Chapter 9
Probabilistic Model Checking of SysML Activity Diagrams

Incorporated modeling and analysis of both functional and non-functional aspects of today's systems behavior represents a challenging issue in the field of formal methods.

In this chapter, we look at integrating such an analysis on SE design models, focusing on probabilistic behavior. Indeed, SysML 1.0 [187] extends UML activity diagrams with probabilistic features. Thus, we propose to translate SysML activity diagrams into the input language of the probabilistic model-checker PRISM [204]. In Sect. 9.1, we explain our approach for the verification of SysML activity diagrams. In Sect. 9.2, we present the algorithm implementing the translation of SysML activity diagrams into PRISM input language. Section 9.3 is dedicated to the description of the property specification language, namely PCTL*. Finally, Sect. 9.4 illustrates the application of our approach on a SysML activity diagram case study.

9.1 Probabilistic Verification Approach

Our objective is to provide a technique by which we can analyze SysML activity diagrams from functional and non-functional points of view in order to discover subtle errors in the design. This allows one to reason with regard to the correction of the design from these standpoints, before the actual implementation. In these settings, probabilistic model checking allows the performance of both qualitative and quantitative analyses of the model. It can be used to compute expectation on systems performance by quantifying the likelihood of a given property being violated or satisfied in the system model. In order to carry out this analysis, we design and implement a translation algorithm that maps SysML activity diagrams into the input language of the selected probabilistic model checker. Thus, an adequate performance model that correctly captures the meaning of these diagrams has to be derived. More precisely, the selection of a suitable performance model depends on the understanding of the behavior captured by the diagram as well as its underpinning characteristics. It also has to be supported by an available probabilistic model checker.

M. Debbabi et al., *Verification and Validation in Systems Engineering*,
DOI 10.1007/978-3-642-15228-3_9,

The global state of an activity diagram can be characterized using the location of the control tokens. A specific state can be described by the position of the token at a certain point in time. The modification in the global state occurs when some tokens are enabled to move from one node to another. This can be encoded using a transition relation that describes the evolution of the system within its state space. Therefore, the semantics of a given activity diagram can be described using a transition system (automata) defined by the set of all the states reachable during the system's evolution and the transition relation thereof. SysML activity diagrams allow modeling probabilistic behavior, using probabilistic decision nodes. The outgoing edges of these nodes, quantified with probability values, specify probabilistic branching transitions within the transition system. The probability label denotes the likelihood of a given transition's occurrence. In the case of a deterministic transition, the probability is equal to 1. Furthermore, the behavior of activity diagrams presents non-determinism that is inherently due to parallel behavior and multiple instances execution. More precisely, fork nodes specify unrestricted parallelism, which can be described using non-determinism in order to model interleaving of flows' executions. This corresponds, in the transition system, to a set of branching transitions emanating from the same state, which allows the description of asynchronous behavior. In terms of probability labels, all transitions occurring due to non-determinism are labeled with probability equal to 1.

In order to select the suitable model checker, we need to first define the appropriate probabilistic model for capturing the behavior depicted by SysML activity diagrams. To this end, we need a model that expresses both non-determinism and probabilistic behavior. Thus, Markov decision process (MDP) might be a suitable model for SysML activity diagrams. Markov decision processes describe both probabilistic and non-deterministic behaviors. They are used in various areas, such as robotics [9], automated control [100], and economics [111]. A formal definition of MDP is given in the following [209]:

Definition 9.1 A Markov decision process is a tuple $M{=}(S, s_0, Act, Steps)$, where

- S is a finite set of states;
- $s_0 \in S$ is the initial state;
- Act is a set of actions;
- $Steps\colon S \to 2^{Act \times Dist(S)}$ is the probabilistic transition function that assigns to each state s a set of pairs $(a, \mu) \in Act \times Dist(S)$, where $Dist(S)$ is the set of all probability distributions over S, i.e., the set of functions $\mu\colon S \to [0, 1]$ such that $\sum_{s \in \mathcal{S}} \mu(s) = 1$.

We write $s \xrightarrow{a} \mu$ if, and only if, $s \in S$, $a \in Act$, and $(a, \mu) \in Steps(s)$, and we refer to it as a step or a transition of s. The distribution μ is called an a-successor of s. For a specific action $\alpha \in Act$ and a state s, there is a single α-successor distribution μ for s. In each state s, there is a non-deterministic choice between elements of $Steps(s)$ (i.e., between the actions). Once an action–distribution pair (a, μ) is selected, the action is performed and the next state, for example, s', is

determined probabilistically according to the distribution μ, i.e., with a probability equal to $\mu(s')$. In the case of μ of the form $\mu^1_{s'}$ (meaning the unique distribution on s', i.e., $\mu(s') = 1$), we denote the transition as $s \xrightarrow{a} s'$ rather than $s \xrightarrow{a} \mu^1_{s'}$.

Among the existing probabilistic model checkers, we have selected PRISM model checker. The latter is a free and open-source model checker that supports MDPs analysis and whose input language is both flexible and user-friendly. Moreover, PRISM is widely used in many application domains on various real-life case studies and is recognized for its efficiency in terms of data structure and numerical methods. In summary, to apply probabilistic model checking on SysML activity diagrams, these diagrams will need to be mapped into their corresponding MDPs using PRISM input language. With respect to properties, they must be expressed using probabilistic computation tree logic (PCTL*), which is commonly used in conjunction with discrete-time Markov chains (DTMC) and MDP [20]. Figure 9.1 illustrates the synopsis of the proposed approach.

In the next section, we present the algorithm that we devise for the systematic mapping of SysML activity diagrams into the corresponding PRISM MDP code.

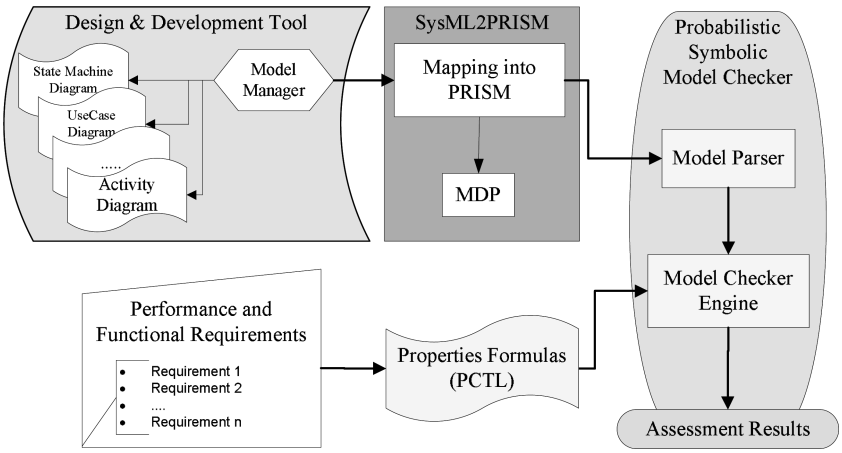

Fig. 9.1 Probabilistic model checking of SysML activity diagrams

9.2 Translation into PRISM

To translate SysML activity diagrams into PRISM code, we assume a single initial node and a single activity final node. Nevertheless, this is not a restriction, since we can replace a set of initial nodes by one initial node connected to a fork node and a set of activity final nodes by a merge node connected to a single activity final node.

Definition 9.2 A SysML activity diagram is a tuple $A = (N, N_0, type, next, label)$ where

- N is the set of activity nodes of types action, initial, final, flow final, fork, join, decision, and merge;
- N_0 is the initial node;
- $type$: $N \rightarrow \{action, initial, final, flowfinal, fork, join, decision, merge\}$ that associates with each node its corresponding type;
- $next$: $N \rightarrow \mathcal{P}(N)$ a function that returns for a given node the set (possibly singleton) of nodes that are directly connected to it via its outgoing edges;
- $label$: $N \times N \rightarrow Act \times]0, 1]$ a function that returns the pair of labels (g,p), namely the guard and the probability on the edge connecting two given nodes.

We rely on a fine-grained iterative translation of SysML activity diagrams into MDP. Indeed, the locus of control is tracked on both action and control nodes. Thus, each of these nodes is represented by a variable in the corresponding PRISM model. The join node represents a special case since the corresponding control passing rule is not straightforward [188] compared to the other control node rules. More precisely, a join node has to wait for a locus of control on each incoming edge in order to be traversed. Therefore, we need to keep a variable for each pin of a given join node. We also define a boolean formula corresponding to the condition of synchronization at each join node. Moreover, multiple instances of execution are allowed, thus the number of tokens in a given node is represented by an integer number denoting active instances at a certain point in time. At this point, we consider that in realistic systems a certain number of instances are active at the same time. Therefore, we model each variable as being an integer within a range $[0, \ldots, max\_inst]$ where the constant *max_inst* represents the maximum supported number of instances. This value can be tailored according to the application's needs.

Aside from the variables, the commands encode the behavior dynamics captured by the diagram. Thus, each possible progress of the locus of control corresponds to a command in PRISM code. The predicate guard of a given command corresponds to the precondition for triggering the control passing whereas the updates represent its effect on the global state. A given predicate guard expresses the ability of the source nodes to pass the control and also the destination nodes to accept it. A given update expresses the effect that control passing has on the number of active instances of the source and destination nodes. For instance, the fork node `F1` in Fig. 9.5 passes the control to each of its outgoing edges on condition that it possesses at least one locus of control and that the destination nodes are able to receive the token (did not reach their maximum number of instances). The modification in the control configuration has to be reflected in the updates of the command, where the fork node looses one locus of control and the number of active instances of the destination nodes increases. The corresponding PRISM command can be written as follows:

```
[F1]       F1>0 & Autofocus<max_inst & DetLight<max_inst &
                              D3<max_inst & !End  →
                  F1'=F1-1 & Autofocus'=Autofocus-1 &
                          DetLight'=DetLight-1 & D3'=D3-1;
```

This dependency of the predicates and updates on the nodes at source and at destination of the control passing inspired us to develop the systematic mapping procedure. In fact, the principle underlying our algorithm is that the predicates and updates for the source and destination nodes are generated separately so that, when composed together, they provide the whole final command. The commands are generated according to the type of the source node and the number of outgoing edges. For instance, in the case where the source node is a non-probabilistic decision node, the algorithm generates as many commands as outgoing edges. Concerning the probabilistic decision node, a only single command is needed, where the updates are the sum of all the probabilistic occurrences associated with different probabilistic choices. For a fork node, a single command enables all the outgoing target nodes. Finally, a single command suffices for nodes with a unique outgoing edge, such as action, join, merge, and initial.

The algorithm translating SysML activity diagrams into the input language of PRISM is presented in Figs. 9.2, 9.3, and 9.4. The algorithm visits the activity nodes using a depth-first search procedure and generates on-the-fly the PRISM commands. The main procedure $T(A,N)$ is illustrated in Fig. 9.2 and is continued in Fig. 9.3. Initially, the main procedure $T(A,\{N_0\})$ is called where A is the data structure representing the activity diagram and N_0 is the initial node. It is then called recursively, where N represents the set (possibly singleton) of the next nodes to be explored. The algorithm uses a function $C(n, g, u, n', p)$ illustrated in Fig. 9.4 where n is the current node representing the action name of the command, g and u are expressions, and n' is the destination node of n. The function C serves the generation of different expressions related to the destination node n', returning the final resulting command to be appended into the output of the main algorithm.

We make use of the usual *Stack* data structure with fundamental operations such as *pop*, *push*, and *empty*. We define user-defined types such as

- *PrismCmd* : a record type containing the fields act, grd, and upd corresponding respectively to the action, the guard and the update of the command of type *PrismCmd*.
- *Node* : a type defined to handle activity nodes.
- *PRISMVarId* : a type defined to handle PRISM variables identifiers.

The variable *nodes* is of type $Stack$ and serves to temporarily store the nodes that are to be explored by the algorithm. At each iteration, a current node *cNode* is popped from the stack *nodes* and its destination nodes in the activity diagram are stored in the list of nodes *nNode*. These destination nodes will be pushed in the stack in the next recursive call of the main algorithm. If the current node is already visited

```
nodes as Stack;
cNode as Node;
nNode as list_of_ Node;
vNode as list_of_ Node;
cmd as PrismCmd;
varfinal, var as PRISMVarId;
cmdtp as PrismCmd;
procedure T(A,N)
        / * Stores all newly discovered nodes in the stack */
    for all n in N do
        nodes.push(n);
    end for
    while not nodes.empty() do
        cNode := nodes.pop();
        if cNode not in vNode then
            vNode := vNode.add(cNode);
            if type(cNode)=final then
                cmdtp := C(cNode, eq(varfinal,1), raz(vars), null, 1.0);
            else
                nNode := next(cNode);
        / * Return the PRISM variable associated with the cNode */
                var := prismElement(cNode);
                if type(cNode)= initial then
                    cmdtp := C(cNode, eq(var,1), dec(var), nNode, 1.0)
                end if
                if type(cNode) in {action,merge} then
        / * Generate the final PRISM command for the edge cNode-nNode */
                    cmdtp := C(cNode, grt(var,0), dec(var), nNode, 1.0);
                end if
                if type(cNode)= join then
                    cmdtp := C(cNode, var, raz(pinsOf(var)), nNode, 1.0);
                end if
                if type(cNode)= fork then
                    cmdtp1 := C(cNode, grt(var,0), dec(var), nNode[0], 1.0);
                    cmdtp := C(cNode, cmdtp1.grd, cmdtp1.upd, nNode[1], 1.0));
                end if
```

Fig. 9.2 Translation algorithm of SysML activity diagrams into MDP – part 1

by the algorithm it is stored in the set of nodes *vNode*. In accordance with the current node's type, the parameters to be passed to the function *C* are computed. We denote by *varfinal* the PRISM variable identifier of the final node and *vars* represents the set of all PRISM variables of the current activity diagram. Finally, *max* is a constant value specifying the maximum value of all PRISM variables (of type integer). The algorithm terminates when the stack is empty and all instances of the main algorithm have stopped running. All the PRISM commands generated by the algorithm T are appended into a list of commands *cmd* (using the utility function *append*), which allows us to build the performance model.

We make use of the following utility functions:

- The functions *type*, *next*, and *label* are related to the accessing of the activity diagram structure and components.

```
                if type(cNode)= decision then
                    g := Π(label(cNode, nNode[0]), 1);
                    upd := and(dec(var), set(g,true)) ;
                    cmdtp1 := C(cNode, grt(var,0), upd, nNode[0], 1.0);
                    g := Π(label(cNode, nNode[1]), 1);
                    upd := and(dec(var), set(g,true)) ;
                    cmdtp2 := C(cNode, grt(var,0), upd, nNode[1], 1.0);
        /*Append both generated commands together before final appending */
                    append(cmdtp, cmdtp1);
                    append(cmdtp, cmdtp2);
                end if
                if type(cNode)= pdecision then
                    g := Π(label(cNode,nNode[0]), 1);
                    p := Π(label(cNode,nNode[0]), 2);
                    upd := and(dec(var), set(g,true)) ;
                    cmdtp1 :=C(cNode, grt(var,0), upd, nNode[0], p);
                    g := Π(label(cNode,nNode[1]), 1);
                    q := Π(label(cNode,nNode[1]), 2);
                    upd := and(dec(var), set(g,true)) ;
                    cmdtp2 := C(cNode, grt(var,0), upd, nNode[1], q);
        /* Merge commands into one final command with a probabilistic choice */
                    cmdtp :=merge(cmdtp1,cmdtp2);
                end if
            end if
        /*Append the newly generated command into the set of final commands */
            append(cmd, cmdtp);
            T(A,nNode);
        end if
    end while
end procedure
```

Fig. 9.3 Translation algorithm of SysML activity diagrams into MDP – part 2

- The function PRISMELEMENT takes a node as parameter and returns the PRISM element (either a variable of type integer or a formula) associated with the node.
- The function PINPRISMELEMENT takes two nodes as parameters, where the second is a $join$ node and returns the PRISM variable related to the specific pin.
- Various functions are used in order to build the expressions needed in the guard or the updates of the commands. The function *raz* returns the expression that is the conjunction of the expression of resetting the variables taken as parameter to their default values. The function $grt(x,y)$ returns the expression $x > y$, while function $less(x,y)$ returns the expression $x < y$. The function $dec(x)$ returns the expression $x' = x - 1$. The function $inc(x)$ returns the expression $x' = x + 1$. The function $not(x)$ returns the expression $!x$. The function $and(x,y)$ returns the expression $x\&y$. The function $eq(x,y)$ returns the expression $x = y$. The function $set(x,y)$ returns the expression $x' = y$.
- The Π is the conventional projection that takes two parameters, a pair (x, y) and an $index$ (1 or 2), and returns x, if $index = 1$, and y if $index = 2$.
- The function *pinsOf* takes as input the PRISM formula corresponding to a join node and extracts the corresponding pins variables into a list of PRISM variables.

```
function C(n, g, u, n′, p)
    var := prismElement(n′);
    if type(n′)=flowfinal then
        /∗ Generate the final PRISM command ∗/
        cmdtp := command(n,g,u,p);
    end if
    if type(n′)=final then
        u′ := inc(var);
        cmdtp := command(n, g, and(u,u′), p);
    end if
    if type(n′)=join then
        /∗ Return the PRISM variable related to a specific pin of the join ∗/
        varpin :=pinPrismElement(n,n′);
        varn :=prismElement(n);
        g1 := not(varn);
        g2 :=less(varpin,max);
        g′ := and(g1,g2);
        u′ :=inc(varpin,1);
        cmdtp = command(n,and(g,g′),and(u,u′),p);
    end if
    if type(n′) in {action,merge,fork,decision,pdecision} then
        g′ := less(var,max);
        u′ := inc(var,1);
        cmdtp = command(n,and(g,g′),and(u,u′),p);
    end if
return cmdtp;
end function
```

Fig. 9.4 Function generating PRISM commands

- The function *command* takes as input, in this order, the action name a, the guard g, the update u, the probability of the update p and returns the expression $[a]\ g \rightarrow p : u$.
- The function *merge* merges two sub-commands, taken as parameters, into one command consisting of a set of probabilistic updates. More precisely, it takes two parameters $cmdtp1 = [a]\ g1 \rightarrow p : u1$ and $cmdtp1 = [a]\ g2 \rightarrow q : u2$, then generates the command $[a]\ g1\ \&\ g2 \rightarrow p : u1 + q : u2$.

9.3 PCTL* Property Specification

In order to apply probabilistic model checking on the MDP model resulting from the translation algorithm, we need to express the properties in an appropriate temporal logic. For MDP models, we can use either LTL [248], PCTL [44], or PCTL* [248]. The probabilistic computation tree logic (PCTL) [44] is an extension of CTL [48], mainly with the added probability operator $\mathcal{P}$. PCTL* subsumes PCTL and LTL [248]. It is based on PCTL, where arbitrary combinations of path formulas and only propositional state formulas are allowed [10].

PCTL* syntax according to [10] is as follows:

$$\begin{aligned} \phi &::= true \mid a \mid \neg\,\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}[\psi] \\ \psi &::= \phi \mid \psi_1\,\mathcal{U}^t\,\psi_2 \mid \psi_1\,\mathcal{U}\,\psi_2 \mid \mathcal{X}\psi \mid \psi_1 \wedge \psi_2 \mid \neg\,\psi \end{aligned}$$

where a is an atomic proposition, $t \in \mathbb{N}$, $p \in [0, 1] \subset \mathbb{R}$, and $\bowtie \in \{>, \geq, <, \leq\}$.

PRISM extends the latter syntax in order to quantify probability values with the operator $\mathcal{P} =?$. For the case of MDP, all non-determinism has to be resolved. Thus, properties quantifying the probability actually reason about the minimum or maximum probability, over all possible resolutions of non-determinism, that a certain type of behavior is observed. Measuring the minimum/maximum probabilities provides the worst/best-case scenarios.

9.4 Case Study

In order to explain our approach, we present a SysML activity diagram of a hypothetical model of a digital photo-camera device. The diagram captures the functionality of taking a picture as illustrated in Fig. 9.5. The corresponding dynamics are rich enough to allow the verification of several interesting properties that capture important functional aspects and performance characteristics. We deliberately modeled some flaws into the design in order to demonstrate the applicability as well as the benefits of our approach. The process captured by the digital photo-camera activity diagram starts by turning on the camera (`TurnOn`). Subsequently, three parallel execution flows are spawned. The first one begins by (`AutoFocus`) followed by a decision checking the status of the memory (`memFull` guard). In the case where the memory is full, the camera cannot be used and it is turned off. The second parallel flow is dedicated to the detection of the ambient lighting conditions (`DetLight`) and it determines whether the flash is needed in order to take a picture. The third flow allows charging the flash (`ChargeFlash`) if it is not already charged. The action (TakePicture) executes in two possible conditions: either it is sunny (`sunny` $= true$) and the memory is not full (`memFull` $= false$) or the flash (`Flash`) is needed because of the lack of luminosity (`sunny`$= false$). Thereafter, the picture is stored in the memory of the camera (`WriteMem`) and the activity diagram ends after turning off the camera (`TurnOff`).

By applying our algorithm on the SysML activity diagram case study, we end up with the MDP described using PRISM language as shown in Fig. 9.6 and continued in Fig. 9.7. After supplying the model to PRISM, the latter constructs the reachable state space in the form of a state list and a transition probability matrix.

At the beginning, one can search for the presence of deadlock states in the model. This is expressed using property (9.1). It is also possible to quantify the worst/best-case probability of such a scenario happening using properties (9.2) and (9.3):

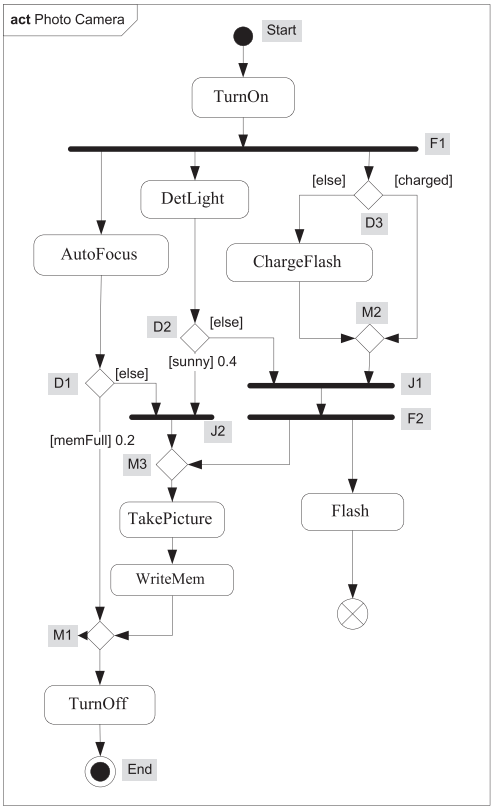

Fig. 9.5 Case study: digital camera activity diagram – flawed design

$$\text{``init''} \Rightarrow \texttt{P>0}\,[\,\texttt{F}\,\text{``deadlock''}\,] \tag{9.1}$$

$$\texttt{Pmax} =?\,[\,\texttt{F}\,\text{``deadlock''}\,] \tag{9.2}$$

$$\texttt{Pmin} =?\,[\,\texttt{F}\,\text{``deadlock''}\,] \tag{9.3}$$

The labels `"init"` and `"deadlock"` in property (9.1) are built-in labels that are true for, respectively, initial and deadlocked states. Property (9.1) states that, from an initial state, the probability of eventually reaching a deadlocked state is greater than 0. This returns *true*, which means that the property is satisfied in some states of the model. However, after further investigation, we found that there is only one deadlocked state due to the activity final node in the activity diagram. This deadlock can be accepted since, according to the desired execution, at the activity final node, the activity terminates and there are no outgoing transitions.

It is also important in the case of activity diagram to verify that we can eventually reach the activity final node once the activity diagram has started. Such a property

```
mdp
const int max_inst = 1;
formula J1 = J1_pin1>0 & J1_pin2>0 ;
formula J2 = J2_pin1>0 & J2_pin2>0 ;

module mainmod
memful : bool init false;
sunny : bool init false;
charged : bool init false;
Start : bool init true; TurnOn : [0 .. max_inst] init 0; F1 : [0 .. max_inst] init 0;
Autofocus : [0 .. max_inst] init 0; DetLight : [0 .. max_inst] init 0;
D3 : [0 .. max_inst] init 0; ChargeFlash : [0 .. max_inst] init 0;
D1 : [0 .. max_inst] init 0; D2 : [0 .. max_inst] init 0; F2 : [0 .. max_inst] init 0;
J1_pin1 : [0 .. max_inst] init 0; J1_pin2 : [0 .. max_inst] init 0;
J2_pin1 : [0 .. max_inst] init 0; J2_pin2 : [0 .. max_inst] init 0;
M1 : [0 .. max_inst] init 0; M2 : [0 .. max_inst] init 0; M3 : [0 .. max_inst] init 0;
TakePicture : [0 .. max_inst] init 0; WriteMem : [0 .. max_inst] init 0;
Flash : [0 .. max_inst] init 0; TurnOff : [0 .. max_inst] init 0; End : bool init false;

[Start] Start & TurnOn<max_inst & !End → Start'=false & TurnOn'=TurnOn + 1;

[TurnOn] TurnOn>0 & F1<max_inst & !End → TurnOn'=TurnOn − 1 & F1'=F1 + 1;

[F1] F1>0 & Autofocus<max_inst & DetLight<max_inst & D3<max_inst & !End →
 F1'= F1 − 1 & Autofocus'=Autofocus + 1 & DetLight'=DetLight + 1 & D3'=D3 + 1;

[Autofocus] Autofocus>0 & D1<max_inst& !End →
 Autofocus'=Autofocus − 1 & D1'=D1 + 1;

[DetLight] DetLight>0 & D2<max_inst& !End →
DetLight'= DetLight − 1 & D2' = D2 + 1 ;

[D3] D3>0 & ChargeFlash <max_inst& !End →
 ChargeFlash'= ChargeFlash + 1 & D3'=D3 − 1& (charged'=false);

[D3] D3>0 & M2<max_inst& !End →
  M2' = M2 + 1 & D3'= D3 − 1& (charged'=true);

[D1] D1>0      & M1<max_inst &  J2_pin1<max_inst & !J2 & !End →
0.2 : (M1'=M1 + 1) & (D1'=D1 − 1)& (memful' = true) +
0.8 : (J2_pin1'=J2_pin1 + 1) & (D1'=D1 − 1)& (memful' = false);

[D2] D2>0 & J2_pin2<max_inst & J1_pin1<max_inst & !J1 & !J2 & !End →
 0.6 : (J1_pin1'=J1_pin1 + 1) & (D2'=D2 − 1) & (sunny'=false) +
 0.4 : (J2_pin2'=J2_pin2 + 1) & (D2'=D2 − 1) & (sunny'=true);

[ChargeFlash] ChargeFlash>0 & M2<max_inst& !End →
 M2'= M2 + 1   & ChargeFlash'=ChargeFlash − 1;
```

Fig. 9.6 PRISM code for the digital camera case study – part 1

```
[M2] M2>0      & J1_pin2<max_inst & !J1 & !End  →
M2'= M2 − 1 & J1_pin2'=J1_pin2 + 1;

[M1]  M1>0 &TurnOff <max_inst      & !End  →
 TurnOff'=TurnOff + 1 & M1'= M1  − 1;

[J2]  J2  &TakePicture<max_inst & !End  →
  TakePicture'=TakePicture + 1 &  J2_pin1'=0 & J2_pin2'=0 ;

[J1]  J1 & F2<max_inst & !End  →
F2'=F2 + 1 & J1_pin1'=0 & J1_pin2'=0 ;

[F2] F2>0      & Flash<max_inst & TakePicture <max_inst & !End  →
  F2'=F2 − 1 & Flash'= Flash  + 1 & TakePicture'=TakePicture + 1;

[TakePicture] TakePicture >0 & WriteMem<max_inst & !End  →
TakePicture' = TakePicture − 1 & WriteMem'= WriteMem  + 1;

[WriteMem] WriteMem>0 & M1<max_inst & !End  →
  WriteMem'= WriteMem − 1 & M1'=M1  + 1;

[TurnOff] TurnOff >0 & !End  →
  TurnOff'=TurnOff  − 1 & End'=true;

[End] End      →
 TurnOn'=0 & F1'=0 & Autofocus'=0 & DetLight'=0 & D3'=0 & ChargeFlash'=0 & D1'=0
& D2'=0 & J1_pin1'=0 & J1_pin2'=0 & F2'=0  & J2_pin1'=0 & J2_pin2'=0 & M1'=0 &
 M2'=0 & M3'=0 & TakePicture'=0 &  WriteMem'=0 & Flash'=0 & TurnOff'=0&
 (memful' = false) & (sunny'=false) & (charged'=false);

endmodule
```

Fig. 9.7 PRISM code for the digital camera case study – part 2

is stated in property (9.4). Properties (9.5) and (9.6) are used in order to quantify the probability of such a scenario happening:

$$\texttt{TurnOn>=1 => P>0 [F End]} \tag{9.4}$$

$$\texttt{Pmax =? [F End]} \tag{9.5}$$

$$\texttt{Pmin =? [F End]} \tag{9.6}$$

Property (9.4) returns *true* and properties (9.5) and (9.6) both return the probability value 1. This represents satisfactory results, the final activity being always reachable.

The first functional requirement states that the `TakePicture` action should not be activated if the memory is full (`memfull=true`) or if the `Autofocus` action is still ongoing. Thus, we would like to evaluate the actual probability for this scenario to happen. Since we are relying on MDP model, we need to compute the minimum (9.7) and the maximum (9.8) probability measures of reaching a state where either the memory is full or the focus action is ongoing while taking a picture:

$$\texttt{Pmin} =? [\ \texttt{true U (memfull} \mid \texttt{Autofocus} \geq 1) \& \texttt{TakePicture} \geq 1\ ] \quad (9.7)$$

$$\texttt{Pmax} =? [\ \texttt{true U (memfull} \mid \texttt{Autofocus} \geq 1) \& \texttt{TakePicture} \geq 1\ ] \quad (9.8)$$

The expected likelihood for this scenario should be null (impossibility). However, the model checker determines a non-zero probability value for the maximum measurement (Pmax $= 0.6$) and a null probability for the minimum. This shows that there is a path leading to an undesirable state, thus pointing out to a flaw in the design. On the activity diagram, this is caused by a control flow path that leads to the TakePicture action and this being done independently of the evaluation of the memfull guard and of the termination of the action AutoFocus. In order to correct this misbehavior, the designer must alter the diagram so that the control flow reaching the action AutoFocus and subsequently evaluating the guard memfull to false has to synchronize with all the possible paths leading to TakePicture. This might be done using a fork node that splits two threads, each having to synchronize with a possible flow before activating TakePicture. Thus, we block the activation of TakePicture action unless AutoFocus eventually ends and memory space is available in the digital camera. Figure 9.8 illustrates the corrected SysML activity diagram.

As the main function of the digital photo camera device is to take pictures, we would like to measure the probability of taking a picture in normal conditions. The corresponding properties are specified as follows:

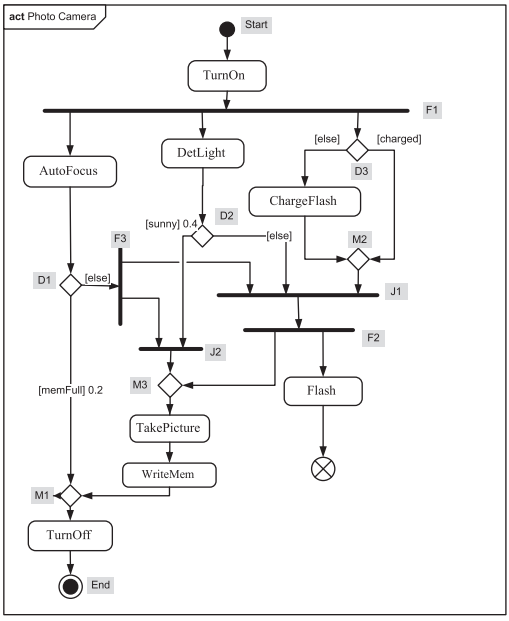

Fig. 9.8 Case study: digital camera activity diagram – corrected design

$$\texttt{P}min =? [\, true \,\texttt{U}\, \texttt{TakePicture} \geq 1] \tag{9.9}$$

$$\texttt{P}max =? [\, true \,\texttt{U}\, \texttt{TakePicture} \geq 1] \tag{9.10}$$

The measures provided by the model checker are, respectively, $\texttt{Pmin} = 0.8$ and $\texttt{Pmax} = 0.92$. These values have to be compared with the desired level of reliability of the system.

We applied probabilistic model checking on the corrected design in order to compare both the flawed and corrected SysML activity diagrams. The comparison is summarized in Table 9.1. The correction of the design has removed the flaw revealed by property (9.8), the probability value became 0. However, we lost in terms of reliability in the best-case scenario, since the maximum probability calculated for property (9.10) has dropped to 0.8 instead of 0.92.

Table 9.1 Comparative assessment of flawed and corrected design models

Properties	Flawed design	Corrected design
(9.1)	*true*	*true*
(9.2)	1	1
(9.3)	1	1
(9.4)	*true*	*true*
(9.5)	1	1
(9.6)	1	1
(9.7)	0.0	0.0
(9.8)	0.6	0.0
(9.9)	0.8	0.8
(9.10)	0.92	0.8

9.5 Conclusion

This chapter presented a translation algorithm that was designed and implemented in order to enable probabilistic model checking of SysML activity diagrams. The algorithm maps these diagrams into their corresponding Markov decision process (MDP) models. The code is written in the input language of the selected probabilistic model checker, i.e., PRISM. Moreover, a case study was presented in order to show the practical benefits of using the presented approach. Finally, MDP allowed the interpretation and analysis of SysML activity diagrams for systems that exhibit asynchronous behavior. In Chap. 10, we present a methodology for analyzing SysML activity diagrams with a specific consideration for time constraints on activity action nodes.

http://www.springer.com/978-3-642-15227-6

Verification and Validation in Systems Engineering
Assessing UML/SysML Design Models
Debbabi, M.; Hassaïne, F.; Jarraya, Y.; Soeanu, A.; Alawneh, L.
2010, XXVI, 248 p., Hardcover
ISBN: 978-3-642-15227-6