal{B}}(t')$ and if $t' \in \mathcal{B}(t)$, generally it does not hold that $t \in \mathcal{B}(t')$. Moreover, the relations are independent of the transitions' effects: since the transitions to fire are selected before their effects, if any, are executed, potential effects are transparent for (both strict and non-strict) bondedness of transitions.

Now we are in the position to define metrics for (strictly and non-strictly) bonded relationships.

Definition 11 (Number of Bonded Transitions). *Given a transition t, the number of transitions* bonded *to t is*

$$\overline{NB}(t) = \#\overline{\mathcal{B}}(t)$$

Definition 12 (Number of Strictly Bonded Transitions). *Given a transition t, the number of transitions* strictly bonded *to t is*

$$NB(t) = \#\mathcal{B}(t)$$

These two metrics reflect the complexity of transitions that may be fired simultaneously, i.e., within one execution step.

In Fig. 6 the transition t_{CC} from `CrystalRoom` to `CrystalPedestal` is strictly bonded (and hence bonded) to each of the transitions triggered by the `investigate` event in the lower region: the transition t_{CW} from `Curious` to `Wander`, the transition t_{CD} from `Curious` to `DiscoverSecretTrapdoor` and the transition t_{II} from `Idle` to `Investigating`. Therefore $\overline{NB}(t_{CW}) = NB(t_{CW}) = 1$, $\overline{NB}(t_{CD}) = NB(t_{CD}) = 1$ and $\overline{NB}(t_{II}) = NB(t_{II}) = 1$. These numbers show that each of these transitions may be accompanied by a simultaneous transition in an orthogonal region, but that if a concurrent execution step takes place in the orthogonal region, it always progresses by triggering the same transition for each of t_{CW}, t_{CD} and t_{II}. (The metric is not precise enough to indicate that all three transitions are bonded to the same transition t_{CC})

On the other hand, the transitions t_{CW} and t_{CD} are both strictly bonded to t_{CC} since no transition starting from a substate of `Curious` exists. However the transition t_{II} is bonded to t_{CC} but not strictly bonded, since t_{CW} and t_{CD} are bonded to t_{CC} and their initial states are substates of `Idle`. Therefore, we have $\overline{NB}(t_{CC}) = 3$, $NB(t_{CC}) = 2$. These metrics show that the behavior in the region orthogonal to t_{CC} is much more complicated: There are at least two different transitions in the innermost relevant region that may each fire concurrently with t_{CC}, (because $NB(t_{CC}) = 2$), and there is one transition that may be inhibited by locally invisible transitions inside its source state (because $NB(t_{CC}) = \overline{NB}(t_{CC}) + 1$). When looking at bonded transitions, the measures NB and $\overline{NB}$ for the transitions bonded to a transition t are more revealing than the measures for t itself: By seeing the numbers $NB(t_{CC})$ and $\overline{NB}(t_{CC})$ when looking at t_{II} we are immediately alerted that there are other transitions that may prevent t_{II} from firing, even when the submachine inside `Idle` is hidden.

Competing States. There are also states which are, at least under certain conditions, not supposed to be active simultaneously. This is often implemented in state machines by adding `entry`, `do`, or `exit` actions to a state that change the value of some variable x, while there are states in other regions with incoming transitions where x is consulted; [11] discusses implementation of mutual exclusion using this technique in greater detail.

We have to take care of two facts:

1. the target of the transition may be a composite state, or a substate of some composite state, and therefore firing this transition may actually cause many other states to be active and their entry actions to be executed;
2. the source of the transition may be a composite state, or a substate of some composite state, and therefore firing this transition may actually cause many other states to be inactive and their exit actions to be executed.

While in case 1 we can precisely calculate the states to be activated and thus their entry actions, in case 2 it is in general not possible to calculate statically the precise "path" of state deactivation and thus the exit actions. We therefore make some approximation for this case.

We first define auxiliary functions for the entry and exit actions that are executed when composite state or their substates get active or inactive.

Definition 13 (Compound Entry Action). *Given a transition t, we define its* Compound Entry Action *as* $\mathsf{Entry}_{\mathcal{C}}(t) = \{\mathsf{entry}(s) \mid s \in Act(t)\}$.

Definition 14 (Compound Exit Action). *Given a transition t, we define its* Compound Exit Action *as* $\overline{\mathsf{Exit}_{\mathcal{C}}}(t) = \{\mathsf{exit}(s) \mid s \in Dct(t)\}$.

Note that $\overline{\mathsf{Exit}_{\mathcal{C}}}(t)$ also contains actions that are not executed, since $Dct(t)$ contains states that are not active when t is fired. Again, finding out which states are actually active when t is fired requires actually executing the state machine; static analysis does not suffice.

Definition 15 (Strict Compound Exit Action). *Given a transition t, we define its* Strict Compound Exit Action *as*

$$\mathsf{Exit}_{\mathcal{C}}(t) = \{\mathsf{exit}(S) \mid S \in \mathsf{substate}^*(\mathsf{Csr}(\mathsf{LCR}(\mathsf{source}(t), \mathsf{target}(s)))) \wedge \mathsf{target}(t) \in \mathsf{substate}^*(S)\}).$$

$\mathsf{Exit}_{\mathcal{C}}(t)$ *underestimates* the exit actions, since it only contains the exit actions of those states that are *definitely* deactivated when t is fired, i.e., all composite states containing $\mathsf{source}(t)$, contained in the region $\mathsf{LCR}(\mathsf{source}(t), \mathsf{target}(s))$.

Definition 16 (Modified Variables). *Given a transition t, the set of variables it* modifies, *$\overline{\mathcal{W}}(t)$ is the set of all variables written by* $\overline{\mathsf{Exit}_{\mathcal{C}}}(t) \cup \{\mathsf{effect}(t)\} \cup \mathsf{Entry}_{\mathcal{C}}(t)$.

Definition 17 (Strictly Modified Variables). *Given a transition t, the set of variables it* strictly modifies, *$\mathcal{W}(t)$ is the set of all variables written by* $\mathsf{Exit}_{\mathcal{C}}(t) \cup \{\mathsf{effect}(t)\} \cup \mathsf{Entry}_{\mathcal{C}}(t)$.

Definition 18 (Do-Modified Variables). *Given a state s, the set of variables it* do-modifies *is the set $\mathcal{W}(s)$ of all variables written in* $\mathtt{do}(s)$.

As stated in Sect. 2.1, in order to calculate our metrics, we ignore possible orders of execution of these actions, and instead simply assume a non-deterministic execution order.

Definition 19 (Read Variable). *Given a transition t, the set of variables it* reads *is $\mathcal{R}(t) = \{v \in \mathcal{V} \mid v$ is read by $\mathsf{guard}(t)\}$.*

Now we can define relations describing cross-region dependencies:

Definition 20 (Controlling). *Given a transition t_1 and a state s_2, we say s_2 is* (weakly) controlled *by t_1, and write $\overline{\mathsf{control}}(t_1, s_2)$, if there exists a transition t_2 with target s_2 such that t_1 modifies a variable read by t_2, i.e., if*

$$\exists t_2 \cdot \exists v \in \mathcal{V} \cdot [\mathsf{target}(t_2) = s_2 \wedge v \in \overline{\mathcal{W}}(t_1) \wedge v \in \mathcal{R}(t_2)].$$

Given two states s_1 and s_2, we say s_2 is (weakly) controlled *by s_1 if there exists a transition t_1 with target s_1 that controls s_2 or if the* $\mathtt{do}$ *activity of s_1 modifies a variable that is read by a transition leading into s_2, i.e., if*

$$\exists t_1, t_2 \cdot \exists v \in \mathcal{V} \cdot [\mathsf{target}(t_1) = s_1 \wedge \mathsf{target}(t_2) = s_2 \\ \wedge (v \in \overline{\mathcal{W}}(t_1) \vee v \in \mathcal{W}(s_1)) \wedge v \in \mathcal{R}(t_2)]$$

We write this as $\overline{\mathsf{control}}(s_1, s_2)$. If $v \in \mathcal{W}(t_1) \vee v \in \mathcal{W}(s_1)$ holds in the above formula instead of $v \in \overline{\mathcal{W}}(t_1) \vee v \in \mathcal{W}(s_1)$, then we say s_1 strictly controls *s_2 and write it as $\mathsf{control}(s_1, s_2)$.*

A state s_1 therefore controls a state s_2 if there exists a variable v such that a guard in at least one transition t leading into s_2 depends on v and either (1) a transition leading into s_1 modifies v in its compound entry action, its effects or its compound exit actions, or (2) the $\mathtt{do}$ activity of s_1 modifies v. This can also be expressed as

$$\Big(\bigcup_{\mathsf{target}(t_i)=s_1} \overline{\mathcal{W}}(t_i) \cup \mathcal{W}(s_1) \Big) \cap \mathcal{R}(t) \neq \emptyset$$

If s_1 controls s_2 it may only influence some paths leading into s_2, or it may control all transitions leading into s_2. The latter case is important if we want to ensure, e.g., mutual exclusion between states. We therefore define the notions of partial and total control:

Definition 21 (Partial Control). *Given two states s_1 and s_2, we say s_2 is* partially controlled *by s_1, and write $\overline{\mathsf{control}}^p(s_1, s_2)$, if s_2 is controlled by s_1 but there exists a transition t with target s_2 that is not controlled by s_1, i.e., if the predicates*

$$\exists t \cdot \Big[\mathsf{target}(t) = s_2 \wedge \Big(\bigcup_{\mathsf{target}(t_i)=s_1} \overline{\mathcal{W}}(t_i) \cup \mathcal{W}(s_1) \Big) \cap \mathcal{R}(t) \neq \emptyset\Big]$$

$$\exists t \cdot \Big[\mathsf{target}(t) = s_2 \wedge \Big(\bigcup_{\mathsf{target}(t_i)=s_1} \overline{\mathcal{W}}(t_i) \cup \mathcal{W}(s_1) \Big) \cap \mathcal{R}(t) = \emptyset\Big]$$

both hold. If additionally $\mathsf{control}(s_1, s_2)$ *holds, then we say* s_1 partially strictly controls s_2 *and notate it by* $\mathsf{control}^p(s_1, s_2)$.

Definition 22 (Total Control). *Given two states* s_1 *and* s_2*, we say* s_2 *is* totally controlled *by* s_1*, and write it as* $\overline{\mathsf{control}}^t(s_1, s_2)$*, if* s_1 *controls all transitions leading into* s_2*, more precisely, if* $\overline{\mathsf{control}}(s_1, s_2)$ *and*

$$\forall t_1, t_2 \cdot [\mathsf{target}(t_1) = s_1 \wedge \mathsf{target}(t_2) = s_2 \implies (\overline{\mathcal{W}}(t_1) \cup \mathcal{W}(s_1)) \cap \mathcal{R}(t_2) \neq \emptyset].$$

If $(\mathcal{W}(t_1) \cup \mathcal{W}(s_1)) \cap \mathcal{R}(t_2) \neq \emptyset$ *holds in this equation instead of* $(\overline{\mathcal{W}}(t_1) \cup \mathcal{W}(s_1)) \cap \mathcal{R}(t_2) \neq \emptyset$*, then we say* s_1 totally strictly controls s_2 *and write it as* $\mathsf{control}^t(s_1, s_2)$.

If s_1 totally controls s_2 we therefore have

$$\forall t \cdot \left[\mathsf{target}(t) = s_2 \implies \Big(\bigcup_{\mathsf{target}(t_i) = s_1} \overline{\mathcal{W}}(t_i) \cup \mathcal{W}(s_1)\Big) \cap \mathcal{R}(t) \neq \emptyset\right]$$

and if there are any transitions leading into s_2 this is a necessary and sufficient condition for total control.
Based on the controlling relationship, we now define the following metrics:

Definition 23 (Number of Partially Controlled States). *Given a state* s*, its* Number of Partially Controlled States *is*

$$NPC(s) = \#\{s' \mid \overline{\mathsf{control}}^p(s, s')\}$$

and its Number of Partially Strictly Controlled States *is*

$$NPSC(s) = \#\{s' \mid \mathsf{control}^p(s, s')\}$$

Definition 24 (Number of Totally Controlled States). *Given a state* s*, its* Number of Totally Controlled States *is*

$$NTC(s) = \#\{s' \mid \overline{\mathsf{control}}^t(s, s')\}$$

and its Number of Partially Strictly Controlled States *is*

$$NTSC(s) = \#\{s' \mid \mathsf{control}^t(s, s')\}$$

As indicated above, these metrics may be used to determine possible sources of concurrency errors: if $NTC(s) < NPC(s)$ the state machine contains a state s' whose reachability on some paths depends on s but some other transitions into s' do not depend on s. If s' is meant to be mutually exclusive to s this may indicate synchronization bugs. (Note that an analysis based solely on these metrics is not sufficiently precise to determine whether s and s' are synchronized or not since, e.g., predecessor states of s' may be synchronized with s and therefore prevent the unguarded transitions from being reached).

3.5 Applications

While the metrics presented in this paper represent only a rough estimate of the complexity caused by behavior depending on non-local properties, we believe that they could serve a useful purpose in alerting modelers to unexpected features of their state machines. In the following, we demonstrate the usefulness of our metrics by means of two simple extensions of the game, which contain several potential modeling mistakes that can be identified using our metrics, see Figs. 4 and 6.

Hidden Deactivation. Figure 4 contains several potential mistakes. The first one 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, and a developer focusing on this region might suppose that casting a spell always leads to a fight, and that fights are always terminated by either winning or escaping. However, the number of deactivating transitions for every state in the lower region is higher than the number of directly visible transitions, which clearly indicates that there are other ways to exit the states in this region than the locally visible ones. For example, in state `Fighting` there are two locally visible transitions (to `Spelling` and to the final state), but the number of deactivating transitions is 3. By looking at the metrics the developer is therefore immediately alerted to the existence of deactivating transitions operating in a "more global" manner.

Post-Domination. While a simple comparison of locally visible activations and deactivations with the metrics proposed in this paper is strong enough to point to some problems, this analysis is relatively indiscriminate and may capture non-local effects intended by the system's designers. By combining the metrics with flow-analysis techniques it becomes possible to identify more precisely situations which are likely to be incorrect.

For example, a slightly more sophisticated analysis of Fig. 4 shows that it is possible for the wizard to obtain arbitrarily large power ups: Taking into account only local transitions, state `PowerFighting` *post-dominates* [1] state `PowerSpelling`, i.e., each path from `PowerSpelling` to a final state goes through `PowerFighting`. This is an important property because states that release acquired resources or undo changes to variables have to post-dominate all resource acquisitions or variable changes for the state machine to be correct. In the example, the player obtains a boost of its `power` in state `PowerSpelling` which is undone in state `PowerFighting`.

Looking only at the lower state machine one might thus be led to assume that the exit action of `PowerFighting` will always be executed after state `PowerSpelling` has been entered. However, the measure D^{tr} of deactivating transitions for `PowerSpelling` has the value 3; since there are only two locally

visible outgoing transitions it is clear that it is, indeed, possible to exit this state by a non-local transition: If the player enter state `PowerSpelling` while in the `Ladder` room, she can take the transition `toLadder` to exit from `PowerSpelling` without decreasing the value of the `power` variable.

By combining the metrics presented in this paper with static analysis it would therefore be possible for tools to identify possible sources of errors resulting from non-local activations or deactivations.

"Stolen" Events. One of the pitfalls of the semantics of UML state machines is that if several transitions with the same trigger are enabled, and for two of them, say t_1 and t_2, it holds that $\mathsf{source}(t_1)$ is directly or recursively contained in $\mathsf{source}(t_2)$, then only t_1 is fired. This is a bit dangerous since when modeling on a higher level of abstraction (with states on a higher level in the hierarchy), the modeler may intend to have several transitions bonded, i.e., they should be fired at the same time, and this bondedness may easily get lost when one of the higher level states is later refined and a substate reacts to the same event. For example, in Fig. 6, since state `Curious` also reacts to event `investigate` (and one of the conditions `[TD]` and `[!TD]` is always true), the transition from `Idle` (which contains `Curious`) to `Investigating` will not be fired when `Curious` is active. Using our metrics, this mistake can be easily detected: Let t be the transition from `CrystalRoom` to `CrystalPedestal`, then $\overline{NB}(t) \neq NB(t)$, and the modeler can be alerted to double check this transition and those bonded to it.

Mutual Exclusion. Another important application of our metrics is the detection or validation of mutual exclusions of states. In general, mutual exclusion is in UML state machines often hard to model and to comprehend, since the exclusion logic is "hidden" in several model elements, scattered in several regions [11]. A mechanism to make mutual exclusions in a state machine "visible" is therefore desirable.

Our metrics provide a simple means to make hidden mutual exclusions visible. For two states s_1 and s_2, we say s_1 *excludes* s_2 if s_1 strictly controls s_2, and there do not exist transitions $t_1, t_2, \mathsf{target}(t_1) = s_1, \mathsf{target}(t_2) = s_2$, such that after t_1 has been fired (and all exit actions, t_1's effect, and all entry actions have been executed), the guard of t_2 is satisfiable. If s_1 excludes s_2 and s_2 excludes s_1, then s_1 and s_2 mutually exclude each other from being active. For example, in Fig. 6, the states `CrystalRoom` and `Spelling` mutually exclude each other, since the condition for `CrystalRoom` to be active (via any transition) is `c==0`, and its entry action then increases the value of umlc, setting it to 1, thus making it impossible for the guard of any of the transitions leading to `Spell` satisfiable; and in the same manner, entering `Spell` also prevents `CrystalRoom` from getting active. This way, mistakes of the modeler failing to constrain any of transitions leading to these states are easy to detect by the measures for partial and total control as described on p. 20.

4 Related Work

Complexity metrics of state machines have been recognized as useful indicators [9]. Hierarchical states are considered in [3,4]. Strictly speaking, these approaches do not consider UML state machines, but rather State Transition Systems (STSs). The syntax of STSs is considerably simpler than that of UML state machines, and their semantics do not contain cross-region effects, such as bonded transitions, stolen transitions, or a transition originating from one region deactivating states in another. In comparison, our approach provides a set of metrics for UML state machines, where it is much more involved to determine the model elements that may activate or deactivate certain states. Moreover, it is also clearer which elements are actually responsible for the complexity.

Cyclomatic complexity [8] is a very widely-used metric for 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.

Our previous paper [12] was the first one to study remote activation and deactivation of states. The current paper extends [12] by the metrics regarding cross-region dependencies.

The determination of the transitions activating or deactivating a certain state is also an essential technique for weaving aspect-oriented state machines [14].

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 metrics to reflect the complexity of transitions leading to or leaving composite states, as well as the complexity caused by cross-region dependencies. 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, and may alert them to potential modeling mistakes.

Based on this work, we plan to define more precise metrics which also take into account, e.g., the partial order of the execution of entry and exit actions when composite states are activated or deactivated. Metrics on their own can only provide relatively coarse indications of problems in state machines, and tools based solely on metrics will probably often report possible errors when structural properties of the state machine are used by designers to ensure invariants of the model. Therefore we also intend to pursue the integration of the measures presented in this paper with stronger structural analysis techniques for state machines. Finally, we plan to validate our metrics in more realistic models, as well as to implement support for the metrics in modeling tools.

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

References

1. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Kenneth Zadeck, F.: An efficient method of computing static single assignment form. In: Conference Record of the 16th Annual ACM Symposium Principles of Programming Languages (POPL 1989), pp. 25–35. ACM Press (1989)
2. Drusinsky, D.: Modeling and Verification Using UML Statecharts. Elsevier, Amsterdam (2006)
3. Guo, L., Sangiovanni-Vincentelli, A.L., Pinto, A.: A complexity metric for concurrent finite state machine based embedded software. In: 8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013), pp. 189–195. IEEE (2013)
4. Hall, M.: Complexity metrics for hierarchical state machines. In: Cohen, M.B., Ó Cinnéide, M. (eds.) SSBSE 2011. LNCS, vol. 6956, pp. 76–81. Springer, Heidelberg (2011)
5. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. Specification, OMG (2011). http://www.omg.org/spec/UML/2.4.1/Superstructure
6. Schmidt, C.D.: Model-Driven Engineering. IEEE Comput. **39**(2), 25–31 (2006)
7. Vallecillo, A., Koch, N., Cachero, C., Comai, S., Fraternali, P., Garrigó, I., Gómes, J., Kappel, G., Knapp, A., Matera, M., Meliá, S., Moreno, N., Pröll, B., Reiter, T., Retschitzegger, W., Rivera, J.E., Schauerhuber, A., Schwinger, W., Wimmer, M., Zhang, G.: MDWEnet: A practical approach to achieving interoperability of model-driven web engineering methods. In: Koch, N., Vallecillo, A., Houben, G.-J. (eds.) Proceedings of the 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007), vol. 261 of CEUR-WS (2007)
8. http://en.wikipedia.org/wiki/Cyclomatic_complexity. Accessed on 2014–04-30
9. http://code.google.com/p/umple/wiki/MasuringStateMachineComplexity. Accessed on 2014–04-30
10. Zhang, G.: Aspect-Oriented State Machines. Ph.D thesis, Ludwig-Maximilians-Universität München (2010)
11. Zhang, G.: Aspect-oriented modeling of mutual exclusion in UML state machines. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 162–177. Springer, Heidelberg (2012)
12. Zhang, G., Hölzl, M.: A set of metrics for states and transitions in UML state machines. In: Proceedings of the 6th International Workshop on Behaviour Modeling-Foundations and Applications (BM-FA 2014). ACM, New York (2014)
13. Zhang, G., Hölzl, M.: HiLA: high-level aspects for UML state machines. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 104–118. Springer, Heidelberg (2010)
14. Zhang, G., Hölzl, M.M.: Weaving semantic aspects in HiLA. In: Hirschfeld, R., Tanter, É., Sullivan, K.J., Gabriel, R.P. (eds.) Proceedings of the 11th International Conference on Aspect-Oriented Software Development, (AOSD 2012), pp. 263–274. ACM Press (2012)