A Method for Developing UML State Machines
in case we need a subtitle put here

Christine Choppy · Gianna Reggio

Received: date / Accepted: date

Abstract While UML is one of the most used notation for describing a system to build, there are many issues to tackle to develop an appropriate UML model. In this work, we concentrate on the development of state machines describing the behaviour of a class. We show how relevant information can be searched for in the textual description and organized to obtain a behaviour description from which the state machine can be derived. Our method is quite systematic and should help to prevent common mistakes in building state machines.

Keywords UML · State Machine · Modelling method

1 Introduction

The UML [8] is nowadays one of the most known and used visual notation for presenting the requirements on a system to build. The different diagrams parts of a UML model may describe different aspects of the system to build; e.g., a use case diagram may summarize the required functionalities, and a sequence diagram may present some required interactions between that system and its users. The state machines are the UML diagrams that allow to model the behaviour of entities of a specific type, and they may also play a relevant role in many different kinds of models, e.g., those corresponding to the requirement and to the design of a software system, or those describing a business or a service in a SOA.

Thus to be able to build the UML state machine correctly modelling the intended behaviour of given entities is truly an important task any time we want to model using the UML. ??? state machines in conceptual modelling Furthermore, in many cases the dimensions (i.e., number of states and of transitions) of the state machine to be produced may be relevant, and thus it is of the paramount importance to have a method to lead their development, ease this activity, and help to avoid the most common mistakes.

Out of our more than decennial experience in using the UML state machines and in teaching them in software engineering courses we have seen that there are various common errors made by people developing them:
– drawing transitions without triggers, forgetting that in a state machine a transition represents a reaction to a trigger (usually the transitions are labelled with a UML action, instead of an event);
– guards and effects referring to elements not defined in the context class of the state machine;
– too many states, i.e., different states where the reactions to the events are exactly the same.

The guidelines of our method try to help the designer to avoid these common mistakes.

To start with, state observers and events are found by analysing a textual description of a given UML class. The behaviour of this class is identified by looking for the invariant properties of state observers and for the reactions to events. These informations are then used to build the state machine of the class considered.

An overview of our method is given in Sect. 2. Then Sect. 3 and Sect. 4 are respectively devoted to describe in detail how to find the behaviour of a class and how
to build the state machine. This is illustrated on the example of the management of a book copy in a library, whereas other applications of the method are reported in Appendix 5. We conclude and discuss related works in Sect. 5.

2 Overview of the method

Our goal is to propose a method to build a state machine associated with a class while developing a UML model, at a stage where there is already a first version of a class diagram. The various steps required by our method are presented in Fig. 1 by means of a UML activity diagram.

We assume there is a UML class diagram including a class CL. CL may have already some attributes and operations, and there may be associations linking it to other classes in the diagram. Moreover, we have a textual description of the behaviour of the instances of the class CL, and we want to design a state machine modelling the behaviour of the instances of CL. Notice that also a set of use case scenarios expressed using the natural language may be used as description. 

The aim of the first tasks is to identify the behaviour of CL. The text is examined and syntactically analysed to look for state observers and events related with CL. For each state observer, an invariant condition should be expressed, and conditions regarding the initial state and, if any, the final state, are expressed. For each event, the reaction of instances of CL under some condition is expressed as an UML action, see [8, Sect. 11]. The resulting invariants and actions are then screened with the help of a checklist (and possibly corrected). Of course, while invariants and reactions are found and expressed, the need to update the state observers and events may arise (as represented by the loop in Fig. 1).

The tasks in the lower part of Fig. 1 are devoted to building the state machine. This starts with identifying the relevant states determined by the values of a subset of the state observers. There are several possibilities that are related with a choice for the use of a state observer in this task, and the class diagram should be updated accordingly. Once the states are determined, finding the transitions is quite straightforward.

In the following, our method is explained in detail and illustrated on a library case study presented below. Example Copy: The textual description. The case study is about giving a conceptual model of a library, as a starting point to develop a software to assist the librarians. A sketchy class diagram is shown in Fig. 2. The behaviour of the (instances of the) class Copy is described as follows.

The typical life of a copy of a book in the library is as follows. The copy may be lent to a user of the library, and after brought back. While the copy is on loan, it may be booked by another user, and when it will be brought back only the Booker may get it on loan. Multiple bookings are not allowed. The librarians may put a red tag on the copy, meaning that the book cannot be lent, and it will be in this situation till the red tag will be removed. Obviously, it is not possible to put the red tag while the copy is on loan.

3 Find the behaviour

3.1 Find State Observers and Events

A state observer models a quantity, represented by the values of some type, that can be observed in each moment during the lives of the instances of CL. A state observer is characterized by a name and a type provided by the class diagram, and it is denoted by stateO: T.

We assume that there is a special boolean state observer identifying the final situations, final: Boolean i.e., when the activity (and the life) of the objects of class CL is ended. It will help to find the final states of the state machine to be built.
An event models something to which the instances of CL react, and that will be generated from their environment. More precisely, an event for an instance of CL may be:

a message event the reception of a message (sent by other objects), i.e., of a call of one of the operations of CL. A message event is characterized by a name and some parameters, typed with the types provided by the class diagram and it is denoted by event(T₁,...,Tₖ).

a time event the reaching of a given (absolute) time or the elapsing of a given amount of time (absolute and relative time events [8, page 451]), the time elapsing is counted as the time a given condition is true and nothing happen. The time events are denoted by

at time (absolute time) and

after time that cond holds and nothing happens (relative time).

where cond is an OCL boolean expression well-formed w.r.t. the context given by CL, where the state observers are considered as attributes of the class CL and time is a constant in the UML is not very clear if it must be a constant or may be an expression built by using the attributes denoting a time.

a change event the changing of the truth of a condition (change events [8, page 450]). The change events are denoted by when cond changes, where cond is an OCL boolean expression built using the state observers.

The first task of our method requires to find the initial lists of state observers and events. Any attribute and any association attached to the class CL already present in the class diagram are considered by default as state observers; similarly, any operation of the class without return type is considered by default as an event.

and the operations with a return type they may be considered as state observers when we will introduce parameterized state observers.

To find the state observers and the events we assume to have a textual description of the behaviour of the instances of the class CL and to syntactically analyse it looking for:

- sentences qualifying the instances of CL (e.g., “it has ...”, “they are ...”, “... is associated with them”), adjectives related to them, and nouns related to parts or components of them; they will provide the candidate state observers.

- verbal phrases where the verb is in active form and the instances of CL are the object complement or the verb is in passive form and the instances of CL are the subject; they will provide the candidate events.

- sentences about absolute and relative time, e.g., “at ....”, “after ...”.

- sentences about changes, e.g., “when ... becomes true/holds”, “when ... will be false/will not hold”.

Appropriate names and typing should be chosen for the identified state observers and message events.

Example Copy: Find events and state observers

The analysis of the informal description of the behaviour of the instances of the Copy class results in the initial lists of events and state observers that are given in Fig. 3. The first two state observers code and copyOf are by default, since they correspond respectively to an attribute and to an association already present on the class diagram of Fig. 2. The implicit state observer final is not reported in the table.

3.2 Find invariants and reactions

To find the behaviour of the instances of the class under consideration, CL, one should answer several questions about the state observers and the events and fill the following forms.

Invariants

For each state observer, find the condition concerning it that must hold in any moment of the life of the instances of CL (invariant).

<table>
<thead>
<tr>
<th>state observer</th>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>stateO₁</td>
<td>inv₁</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>stateOₖ</td>
<td>invₖ</td>
</tr>
</tbody>
</table>

where each invᵢ is an OCL boolean expression well-formed w.r.t. the context given by CL, where the state observers are considered as attributes of the class CL, in which stateOᵢ must appear.

All these conditions, as well as those used to fill the other forms, should be written respecting the OCL rules. For legibility and traceability sake, it is better at this stage not to try to simplify the expressions.

<table>
<thead>
<tr>
<th>events</th>
<th>state observers</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertLib(string)</td>
<td>code: string</td>
</tr>
<tr>
<td>lend(User)</td>
<td>copyOf: Book</td>
</tr>
<tr>
<td>bringBack()</td>
<td>onLoan: Boolean</td>
</tr>
<tr>
<td>book(User)</td>
<td>booked: Boolean</td>
</tr>
<tr>
<td>putTag()</td>
<td>booker: User[0...1]</td>
</tr>
<tr>
<td>removeTag()</td>
<td>tagged: Boolean</td>
</tr>
</tbody>
</table>

Fig. 3 Copy example: initial list of events and state observers.
Reactions  For each event $evt$ find the pairs $(cond_i, act_i)$ such that: if an instance of class $CL$ is in a state where the values returned by the state observers satisfy $cond_i$, then this instance will react to $evt$ and the reaction will be to execute $act_i$. Obviously, all the conditions should be of the form "not final and ...", thus for readability the part “not final” will be omitted. In the forms we have also to consider the reaction to the special event $created$; recall that in the UML state machine the only transition leaving the initial state is implicitly triggered by it (but visually it is shown as a transition without a trigger). In this case the condition is assumed to be false.

Summarize all the pairs $(cond_i, act_i)$ by filling the following form.

<table>
<thead>
<tr>
<th>event</th>
<th>cond</th>
<th>reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$evt$</td>
<td>$cond_1(X_1, \ldots, X_k)$</td>
<td>$act_1(X_1, \ldots, X_k)$</td>
</tr>
<tr>
<td></td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td></td>
<td>$cond_h(X_1, \ldots, X_k)$</td>
<td>$act_h(X_1, \ldots, X_k)$</td>
</tr>
</tbody>
</table>

- if $evt$ is a time or a change event, then $k = 0$, otherwise $X_1, \ldots, X_k$ are the formal parameters of the event, i.e., $evt = event(X_1, \ldots, X_k)$;
- each $cond_i(X_1, \ldots, X_k)$ is an OCL boolean expression built using the state observers, where only the parameters $X_1, \ldots, X_k$ may appear as free-variables.
- each $act_i(X_1, \ldots, X_k)$ is an action, where only the parameters $X_1, \ldots, X_k$ may appear as free-variables, built using the standard control flow combinators in the form provided by the UML actions (“loop”, “alt” and “;”) and the basic actions, which are state observer modifications (written $stateO := exp$), call of other objects, operations, and creation and destruction of other objects.

There may be various pairs $(cond, act)$ for the same event, and there is no need for the conditions to be non-overlapping nor to cover all the possible cases. Indeed, the “run-to-completion-semantics” of UML state machines specifies that the reaction to an event may be nondeterministic (in the case there are different transitions with that event whose guards are satisfied), and that if there is no transition with that event whose guard is satisfied, then there will be no reaction and the event will be lost. We add some special rows to define when an event may be deferred, and to keep in mind the cases in which an event is lost.

<table>
<thead>
<tr>
<th>event</th>
<th>cond</th>
<th>reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$evt$</td>
<td>$cond$</td>
<td>may be deferred</td>
</tr>
<tr>
<td>$evt$</td>
<td>not $(or_{i=1}^h cond_i)$</td>
<td>ignored</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state observer</th>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td></td>
</tr>
<tr>
<td>copyOf</td>
<td></td>
</tr>
<tr>
<td>onLoan</td>
<td>onLoan =&gt; not tagged</td>
</tr>
<tr>
<td>booked</td>
<td>booked =&gt; (booker &lt;&gt; undef and not tagged)</td>
</tr>
<tr>
<td>booker</td>
<td>booker = undef &lt;&gt; not booked</td>
</tr>
<tr>
<td>tagged</td>
<td>tagged =&gt; (not onLoan and not booked)</td>
</tr>
<tr>
<td>final</td>
<td>not final</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>event</th>
<th>cond</th>
<th>reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>created</td>
<td>$onLoan$ = false; $booked$ = false; $tagged$ = false</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4  Copy example: Invariants and reactions (first version)

When filling the cond cell, for each existing state observer investigate whether its value may influence the fact that the instances of CL may react to that event; similarly when filling the reaction examine for each state observer if the reaction to that event may influence its value.

While filling the forms about the behaviour, very frequently we may discover that we need other state observers or events, or that the type of a state observer or the arguments of an event must be changed, or also that a state observer or an event is useless.

Example Copy: Find the behaviour Fig. 4 shows the initial versions of the invariants and of condition/reactions for Copy. Notice that we have also an invariant for the implicit state observer final. Since the reaction to the $created$ special event does not assign a value to code and copyOf, i.e., they are left undefined\(^3\), this leads us to modify the multiplicity of the the attribute code and of association copyOf from 1 to 0..1, and consequently to modify the corresponding state observers to allow them to be undefined. This change leads also to add some properties on code and on copyOf. The modified artifacts are reported in Fig. 5.

Then we can fill the forms about conditions and reactions reported in Fig. 6. When filling the row for putTag, we ask to ourselves if the values of booked and booker should be considered. This point to an under-specified aspect in the behaviour of Copy: could the tag be put when the book is not on loan but booked? In this case we decide no. Note how the checks required by our method led to unveil some aspects of the behaviour of the modelled entities that were not considered in the original description.

Checklist on the filled forms At this point we can perform some checks on the forms filled up to now

\(^3\) $e = undef$ is a shortcut for the OCL expression $e.oclIsUndefined()$.\n
that help discover quite early problematic aspects of the behaviour of the instances of CL, or of our understanding.

- Check if the reaction of the special event created gives a value to all the state observers; in case of a negative answer be aware that some state observers may be undefined will be undefined.

- Each pair of invariants cannot contradict each other.
- For each triple (evt, cond, react), cond cannot contradict an invariant.
- Each invariant inv should be respected by each triple (evt, cond, act); precisely, we have to check that if the values of the state observers satisfies inv and cond, then after the execution of act the values of the state observers will be such that inv is satisfied. Notice, that we do not require a formal proof, but just an informal examination of the reactions. Usually, since the reactions are quite simple this task is not very complicated, whereas it allows to detect many problems both in the behaviour of the instances of CL and in its modelling by the UML.
- Each invariant inv should be respected by each condition used in a time or change event, precisely for each “after time that cond holds and nothing happens” and each “when cond changes” inv and cond cannot contradict each other.
- Each state observer must be read at least once (i.e., it must appear at least in one condition or in one expression part of a reaction) and should be written at least once (i.e., it must appear at least once on the left side of an assignment part of a reaction).
- Each event evt should have at least one triple (evt, cond, react) and at least a non trivial condition and at least a non-null reaction. Notice that here we are using the word “should”, since there are limit cases where this requirement is not satisfied (e.g., a reset event that can happen in any possible situation, or a null command that will have no reaction in any possible situation). In these cases a second thought should be given to this specific event, to be sure about the related behaviour.
- Each argument of an event must appear either in a condition or in a reaction.

Example Copy: Checking the behaviour The results of the check list on the example of the Copy class are as follows.

The state observer copyOf is neither read nor written, whereas code is only read to check that it is undefined, but once it gets a value, then it will be never read again. This observation leads us to consider that copyOf and code are not relevant at all in the behaviour of Copy. We have thus discovered that there is a problem in the definition of the behaviour of the copies of

---

4 That is, the conjunction of the two conditions must be satisfiable.
the books. It seems that neither the code nor the information about the associated book are needed. The solution of this point is to extend the informal description of the copy behaviour by considering that a copy may be queried to discover what is the book of which it is a copy and its code; otherwise the two state observers must be dropped. In this paper we choose the second case.

Now, we modify the various forms by dropping the two state observers code and copyOf. However, now we cannot express the condition that the event insertLib cannot happen if the copy is already in the library; thus we add a new boolean state observer inLib. At this point the various artifacts related to our example are shown in Fig. 7.

4 Build the state machine

4.1 Find the states

Once the class behaviour was identified through the answers to the questions raised by our method, and after the result was checked, all the information needed to build a state machine modelling that behaviour of the instances of CL is available. To be more precise many different semantically equivalent state machines may be produced. Indeed, for each UML state machine there exist many others semantically equivalent ones differing by the number of states, and clearly in some cases also the attributes of the context class\(^5\) may vary.

We try to illustrate this point with a very simple example.

In Fig. 8 we have different equivalent state machines together with the corresponding context class. Among the various possible choices there are those surely criticizing since they exploit the graphical expressive power of the state machine in a wrong way (e.g., suggesting that similar situations are different or that different situations are the same), whereas others are acceptable but highlight different aspects of the behaviour of the instances of the context class.

Any state machine can be transformed into an equivalent one having one state plus the initial (pseudo) state; Fig. 8 A) shows one example of those “minimalistic” state machines. Obviously, in general this choice is not recommended, since it does not exploit the visual expressive power of state machines.

There is a criteria to judge if there are too many states: there cannot be two states S1 and S2 such that

\(^5\) UML terminology: if a state machine defines the behaviour of the instances of class CL, then CL is called the context class of the state machine.
and by the two states Ready and ToSet. Partial redundancy is acceptable, and corresponds to cases where the states represent some information that can be computed using the value of some attributes. For example, Fig. 8 D) shows an acceptable case: the fact that the value of the attribute \( a \) is greater than 0 is also modelled by the states ReadyA>0 and ReadyA≤0.

The case of Fig. 8 C) shows a choice of states without any redundancy w.r.t. the attributes. Both cases C) and D) are acceptable, they differ on the emphasis put by the modeller on the fact that attribute \( a \) is greater than 0.

Thus it is not possible to propose a method that finds the “unique” or the “best” state machine to model the found behaviour. Our method will help to build a state machine without too many states and unacceptable redundancy, whereas there may be some explicitly wanted redundancy.

Our method helps to determine the states by using the state observers typed using data types with a finite number of elements (i.e., boolean and enumeration with a small number of values, more than 10 should be considered as infinite).

The first task is to find these “discrete” state observers, let DSO be the set of them (it must include final). Then, we have to examine the state observers in DSO to detect if there are any derived ones. When a state observer is derived from others already in DSO, it should be removed from DSO. If, instead, it is derived from the other state observers, it should be decided whether to eliminate it, avoiding any redundancy or to keep it, having more states, and emphasizing the condition modelled by it. Moreover, if we choose to have some condition visually presented by the states, then some more derived state observers may have to be added.

We have now to check whether the part concerning the state observers in DSO may be singled out in each of the conditions and the actions filling the various forms. Check if the following condition holds:

- if for each triples \((\text{evt}, \text{cond}, \text{act})\), the condition \( \text{cond} \) can be rewritten into an equivalent one having form \( \text{cond}' | \text{DSO} \) and \( \text{cond}' \) where \( \text{cond}' | \text{DSO} \) is built using \text{and} , or , not and equation between state observers in DSO and constants, and the state observers in DSO do not appear in \( \text{cond}' \), and the action \( \text{act} \) can be rewritten in an equivalent one of the form \( \text{act}' | \text{DSO} ; \text{act}' \), where \( \text{act}' | \text{DSO} \) is a sequence of assignments of constants to state observers in DSO and the state observers in DSO do not appear in \( \text{act}' \).
- Only the state observers in DSO may appear in the conditions part of the time events, and in the conditions associated with the may be deferred reactions.

Build a table having as columns the state observers in DSO and as rows possible tuples of values returned by these state observers. Notice that if a state observer in DSO is partial you should consider also the case when it is undefined, thus among the possible values used to fill the form consider also \text{undef}.

Delete the non admissible rows, i.e., those which are in contradiction with some of the invariant properties. Each remaining tuple will identify a state. The case of Fig. 8 E) unacceptable redundancy shows an acceptable case: the fact that the

Fig. 8 Semantically equivalent state machines
should be in the form of a sentence qualifying a moment in the life of the instances of CL.

Once the states are determined, add the initial state that must be always present and unique.

The implicit state observer final will then determine the final states, they will be all those where its value is true (none if a constraint requires final to be always false).

**Example Copy: Find the states** In our example, we have one derived state observer, precisely booked (booked <=> booker <> undef). Since we think that it is relevant to visually represent the situation of being booked, we consider it while finding the states. Moreover, the only non discrete state observer is booker, that thus will be not considered when building the states; and there are no final states since final is always false:

<table>
<thead>
<tr>
<th>inLib</th>
<th>onLoan</th>
<th>booked</th>
<th>tagged</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>impossible</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>OnLoanBooked</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>impossible</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>OnLoanNotBooked</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>impossible</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>Booked</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>Tagged</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>impossible</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>impossible</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>impossible</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>impossible</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>NotInLibrary</td>
</tr>
</tbody>
</table>

...Fig. 9 Example Copy: finding the states...

servers in DSO satisfy cond_{DSO} and chg, there is a transition leaving S labelled by when not chg [cond']/act';
whereas if the state satisfies cond_{DSO} and not chg, the transition label will be when chg [cond']/act'.

In all the cases the target state will be the one corresponding to the set of values of the state observers in DSO determined by act_{DSO} (the state observers not modified by act_{DSO} have the same value as in S).

The triple (created, true, act) will determine the unique transition leaving the initial state.

Each triple (evt, cond, may be deferred) determines that the event evt should be deferred in all states whose corresponding tuple of values satisfy cond.

It is possible to list some restrictions on the filling of the table of the conditions-reactions guaranteeing that the built state machine does not have unacceptable redundancy, but, unfortunately, they are quite complex (roughly: if the types of the state observers in DSO have useless values, i.e., never tested by a condition, such values will contribute to build equivalent copies of states). Thus, it is much more simple to check for the presence of states from which the same transitions start and to collapse them in the produced state machine.

**Example Copy: Find the transitions** Using the tables in Fig. 7 and in Fig. 9 we can build the state machine modelling the behaviour of the instances of Copy, shown in Fig. 10. In this case there are no different states from whom the same transitions start.

---

6 For this reasons we do not report its values in the table.
On the process, the initial class diagram is updated. The tasks are adequately performed, and we show the update to each event. We provide a checklist to insure that the conditions to be satisfied in the initial state (and possibly the final state), as well as events and the reacting state observers with their invariant property, and be used to build the state machine. This involves finding state observers with their invariant property, and the conditions to be satisfied in the initial state (and possibly the final state), as well as events and the reaction to each event. We provide a checklist to insure that the tasks are adequately performed, and we show the advantages/drawbacks when choosing between several possibilities concerning the states of the state machine. On the process, the initial class diagram is updated.

As usually when applying a formal (or just a rigorous) method for modelling/specifying some entities, the proposed method helps to fully understand the specified class, and to detect problematic or obscure aspects of its behaviour.

While there are methods for developing UML models (that is when developing the different diagrams, etc.), we are not aware of much work concentrating on the UML state machine development for a given class behaviour.

In [5], five approaches using tabular notations for state machines are reviewed. Obviously the idea to find states from the set of conditions to be fulfilled has been used in different works, as, e.g., in [7] for LOTOS specifications, or [4] in the SCR tabular notation where condition tables, event tables and mode transition tables are used. Let us note that other approaches, like in [3], aim at synthesizing from sequence charts or paths of executions.

In [2] various properties of state observers and elementary interactions of a whole system are filled and used to derive its algebraic specifications. In [1], the textual description is used to find state observers and events of a whole system, as well as some information on its structure, then the properties of state observers and events are filled and used to derive a structured coloured Petri net specification of the system. Our method instead aims at deriving a UML state machine, it is simpler since it focuses on the behaviour of a given UML class, on the invariant properties of state observers and reactions to events.

As mentioned in the introduction, for simplicity sake, we chose for this version of the method to concentrate on the tasks to develop state machines with their “basic” features. We think it can be extended to take into account other features (e.g., hierarchical states, etc.), and we plan to extend the method to cover all relevant features.

The application of the proposed method, as presented in this paper, requires a little paper work, writing the various forms, filling them, perhaps modifying them, and so on. However, it is quite easy to develop a wizard to guide the application of the method, which proposes the various forms to be filled to the user, and performs the automatizable checks of Sect. 3.2. Moreover, such tool may take as input the XMI version of the initial class diagram, and at the end of the procedure return the XMI of the updated class diagram and of the built state machine.

As described in Sect. 3.3, the proposed method considers the UML state machine development for a given class. In this case, the state observers are related to the state machine through their invariant property, and be used to determine the states. This involves finding state observers with their invariant property, and the conditions to be satisfied in the initial state (and possibly the final state), as well as events and the reaction to each event. We provide a checklist to insures that the tasks are adequately performed, and we show the advantages/drawbacks when choosing between several possibilities concerning the states of the state machine. On the process, the initial class diagram is updated.

As usually when applying a formal (or just a rigorous) method for modelling/specifying some entities, the proposed method helps to fully understand the specified class, and to detect problematic or obscure aspects of its behaviour.

In [5], five approaches using tabular notations for state machines are reviewed. Obviously the idea to find states from the set of conditions to be fulfilled has been used in different works, as, e.g., in [7] for LOTOS specifications, or [4] in the SCR tabular notation where condition tables, event tables and mode transition tables are used. Let us note that other approaches, like in [3], aim at synthesizing from sequence charts or paths of executions.

In [2] various properties of state observers and elementary interactions of a whole system are filled and used to derive its algebraic specifications. In [1], the textual description is used to find state observers and events of a whole system, as well as some information on its structure, then the properties of state observers and events are filled and used to derive a structured coloured Petri net specification of the system. Our method instead aims at deriving a UML state machine, it is simpler since it focuses on the behaviour of a given UML class, on the invariant properties of state observers and reactions to events.

As mentioned in the introduction, for simplicity sake, we chose for this version of the method to concentrate on the tasks to develop state machines with their “basic” features. We think it can be extended to take into account other features (e.g., hierarchical states, etc.), and we plan to extend the method to cover all relevant features.

The application of the proposed method, as presented in this paper, requires a little paper work, writing the various forms, filling them, perhaps modifying them, and so on. However, it is quite easy to develop a wizard to guide the application of the method, which proposes the various forms to be filled to the user, and performs the automatizable checks of Sect. 3.2. Moreover, such tool may take as input the XMI version of the initial class diagram, and at the end of the procedure return the XMI of the updated class diagram and of the built state machine.

As are planning to do an experimental investigation on how the proposed method helps to avoid the most common mistakes in producing the state machines, in the next academic year with the students of the software engineering course.
References


to extend our method to cope also with a form of constructors. Constructors must be defined by means of a stereotype. We plan that the state observer

\[
\begin{array}{|c|c|}
\hline
\text{Basket} & \text{RFID label} \\
\text{code: string } [0..1] & \text{name: string} \\
\text{capacity: float } [0..1] & \text{day: Date} \text{ Fruit} \\
\hline
\end{array}
\]

Fig. 12 Basket example: initial class diagram

\[\begin{array}{|c|c|}
\hline
\text{events} & \text{state observers} \\
\text{mark(RFID label)} & \text{code: string } [0..1] \\
\text{fill(float)} & \text{capacity: float } [0..1] \\
\text{moveToPack} & \text{rfid: RFID label } [0..1] \\
\text{empty()} & \text{inDept: Boolean} \\
\text{backToDept()} & \text{cont: float} \\
\text{atPack} & \text{atPack: Boolean} \\
\hline
\end{array}\]

Fig. 13 Basket example: initial lists of events and state observers

Appendix: Applications of the method

The RFID basket case

The case study is about a system for partly automating the procedure of collecting the fruit in a farm, by equipping the baskets used in the collection with a RFID label.

At a certain point we have a class diagram, shown in Fig. 12, and we want to produce a state machine modelling the behaviour of the objects of class Basket. The life of those objects is as follows.

At the beginning the basket is in the deposit of the farm. Then a farmer may take it, and mark with an RFID label the date, the fruit that it contains, and its name. Then, the basket will be filled with fruits, and afterwards it will be moved to the packaging center, where it will be emptied and the RFID label removed. After that the empty basket will be put back again in the deposit.

We apply our method and find the initial list of events and state observers filling the form presented in Fig. 13. Then we fill the invariant, initially and termination conditions, and reaction forms, shown in Fig. 14.

The required checks lead us to discover that the state observer code is fully useless, and thus it will be dropped. Moreover, capacity is never written, and this fact points out that an event is lacking, precisely acquire corresponding to the fact that the basket is acquired by the farm and gets assigned a capacity.\(^7\)

The required checks also leads us to discover that the state observer rfid is never read, and that the value of cont is never fully exploited. This points out to the fact that the reaction to empty() is incomplete, the value of cont and the data contained in the rfid should be sent to the device that may process them. We assume that empty has now as argument the reference to the device to which those data are sent. The list of state observers and events and the various forms are then modified as shown in Fig. 15. Notice that in the picture, the conditions have been expressed in a minimalistic way; i.e., they have been simplified by using the invariants.

Then we can find the states of the state machine. The discrete state observers are inUse, inDept, atPack, and in Fig. 16 there is the table of their possible values for determining the states of the state machine.

The modified class diagram and the built state machine are then shown in Fig. 17.

\(^7\) This strange solution is due to the fact that in the UML there are no built in constructors for the objects of a class, the constructors must be defined by means of a stereotype. We plan to extend our method to cope also with a form of constructors.
<table>
<thead>
<tr>
<th>event</th>
<th>cond</th>
<th>reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>created</td>
<td></td>
<td>inDept := false; cont := 0; atPack := false inUse := false</td>
</tr>
<tr>
<td>acquire(C)</td>
<td>not inUse</td>
<td>capacity := C; inUse := true; inDept := true</td>
</tr>
<tr>
<td>mark(R)</td>
<td>inDept</td>
<td>rfid := R; inDept := false</td>
</tr>
<tr>
<td>fill(Q)</td>
<td>inUse and Q + cont &lt;= capacity and not inDept and not atPack</td>
<td>cont := cont + Q</td>
</tr>
<tr>
<td>empty(S)</td>
<td>not inUse or Q + cont &gt; capacity or inDept or atPack</td>
<td>ignored</td>
</tr>
<tr>
<td>moveToPack()</td>
<td>not atPack</td>
<td>ignored</td>
</tr>
<tr>
<td>backToDept()</td>
<td>cont := 0 and rfid := undef and atPack</td>
<td>ignored</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state observer</th>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>inUse</td>
<td>inUse &lt;= capacity &lt;= undef</td>
</tr>
<tr>
<td>capacity</td>
<td>capacity &gt;= 0</td>
</tr>
<tr>
<td>rfid</td>
<td>rfid &gt;= undef =&gt; not inDept</td>
</tr>
<tr>
<td>inDept</td>
<td>inDept =&gt; inUse and rfid = undef and cont = 0 and not atPack</td>
</tr>
<tr>
<td>cont</td>
<td>(cont &gt;= 0 =&gt; inUse and capacity &gt;= cont) and (cont &gt;= 0 =&gt; not inDept)</td>
</tr>
<tr>
<td>atPack</td>
<td>atPack =&gt; (not inDept and inUse)</td>
</tr>
<tr>
<td>final</td>
<td>not final</td>
</tr>
</tbody>
</table>

**Fig. 15** Basket example: revised events, state observers, invariants and reactions

**Fig. 16** Basket example: table for determining the states

<table>
<thead>
<tr>
<th>Basket</th>
<th>RFIDlabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>capacity</td>
<td>float [0..1]</td>
</tr>
<tr>
<td>rfid</td>
<td>name: string</td>
</tr>
<tr>
<td>day</td>
<td>day: Date</td>
</tr>
<tr>
<td>fruit</td>
<td>fruit: Fruit</td>
</tr>
</tbody>
</table>

**Fig. 17** Basket example: the revised class diagram and the state machine
The **Cybernaut** case

A cybernaut answers immediately at the mail of the friends, whereas (s)he answers only to one out of three mails of the acquaintances in a row.

New friends and new acquaintances of the cybernaut may be added from time to time.

In the case no one write to she/him for one day (s)he sends an email to a friend chosen by chance, and if (s)he has no friend in that situation (s)he ends hers/his career as a cybernaut.
Fig. 23 Example Cybernaut: the state machine
The Cybernaut case 2

A cybernaut answers to the mail of anyone only when it is at work (9-18) and when the boss is not in his office, in all other case it drops the mail.

In Fig. 25 we present the initial version of the artifacts required by our method for this case. Then we perform the required checks and discover that the two state observers are never updated. Thus we introduce some time and change events (corresponding to reaching 9.00 in the morning and 18.00 in the evening, and to the change of the situation of the boss. We show in Fig. 26 the revised artifact.