Chapter 7
Analysis and Implementation of Embedded System Models: Example of Tags in Item Management Application

Bojan Nokovic and Emil Sekerinski

7.1 Introduction

Verification of probabilistic systems is a technique for establishing if quantitative properties hold for a particular system model. The properties are expressed in temporal logic extended with probabilistic and reward operators. The model can be specified by engineers in a high-level modelling language as a variant of Markov chain processes annotated with costs and rewards, and used as input for a probabilistic model checker, e.g. [1]. Such system models can serve only for analysis.

Traditional state machines are flat and sequential in nature. To effectively allow representing complex behavior, such as that of communication protocols, statecharts, which are hierarchical state machines with concurrency and broadcasting were introduced [2]. Hierarchy is a structuring method that allows the developer to maintain an overview of large and complex applications. The most abstract view is at the outermost level and zooming in reveals details in lower level views. The design process begins with an outline of the application and then stepwise adds functionality. Concurrency and broadcasting are used to describe parallel tasks and communication.

Statecharts are used as a graphical specification tool for reactive systems, but they are executable and compilable like programming languages [3]; pCharts extend statecharts further with probabilistic transitions, timed transitions, stochastic timing, state invariants, and costs/rewards assigned to states and transitions [4, 5]. pCharts

B. Nokovic (✉) • E. Sekerinski
Computing and Software Department, McMaster University, Main Street West, 1280, Hamilton, ON, Canada
e-mail: nokovib@mcmaster.ca; emil@mcmaster.ca

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_7

are supported by *pState*,[1] a tool for the *holistic* design: in addition to generating executable code, *pState* can be used to model the system's environment and to verify quantitative properties like resource consumption (e.g. power), reliability (e.g. lost messages, life expectancy), and performance (e.g. throughput). Such queries can be specified directly on pCharts.

We use an application with electronic tags to illustrate the holistic design process. Section 7.2 reviews the design process. Section 7.3 gives an overview of the architecture and functionality of *pState*. Section 7.4 describes the process of executable code generation from pCharts. Section 7.5 presents the process of generating Markov decision processes and probabilistic timed automata for the PRISM probabilistic model checker. Section 7.6 describes the generation of the executable code framework. The remaining sections present a case study: on the example of the DASH-7 ISO/IEC 18000-7.2 communication protocol, we first give a system collision model in Sect. 7.8, then a model of power consumption in Sect. 7.9, and finally the executable code framework in Sect. 7.10. The final section summarizes the contribution.

7.2 A Holistic Design Process

Existing automated tools for analyzing discrete, timed, probabilistic, or stochastic models have a textual user interface, which makes them less suitable for engineers developing larger systems. Visualization of models in the form of hierarchical state machines, like statecharts, allows a view where the whole system is represented from the perspective of related states. An extension of statecharts with probabilistic transitions, timed transitions, and stochastic timing is proposed in [6]. Invariantcharts, statecharts with state invariants are introduced in [7]. pCharts support probabilistic transitions, timed transitions, stochastic timing, state invariants and add costs/rewards assigned to states or transitions. Through the *pState* editor, the pChart system model is entered. Quantitative queries are specified directly in the pCharts. After validation, a system without timed transition can be verified over a Markov decision process for systems (MDP) and a system with timed transitions over a probabilistic timed automaton (PTA); these are passed to a probabilistic model checker. The correctness of transitions with respect to state invariants is checked with a combination of the probabilistic model checker and a satisfiability modulo theories (SMT) solver. Executable code for the software part of the system can be generated and its worst-case execution time (WCET) analyzed. The architecture of *pState* is in Fig. 7.1.

[1] http://pstate.mcmaster.ca.

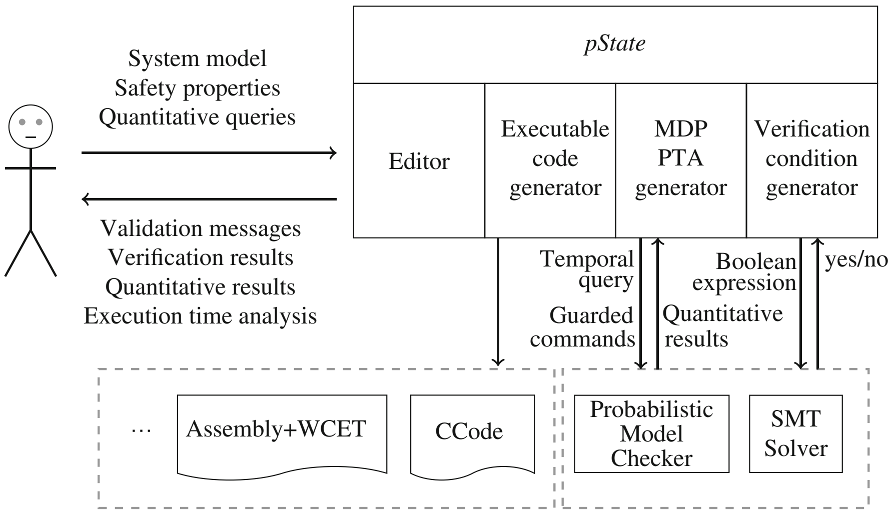
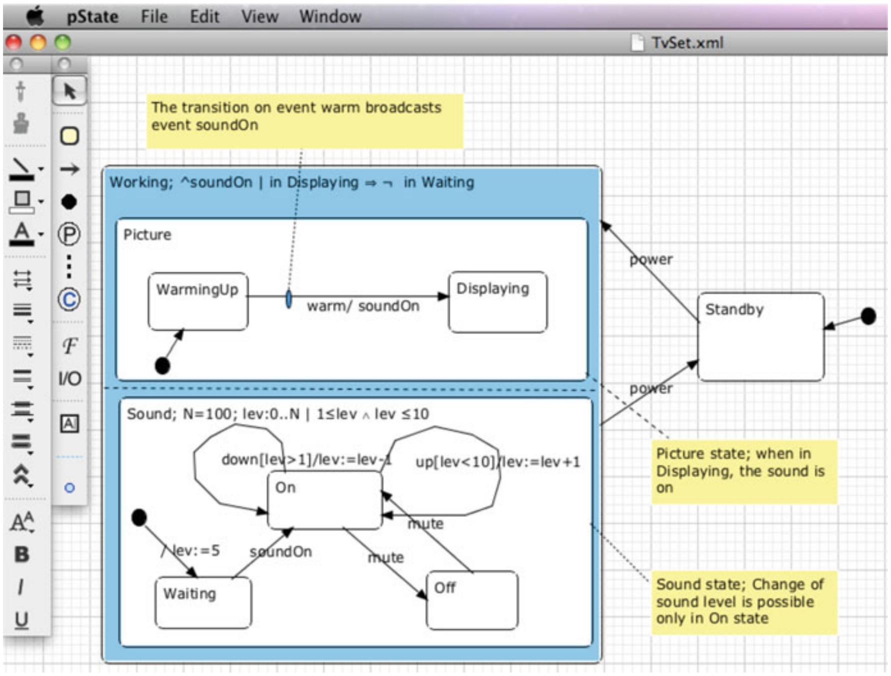


Fig. 7.1 Top-level *pState* architecture

7.3 pState Editor

The editor is designed on the JHotDraw (JHD) 7.6 framework [8]. As a starting point we use frameworks from the *org.jhotdraw.samples* package. Figure 7.2 is a view of the *pState* graphical interface, which shows features of a TV set represented as a pChart. Components like states and transitions are added in a drag-and-drop fashion using icons in the toolbar. States without children are called Basic states. On the TV set, chart states *Standby, WarmingUp, Displaying, Waiting, On*, and *Off* are Basic states. Compositional states are either AND states or XOR states. State *Working* is an AND state, it has two children, *Pictures* and *Sound*, separated by a dashed line. When the chart is in *Working*, it is at the same time in both *Picture* and *Sound*. Composite XOR states are (1) *Picture* with two Basic states *WarmingUp* and *Display*, (2) *Sound* with three children *Waiting, On*, and *Off*, and (3) the top state *root* with two children, *Working* and *Standby*.

The TV control activity is partitioned into two states, the Basic state *Standby*, and AND state *Working*. The initial state is *Standby*. When the chart is in *Working*, it is in both the *Picture* and *Sound* XOR states. Within *Picture* the chart is in one of the basic states *WarmingUp* or *Displaying*, within *Sound* the system is in one of the Basic states *Waiting*, *On*, or *Off*. The invariant of *Working* specifies that whenever *Picture* is in *Displaying*, *Sound* must not be in *Waiting*, i.e. must be either in *On* or *Off*. The invariant of *Sound* specifies that the sound level *lev* must be between 1 and 10; the invariant must be established by the initialization of *Sound* and be preserved by all transitions within *Sound*. The event *power* causes the chart to flip between *Standby* and *Working*, no matter in which substates of *Working* the chart is. The transition on event *warm* broadcasts event *soundOn*. The transition on events *down*



The transition on event warm broadcasts event soundOn

Picture state; when in Displaying, the sound is on

Sound state; Change of sound level is possible only in On state

Fig. 7.2 Statecharts with invariants for TV set

can only be taken if *lev* $>$ 1 and when taken, will decrement *lev*. The transition on *power* to *Working* sets *Picture* and *Sound* to the default initial states *WarmingUp* and *Waiting* and sets *lev* to 5.

The design tools in Fig. 7.2 are for selection, state (basic and composed), transitions, initial pseudostate, probabilistic pseudostate, concurrency line, choice pseudostate, quantitative query, and comment. It is straightforward to add other tools to the button factory. We additionally use the standard attribute bar with all selections from the JHD framework. This bar is an example of how new features, like colour of the figure, can be added to the drawing editor.

7.4 From Hierarchical Charts to Code

pState generates code according to an *event-centric* interpretation, in which events are executable procedures, implying that an event is processed before the next one arrives. This interpretation is according to the *requirements-oriented* semantics [9]. This is in contrast to the *implementation-oriented* semantics based on the *state-centric* interpretation in UML and Statemate [10], in which events are data in queues. The event-centric interpretation was already used by *iState*, the predecessor of *pState* [11]. The event-centric approach is suitable for those kind of reactive systems where events are processed quickly enough that queueing is not needed and where blocking of events is undesirable. This semantic is close to [12]. Currently we do not support *spontaneous* transitions—transitions without an event.

Hierarchical state machine diagrams consist, essentially, of just three components: a set of states, an initial state, and a set of transitions. The system starts at the initial state, then follows transitions on external events to move to other states. States can hold entire sub-state-machines within themselves. Concurrent states express orthogonality or independence. A transition t from a set of source states ss (of distinct concurrent states) to a set of target states tt (of distinct concurrent states), is a tuple written as $t = ss \xrightarrow{E[g]/b\ \$c} tt$, where E is the transition event, g is a Boolean expression, the transition guard, $\$c$ is a non-negative number, the cost of the transition and b is a statement, the transition body. In a *regular* transition, E is the event name, while in a *timed* transition E is the number of time units [5]. Transitions can be *probabilistic*, in which case target states are indicated as probabilistic alternatives [4]. Each transition must have E, while guard g, cost c, and body b are optional. All states are nested in the state *root*, which must not be the source or target of any transition.

In the transformation of a pChart to intermediate code, for each event, code associated with that event is generated and for every XOR state, an enumeration variable is generated holding the names of children states. The code generation is based on the recursive algorithm of [5].

The *scope* of a transition is the innermost state which contains all its source and target states. The grammar of the generated intermediate code has two mutually recursive productions, *Scopeop* and *Childop*. In the intermediate code, one variable for each state in the hierarchy is declared, starting with *root*, representing in which child state the system is. The algorithm for generating the *operation Op* of a regular event visits all transitions of one scope, starting with *root* as scope, before visiting transitions in children. The transitions on one scope are of the form

$$\mathit{Trigger} \rightarrow \mathit{Effect}\ [\,]\ \ldots\ [\,]\ \mathit{Trigger} \rightarrow \mathit{Effect}$$

with a *nondeterministic choice* ([]) among them, and each choice being *guarded* (→). These transitions take *priority* (//) over transitions in children. If the child is an XOR state, there is first a test to determine in which state the system is (Test), followed by the transitions with that child as scope. If the child is an AND state, then transitions on that event in all children are taken in parallel (‖). The *trigger* of a transitions contains tests for all the source states of the transition ($\mathit{Variable} = \mathit{State}$) and the guard (*Expr*). The *effect* of a transition is executing the body of the transition (*Statement*) in parallel with moving to target states (*Goto*), with a probabilistic choice (⊕) among such alternatives, such that the probabilities for each alternative ($\mathit{Probability} : \ldots$) sum up to 1. Thus the intermediate representation *Op* of a regular pCharts event is of the following form:

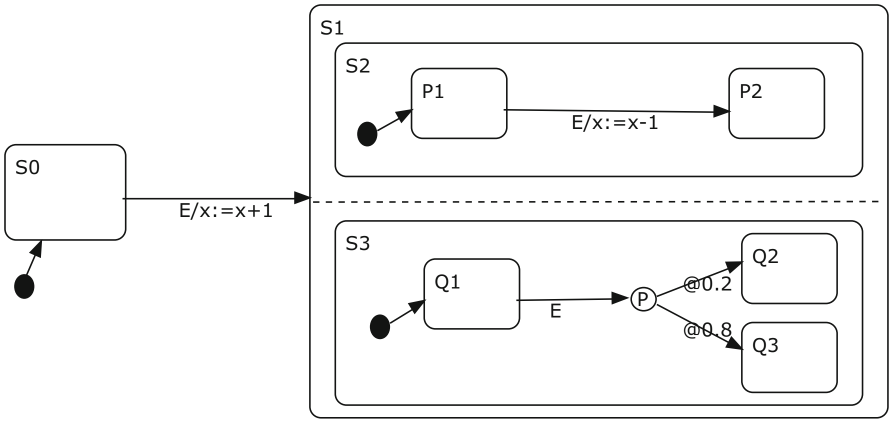


Fig. 7.3 Operation on event E

$$
\begin{array}{lll}
\mathit{Op} & ::= & \mathit{Scopeop} \\
\mathit{Scopeop} & ::= & (\mathit{Trigger} \rightarrow \mathit{Effect}\ [\!]\ \cdots\ [\!]\ \mathit{Trigger} \rightarrow \mathit{Effect})\ /\!/\ \mathit{Childop} \\
\mathit{Childop} & ::= & (\mathit{Test} \rightarrow \mathit{Scopeop}\ [\!]\ \cdots\ [\!]\ \mathit{Test} \rightarrow \mathit{Scopeop})\ /\!/\ \mathbf{skip} \\
 & | & \mathit{Scopeop} \parallel \cdots \parallel \mathit{Scopeop} \\
\mathit{Test} & ::= & \mathit{Variable} = \mathit{State} \\
\mathit{Trigger} & ::= & \mathit{Variable} = \mathit{State} \wedge \cdots \wedge \mathit{Variable} = \mathit{State} \wedge \mathit{Expr} \\
\mathit{Effect} & ::= & \mathit{Probability} : \mathit{Statement} \parallel \mathit{Goto} \oplus \cdots \oplus \mathit{Probability} : \mathit{Statement} \parallel \mathit{Goto} \\
\mathit{Goto} & ::= & \mathit{Variable} := \mathit{State} \parallel \cdots \parallel \mathit{Variable} := \mathit{State}
\end{array}
$$

As an example, the operation $op(E)$ of the event E in Fig. 7.3 is as follows:

$$
\begin{array}{l}
op(E) = \\
\qquad (root = S0 \rightarrow x := x + 1 \parallel root := S1 \parallel s2 := P1 \parallel s3 := Q1) \\
\qquad /\!/ \\
\qquad\quad root = S1 \rightarrow \\
\qquad\qquad (s2 = P1 \rightarrow x := x - 1 \parallel s2 := P2)\ /\!/\ \mathbf{skip} \\
\qquad\qquad \parallel \\
\qquad\qquad (s3 = Q1 \rightarrow 0.2 : s3 := Q2\ \oplus\ 0.8 : s3 := Q3)\ /\!/\ \mathbf{skip} \\
\qquad\quad /\!/ \\
\qquad\qquad \mathbf{skip}
\end{array}
$$

The full description of *generalized program statements* **skip**, **stop**, multiple assignment, guarded statement, nondeterministic choice, probabilistic choice, and parallel composition used to define the meaning of events is in [5, 13]. The body of a transition is an action or *chart statement*, like $x := x + 1$. The grammar of chart statement is

$$
\begin{array}{lll}
\mathit{ChartStatement} & ::= & \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{ChartStatement}\ [\textbf{else}\ \mathit{ChartStatement}] \mid \\
 & & \mathit{ChartStatement} \parallel \cdots \parallel \mathit{ChartStatement} \mid \\
 & & \mathit{Variable}, \ldots, \mathit{Variable} := \mathit{Expr}, \ldots, \mathit{Expr} \mid \\
 & & \mathit{Event} \\
\mathit{Expr} & ::= & \mathit{Variable} \mid \mathit{real} \mid \mathit{integer} \mid \textbf{true} \mid \textbf{false} \mid \mathit{UnOp}\ \mathit{Expr} \mid \\
 & & \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \mid\mid \textbf{in}\ \mathit{State} \\
\mathit{UnOp} & ::= & - \mid \neg \\
\mathit{BinOp} & ::= & + \mid - \mid * \mid \textbf{div} \mid \textbf{mod} \mid = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid \textbf{and} \mid \textbf{or}
\end{array}
$$

If an event leads to broadcasting of another event, the second one is executed in parallel with the first one, which imposes that there are no race conditions in the parallel execution. The translation of parallel statements needs extra processing since for executable code generation, parallel statements have to be converted into sequential statements using auxiliary variables. Parallel composition is first verified to be well defined such that variables assigned in parallel statements are disjoint, and then transformed to multiple assignments using the fact that $(x, y := E, F) = (x := E \parallel y := F)$ [5].

Before code generation, the *validation* performs three checks on charts: (1) Composite states must not be childless, AND state must have at least two children, each child of an AND state must be an XOR state; (2) all XOR states have initial transitions; (3) transitions between concurrent states are not allowed.

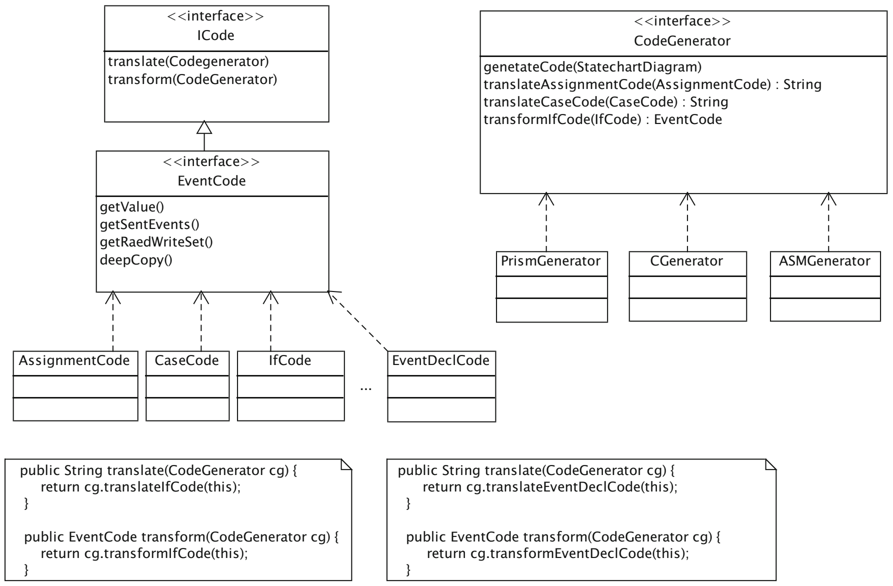
For target code generation, the *visitor* [14] pattern with two methods, *transform* and *translate* is employed, see Fig. 7.4. The elimination of parallel composition is done by *transform* and the creation of either executable code (C, assembly) or input code for a probabilistic model checker by *translate*.

7.5 Model Checker Input Code

From pCharts without timed transitions, *pState* generates an MDP model, and from pCharts with timed transitions, *pState* generates a PTA model as input for the PRISM model checker [15]. As PRISM requires the model to be a flat set of guarded commands with multiple (probabilistic) assignments as commands, after the elimination of parallel composition, the intermediate code is flattened. The full code generation algorithm is given in [5].

7.5.1 MDP

Markov decision processes are a variant of Markov chains that permit both probabilistic and nondeterministic choices. Our presentation of MDP follows [16–18].



```
public String translate(CodeGenerator cg) {
    return cg.translateIfCode(this);
}

public EventCode transform(CodeGenerator cg) {
    return cg.transformIfCode(this);
}
```

```
public String translate(CodeGenerator cg) {
    return cg.translateEventDeclCode(this);
}

public EventCode transform(CodeGenerator cg) {
    return cg.transformEventDeclCode(this);
}
```

Fig. 7.4 Class diagram of the visitor pattern in *pState*

Definition 1. A labelled Markov decision process is a tuple $M = (S, \bar{s}, A, p, l, r)$ where

– S is countable nonempty set of states;
– $\bar{s}$ is the set of initial states;
– A is the finite set of actions;
– p : S × A → Dist(S) is the transition probability function;
– l : S → 2^{AP} is the labelling function;
– r : S × A × S → R is the reward function.

and *AP* is a set of atomic propositions. We assume that *M* is time homogeneous; *S*, *A*, *p*, *l*, and *r* do not vary over time, and that *S* and *A* are discrete.

Example. The pChart of a simple MDP and the generated PRISM code are shown in Figs. 7.5 and 7.6. There are two transitions on *wakeup* from *S*0, the initial state, the choice between them being nondeterministic. One of the transitions is probabilistic, in which with 70 % probability state *S*1 is reached and with 30 % probability the system stays in the initial state. The other transition from *S*0 to *S*1 is deterministic, where on the event *wakeup* state *S*1 is always reached. The transition on the event *send* is deterministic and the transition on event *recv* is probabilistic. Rewards are assigned to the states by $\$r = e$, where and $e \geq 0$ is a real expression.

In *S*3 we specify two queries, the query ?$*r.max* returns maximum *costs* to reach *S*3, and the query ?*P.max* returns maximum *probability* to reach state *S*3. Those two properties are translated into PCTL formulae $R``r''max = ?[F(root = S3)]$ and

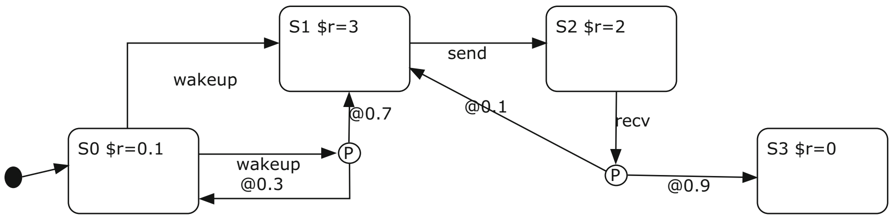


Fig. 7.5 State-transition diagram of the MDP model

```
mdp

const S0=0; const S1=1; const S2=2; const S3=3;

module mdpexample
      root :[0..3]   init  S0;

      [send]  (root=S1) −> (root'=S2);
      [wakeup] (root=S0) −> 0.3:(root'=S0) + 0.7:( root'=S1);
      [wakeup] (root=S0) −> (root'=S1);
      [recv]  (root=S2) −> 0.1:(root'=S1) + 0.9:( root'=S3);
endmodule

rewards "r"
      (root=S0): 0.1;
      (root=S1): 3;
      (root=S2): 2;
      (root=S3): 0;
endrewards
```

Fig. 7.6 MDP PRISM code generated by *pState*

$Pmin = ?[F(root = S3)]$, respectively. The calculated maximum costs to reach state $S3$ is 5.6984, and the maximum probability to reach $S3$ is 0.9999, that is 1. The error comes from floating point rounding of the model checker. In this example, there is a nondeterministic choice between the probabilistic and deterministic *wakeup* transitions from state S0 to state S1. Eventually, in both cases, the transition on event *wakeup* leads to S1 state. Calculated reward of 5.698 is maximum expected long-run reward. This is the same as a long-run average reward, but only if there are no nondeterministic transitions.

7.5.2 PTA

Timed automata (TA) provide a natural way for expressing timing delays of real-time systems [19]. On a TA, we can prove the correctness of finite-state real-time systems using the *trace* semantics originally proposed in a model for communicating sequential processes (CSP) [20]. Probabilistic timed automata (PTA) are an extension of TA used for formal modelling and analysis capabilities for systems with probabilistic, nondeterministic, and real-time characteristics [17]. PTA augmented with quantitative information in the form of costs or reward are called priced probabilistic timed automata. On a PTA model two main classes of properties can be analyzed, the minimum/maximum probability of reaching a target, possibly within a time bound and the minimum/maximum expected reward accumulated until a target is reached, using *quantitative abstraction refinement* and *statistical model checking* verification methods [21].

Definition 2. A probabilistic timed automaton (PTA) is a tuple
$P = (S, \bar{s}, \mathfrak{X}, A, inv, enab, prob, l)$, where

- S is the countable nonempty set of states;
- $\bar{s}$ is the set of initial states;
- $\mathfrak{X}$ is a finite set of clocks;
- A is the finite set of actions;
- inv : S $\rightarrow$ CC($\mathfrak{X}$) is an invariant condition, a clock constraint for each state;
- enab: S $\times$ A $\rightarrow$ CC($\mathfrak{X}$) is an enabling condition;
- prob: S $\times$ A $\rightarrow$ Dist($2^{\mathfrak{X}} \times$ S) is a (partial) probabilistic transition function;
- l : S $\rightarrow 2^{AP}$ is the labelling function;

Example. The pChart of a simple PTA and the generated PRISM code are shown in Figs. 7.7 and 7.8. The PTA has clock *rootclk* with initial value 0. In the state $S0$, the system waits for the *wakeup* event for 1 time unit. State $S0$ also allows a transition to state $S1$ when $rootclk = 1$ and the PRISM invariant $root = S0 \Rightarrow rootclk \leq 1$ forces the transition to be taken when *rootclk* reaches 1. In the state $S0$, the system waits for *wakeup* for a maximum of 1 time unit. If the event does not occur, it goes to next state on timed transition. On this model, properties like the expected time to reach state $S3$ or the probability of reaching state $S3$ in a given number of time units can be verified. In the state $S3$, we specify two queries: $?P.maxF < 10s$ returns 0.9, the maximum probability to reach state $S3$ in 10 time units (seconds) and $?P.minF < 10s$ returns 0.819, the minimum probability to reach $S3$ in 10 time units. Those properties cannot be verified on an MDP model. While PRISM uses abstract time units, in pCharts the time unit, here s, must be explicitly specified.

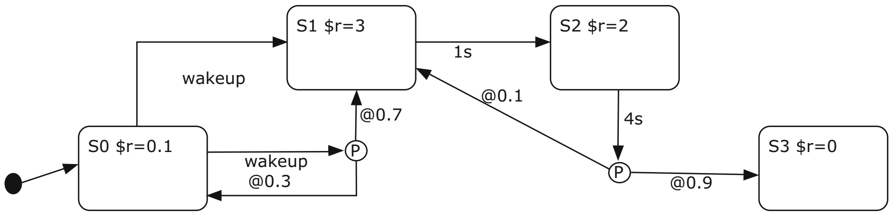


Fig. 7.7 State-transition diagram of the PTA model

```
pta

const S0=0; const S1=1; const S2=2; const S3=3;

module ptaexample
    root :[0..3]  init  S0;      rootclk  :  clock;

    invariant
        (root=S1=>rootclk<=1)& (root=S2=>rootclk<=4)
    endinvariant

    [wakeup] (root=S0) −> (root'=S1)&(rootclk'=0);
    [wakeup] (root=S0) −> 0.3:(root'=S0)&(rootclk'=0) + 0.7:( root'=S1)&(rootclk'=0);
    []  (root=S2)&(rootclk=4) −> 0.1:(root'=S1)&(rootclk'=0) +
                                      0.9:( root'=S3)&(rootclk'=0);
    []  (root=S1)&(rootclk=1) −> (root'=S2)&(rootclk'=0);
endmodule

rewards "r"
    (root=S0): 0.1;
    (root=S1): 3;
    (root=S2): 2;
    (root=S3): 0;
endrewards
```

Fig. 7.8 PTA PRISM code generated by *pState*

A PTA in PRISM is verified by one of two engines, *digital clocks* [22] and *stochastic games* [23]. The specification of queries or *quantitative properties* of a PTA is based on probabilistic computational tree logic PCTL [17, 24]. In the digital clock engine, clock variables are allowed in P (probability) operator expressions, as well as in F (*eventually*) and U (*until*) expressions. However, this engine does not support time-bounded reachability properties and clock constraints cannot use strict comparison operators, e.g. $rootclk < 2$. Also, comparison between clock variables is not allowed. Automata with such constraints are called *closed*, *diagonal-free* probabilistic timed automata. The digital clocks method is based on a language-level translation from a PTA model to an MDP model. In the *stochastic games* engine, properties cannot contain references to clocks. Only unbounded or time-bounded probabilistic reachability properties are allowed. For this, only the P operator is used. The basic types of path properties that can be used inside the P operator are: X (*next*), U (until), F (eventually), G (*always*), W (*weak until*), and R (*release*), but the *stochastic game* engine currently (V 4.2.1) only supports the F path operator. The S operator, used to reason about the steady-state behavior of model, and the R operator, used to calculate reward properties, are not supported.

7.5.3 Properties Specification

pState allows quantitative queries to be placed inside hierarchical states, making use of the state hierarchy, while the specification of properties in PRISM is done separately from the model.

For example, in *pState* we can attach $?P.min$ to a state, say S, to compute the minimal probability to reach S. If S is child of *root*, *pState* generates the PCTL formula $Pmin\ =?[F(root\ =\ S)]$ for PRISM. The same query can be attached to another state, possibly deeper in the hierarchy, and *pState* would generate a corresponding, more complex property specification. Similarly, if the reward property $?\$tran.max$ for computing the maximal reward to reach that state is placed in S, *pState* generates $R\{``tran''\}max\ =?[F(root\ =\ S)]$. Quantitative queries in *pState* are according to the following grammar:

$$\begin{array}{ll}
\textit{Query} & ::= ?(\textit{Probability} \mid \textit{Reward})(\mathbf{.min} \mid \mathbf{.max} \mid > \textit{real} \mid < \textit{real})[\textit{Bound}][\textit{Target}] \\
\textit{Bound} & ::= \mathbf{F} < \textit{Time} \\
\textit{Target} & ::= '('\textit{Expr}')' \\
\textit{Probability} & ::= \mathbf{P} \\
\textit{Reward} & ::= \$\textit{Identifier} \\
\textit{Time} & ::= \textit{digit}\{\textit{digit}\}(\mathbf{d} \mid \mathbf{h} \mid \mathbf{s} \mid \mathbf{ms}) \\
\textit{Identifier} & ::= \textit{letter}\{\textit{letter} \mid \textit{digit}\}
\end{array}$$

Quantitative queries are attached to a state or written in the special property box. Currently only simple properties can be attached to states. For more complex properties, which include more than one condition, property boxes have to be used. For instance, the PCTL formula $Pmax\ =?[F(rootclk\ <\ T)\&(root\ =\ S)]$ has to be specified in a property box. With this property we can calculate the probability that state S will be reached before T time units.

7.6 Executable Code

Target code is created by further translating the intermediate code, provided that there are no probabilistic transitions in the sub-chart for which code is to be generated. The intermediate code may contain parallel compositions emerging from broadcasting (transitions in concurrent states are taken in parallel) and multiple assignments. As multiple assignments are a special case of parallel composition, both are treated uniformly by introducing auxiliary variables and sequentializing, for example:

$$(x := y || y := x) = (x, y := y, x) = (\mathbf{var}\, h = x; x := y; y := h)$$

Specifications of costs/rewards are ignored for code generation. Nondeterministic choice with guarded choices is translated as **if-then-else** or **case** statements in the target code syntax. The abstract syntax of the executable code follows:

$$
\begin{aligned}
\textit{Statement} ::= \;& \textbf{if}\ \textit{Expr}\ \textbf{then}\ \textit{Statement}\ [\textbf{else}\ \textit{Statement}] \mid \\
& \textit{Statement}\ ;\ \ldots\ ;\ \textit{Statement} \mid \\
& \textit{Variable} := \textit{Expr} \mid \\
& \textbf{case}\ \textit{Variable}\ \textbf{of}\ \textit{State} : \textit{Statement} \ldots \textit{State} : \textit{Statement} \mid \\
& \textbf{call}\ \textit{Event} \mid \\
& \textbf{var}\ \textit{Variable} = \textit{Expr}\ ;\ \textit{Statement}
\end{aligned}
$$

pState generates code for PIC16F6xx in C or assembly language, and Libelium/Arduino code for ATmega1281 micro-controller. Both are 8-bit RISC-based micro-controllers.

7.6.1 PIC C Code

All executable files can be divided into two groups, (1) generated files and (2) prewritten files. *pState* generates the file *charts.c*, which defines the behavior of the application. Prewritten files *main.c, setupProcessor.c, Scheduler.h, actions.h* can be divided into two groups: target independent, and target dependent files, similar as for the assembly files shown in Fig. 7.9. Target independent files are *main.c*, and *Scheduler.h*. The file *main.c* defines the entry of the application, and initializes variables, chart states, and the scheduler. Then it enters an infinite loop which

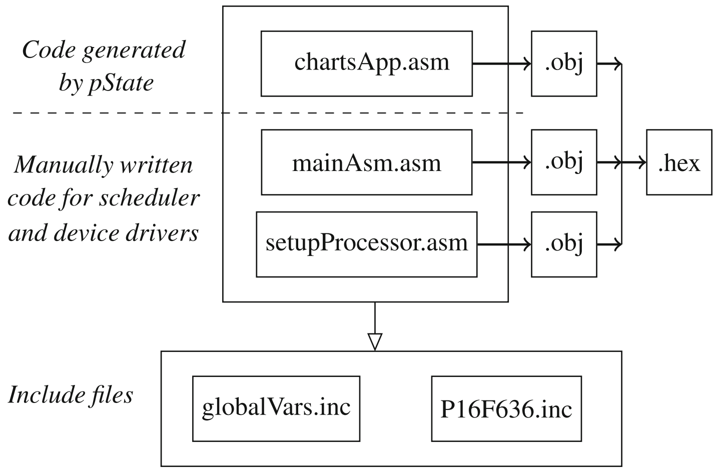


Fig. 7.9 Structure of target code

processes input events and schedules actions. The file *Scheduler.h* defines a data structure which holds timed events and defines functions to schedule and cancel timed events.

The target dependent file *setupProcessor.c* contains routines for processor input/output initialization, timer initialization, etc. The file *actions.c* specifies how external events will be processed. For instance, if a digital signal is connected to PORTA bit 0 of the PIC16F6xx micro-controller, and if the presence of a signal means high voltage on the pin, then that should be defined in *actions.c* as *#define SIGNAL* $(RA0 == 1)$.

7.6.2 PIC Assembly Code

Assembly code is created by translating the abstract executable code. Translation of **if-then-else** and **case** statements is straightforward. Most micro-controllers have an instruction which allow constants to be added immediately. In this approach generation of the code is delayed until the *mode* of an expression is known, which is known as *delayed code generation* [25].

We are using CBLOCK 0x20 or CBLOCK 0x40 to allow the variables declared within the block to automatically increment to the next general register, starting from 0x20 or 0x40. Address 0x40 and beyond are used for constants associated with state names, while 0x20 to 0x39 are used for variables.

Names of variables in assembly code are generated as lowercase letter state names. Variables are either integer subranges or Boolean. The generated code consists of state and variable declarations, assignments and expressions, state transitions, macros, statements, and timed transitions. Scheduler, initialization, and I/O actions are not generated from the specification, they are *write-once* code. In this way we have full control over the structure of the application, similar to the approach described in [26].

Code generation depends not only on individual symbols but also on the values of their attributes. We use a one-pass generation that delays emitting the code until the attributes are known [25]. The generated code depends on the fact if the value is held in a register or it is a known constant. If it is constant, the generated code will be smaller since the value does not need to be stored to working register before the operation is performed. Where the value is stored and how it is to be accessed is indicated by attributes of expressions. In our implementation we have the following attribute modes: *Reg* - special function registers, i.e. PORTA, STATUS, etc., *Var* - general purpose registers, *Const* - constant, *AccW* - working register or accumulator. In addition to those we have special modes for expressions like *VarPlusConst*, *VarMinusConst* which indicate operations of addition or subtraction between factors of type variable and constant. Once we know mode of an expression, optimized code can be generated.

7.6.3 Energia, Arduino-Like Code

Arduino is a C-derived programming language. Energia is an Arduino-like IDE for TI LaunchPad (Tiva C) development board. In our implementation, the target is the TM4C123 ARM micro-controller. The program is structured as two routines, *setup*() and *loop*(). The *setup*() routine contains the initialization of variables and is run only once. The *loop*() routine is then executed continuously, allowing variables to change and the program to respond to and control the board. The code can be compiled on the Energia IDE. Custom routines in Energia can be written to perform reoccurring tasks. They are declared like functions in C/C++, with function return type, name, and parameters. In our implementation we assume that no value is to be returned, so the event function type is *void*.

The code generated from the example in Fig. 7.10 is in Fig. 7.11. All states of the hierarchical structure are nested in the root state, which is declared as the variable *root*. pCharts allows direct declaration only of integer subranges and Boolean variables.

Functions that are unique to the Energia language and used to configure, read, and write specific ports of the micro-controller can be called in the body of transitions. Those functions have to be prewritten. They are ignored for the purpose of verification. If we need to set up some pin to be INPUT or OUTPUT, that is done by the *pinMode(pin,mode)* function; to read digital pin value, which can be HIGH or LOW, the function *digitalRead(pin)* is used, and to write to pin *digital-Write(pin,value)* is used. Handling an analog pin is done by *analogRead(pin,value)* and *analogWrite(pin,value)*. It reads and write the value from a specified analog pin with 10-bit resolution.

Untimed event can be executed by (1) polling the trigger of the event or (2) assigning external interrupt to the event. Polling can be done in a continuous loop or by a timer. In our implementation we call the *dispatcher* function in the loop to check if external trigger that causes the *On* or *Off* event is present. The same functionality can be achieved by calling *dispatcher* after a predefined amount of time (i.e. every 1ms). In the prewritten code of Fig. 7.12, the function that configures hardware, HW_Init(), and the function *dispatcher* are shown.

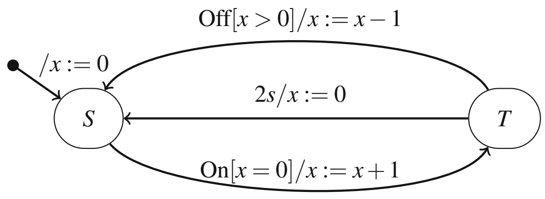


Fig. 7.10 Simple switch operation

```
/*
* Energia (Arduino) code generated from pCharts
*/

#include "OneMsTaskTimer.h"

/* Variables */
#define T 0
#define S 1

int x;
int root;

OneMsTaskTimer_t teExactly0={2000, exactly0, 0, 0};

void exactly0(){
    if ((root==T)) {
        x=0;
        root=S;
    }
}
void Off(){
    if ((root==T)&&(x>0)) {
        x=(x-1);
        root=S;
        OneMsTaskTimer::remove(&teExactly0);
    }
}
void On(){
    if ((root==S)&&(x==0)) {
        x=(x+1);
        root=T;
        OneMsTaskTimer::add(&teExactly0);
    }
}

void setup(){
    /* Initialization */
    HW_Init();
    root=S;
    x=0;
    OneMsTaskTimer::start(); // Start timer
}

void loop(){
    dispatcher();
}
```

Fig. 7.11 Generated code for the chart in Fig. 7.10

```
void HW_Init(){
  // Initialize the pushbutton pin as an input
  pinMode(PUSH1, INPUT_PULLUP);
  pinMode(PUSH2, INPUT_PULLUP);
}

void dispatcher (){
   noInterrupts ();
  if ( digitalRead (PUSH2)==LOW) {
    On();
  }
  if ( digitalRead (PUSH1)==LOW) {
    Off ();
  }
   interrupts ();
}
```

Fig. 7.12 Prewritten hardware-related code, target TM4C123 micro-controller

7.7 Contention Resolution in DASH-7 ISO/IEC 18000-7.2

The ISO/IEC 18000-7.2 [27] standard provides an air interface implementation for wireless, non-contact information system equipment for *item management* applications. The RFID equipment is composed of two principal components: tags and interrogators. We study a *system* with *active* tags, i.e. tags with own source of energy, like battery. Each tag has a unique serial number and other data. It is intended for attachment to a managed item. An interrogator is a device that communicates to tags in its RF communication range. The interrogator controls the master–slave protocol, reads information from the tag, directs the tag to store data, ensures message delivery and validity. We present the method by which an interrogator identifies and communicates with one or more tags present in the operating field of the interrogator over a common radio frequency channel. Tags do not transmit unless commanded to do so by the interrogator. An interrogator can communicate with tags individualy, or with the tag population as a whole.

7.7.1 Tag Collection and Collision Arbitration

The tag collection process is an iterative process that includes methods for coordinating responses from the tag population and handling collisions which occur when multiple tags transmit at the same time. The entire tag collection process is referred to as a *complete collection sequence*. Figure 7.13 shows a complete collection sequence consisting of a *wakeup period* (WP) followed by a series of *collection*

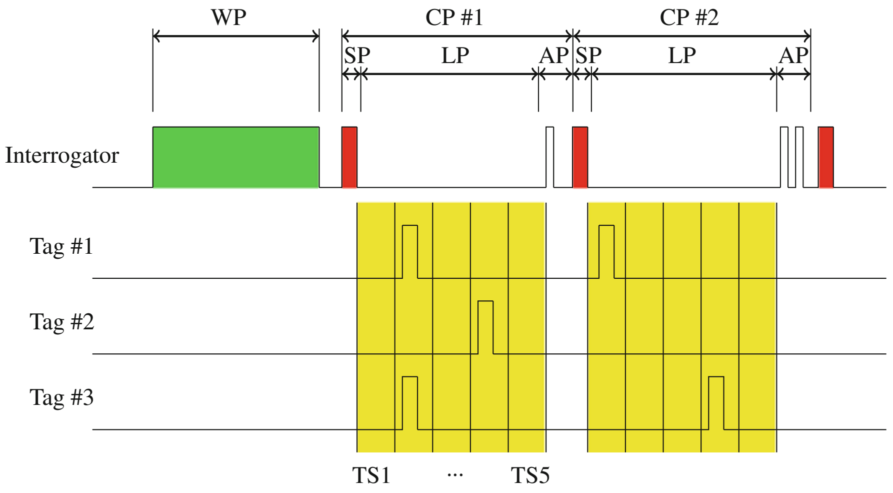


Fig. 7.13 Interrogator-tag communication timing diagram

periods (CP). Each collection period consists of a *synchronization period* (SP), a *listen period* (LP), and an *acknowledge period* (AP). The LP is further divided into multiple time slots (TS).

For three tags and five time slots as shown in Fig. 7.13, in the first communication period, tags #1 and #3 transmit in the same time slot, so there will be a collision. In the first acknowledge period there is an acknowledgment only for the message of tag #2. In the second communication period, tags #1 and #3 retransmit the message, but this time tag #1 transmits in the time slot 1, and tag #3 transmits in time slot 4, so there is no collision, and in the acknowledge period there are two acknowledgement messages.

7.8 Collision Model

We can calculate the collision probability by calculating the number of possible transmissions without collision and divide it by the total number of possible transmissions. For n tags transmitting, the first tag can transmit in any of the m time slots, the second tag should transmit in any of the $m-1$ slots to avoid collision, and so on. The number of transmissions without collision is

$$NC = m \cdot (m-1) \cdot \ldots \cdot (m-n+1) \tag{7.1}$$

while the number of all possible transitions is

$$AT = m \cdot m \cdot \ldots \cdot m = m^n \tag{7.2}$$

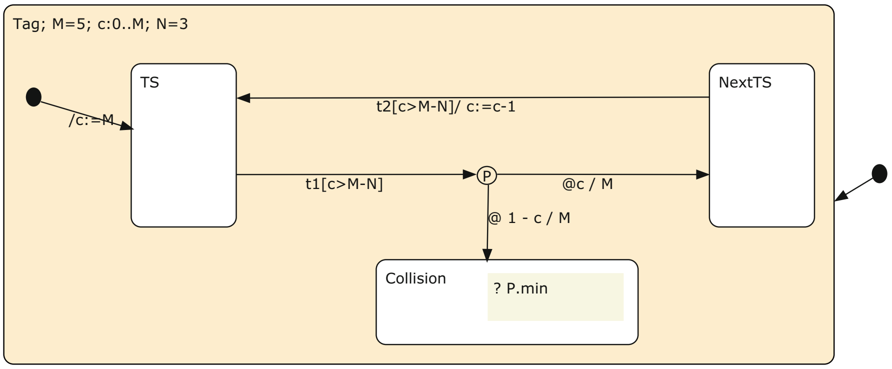


Fig. 7.14 Collision model, three tags, five time slots

The probability that a collision will happen is simply

$$1 - NC/AT \tag{7.3}$$

We assume a model of three tags $N = 3$ and five time slots $M = 5$. The number of possible transmissions without collisions is $NC = 5 \cdot 4 \cdot 3 = 60$, and the number of possible transmissions is $AT = 5 \cdot 5 \cdot 5 = 125$. The probability of at least one collision according to (7.3) is 0.52.

The collision model represented by pCharts is shown in Fig. 7.14. In the *Collision* state, by "? *P.min*″ we query the collision probability, or probability to go to *Collision* state, which is calculated as 0.52. To calculate the collision probability for a different number of time slots or a different number of tags, all we have to do is to assign new numbers to *M* or *N* in *Tag* state declaration.

7.9 Collection Period Power Consumption

The collision model in Fig. 7.14 is without timed transitions and the generated input code for the model checker is an MDP. But, in the power consumption model, we need to know how long tags stay in states and the current consumption in those states. The model of power consumption is shown in Fig. 7.17. From this model, *pState* generates a PTA.

On the PTA model we can query the average power consumption in one collection period (CP), taking into account the collision probability calculated on the collision model. The current consumption for a typical active tag during *transmission* is 9.2 *mA*, in *receiving* mode 0.2 *mA*, and in *standby* 0.0024 *mA* [28]. In the construction of the transitions we use data from Fig. 7.15. In the case of three tags and five time slots, the collision probability is 0.52, or 52 %, so the transition

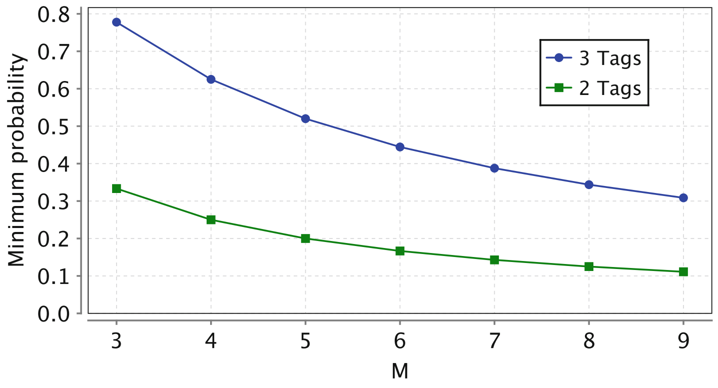


Fig. 7.15 Collision probability for tags $N = [2, 3]$ and time slots $M = [3..9]$

```
mdp

const  M = 5;
const  N = 3;
const  Collision =0;  const  NextTS=1; const TS=2;

module  collision
       tag :[0..2]    init  TS;
       c :[0.. M]  init  M;

       [t2]  (tag=NextTS)&(c>(M−N))  −> (c'=(c−1))&(tag'=TS);
       [t1]  (tag=TS)&(c>(M−N))  −> (1−(c/M)):(tag'=Collision) + (c/M):(tag'=NextTS);
endmodule
```

Fig. 7.16 PRISM code generated by *pState* for model shown in Fig. 7.14

from state *Tx* to *Next* is probabilistic with 52 % probability. That means if a collision happens, the tag needs another collection period to perform the operation. In the case of collision, the probability that all three tags select the same time slots is 4 %, which is modelled by a probabilistic transition from *Tx* to *ThreeCollisions* state. If there is a collision of two tags, in the next collection period, on five time slots, only those two tags are retransmitting. According to Fig. 7.15, the collision probability is 20 %, and that is represented by probabilistic transition from *TwoTags* to *Next* state. The maximum number of collection periods in the model is three (Fig. 7.16).

The queries are verified over the formulae $R\{``cons''\}max =?[F(tagconsum = End)]$ and $Pmax =?[F(tagconsum = End)\&(i = 3)]$. For the first, the calculated value for the electrical charge is 280.72 *mAms* (Fig. 7.17). The probability of not receiving all tags in three collection periods is calculated as 0.1106, or 11.06 %, so about 89 % of time all tags are read in three collection periods (Fig. 7.18).

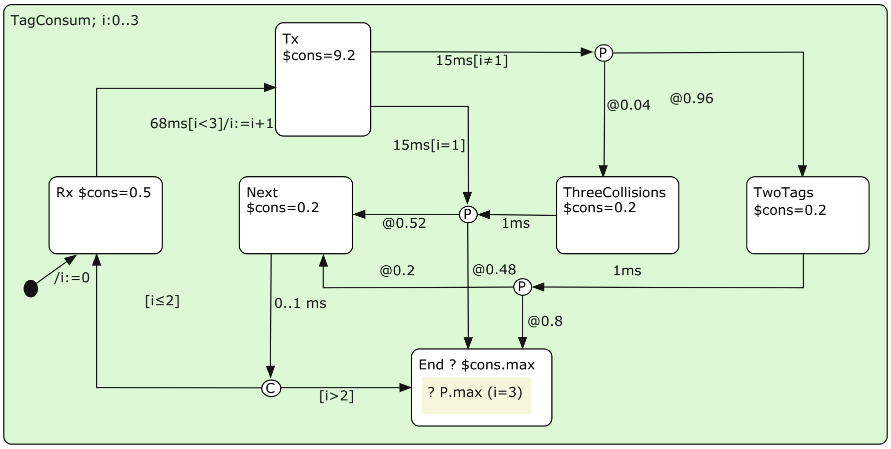


Fig. 7.17 Collection period power consumption

7.10 Executable Tag Code

Figure 7.19 gives the tag operation model. It has two parallel processes, *Tag*, which represent tag operation, and *Mode* which represents tag transition from sleeping to working mode and back. Initially, *Mode* is in *Sleep*, in which periodically, every 1*ms*, the presence of a wakeup signal is checked. If there is no signal, it goes back to *Sleep* and repeats the process after 1*ms*. If there is a wakeup signal, the indicator *field* is set to *true*, and the system goes first into *Field* and immediately to *FieldON*. From that state it goes to *Work* and broadcasts the event *WakeUp*. That event moves *Tag* from *Start* into *Preamble*. On that transition, procedure *WAKEUP* is called. It has to recognize the (*WP*) preamble, Fig. 7.13, and has to be executed in 2.45 to 4.8 seconds. Next, *Tag* goes into *CP* state in which it receives a command form the interrogator, and goes into listen period *LP*, in which the tag transmits its message in a randomly selected time slot. In the *Ack* state, *Tag* waits for confirmation or acknowledgment message. If the received command informs the tag that communication is *done*, it goes to *Finished* and then back to *Start*. On the transition form *Finished* to *Start*, local event *GoToSleep* is broadcasted. This forces the parallel task *Mode* to go from *Work* to *Sleep* mode. Another way to go from *Work* to *Sleep* is on timeout, which is 30 seconds according to the protocol specification.

From the specification we generate the framework for tag application according to the DASH-7 protocol specification. For the full implementation it is necessary to implement the following subroutines (1) *WAKEUP* to detect the low frequency wake up signal, (2) *RECEIVECP* to receive broadcast and point-to-point commands, (3) *LISTENPERIOD* to send a packet message which contains a unique tag identification number to the interrogator in the selected time slot, (4) *ACKPERIOD* to receive an acknowledgement from an interrogator. Those routines should satisfy timing requirements from the general framework model. Each transition can

```
pta

const Next=0; const Rx=1; const Tx=2; const TwoTags=3; const End=4; const
      ThreeCollisions =5;

module powerconscp2
    tagconsum :[0..5]  init  Rx; tagconsumclk : clock;
    i :[0..3]  init  0;

    invariant
        (tagconsum=Next=>tagconsumclk<=1)
        & (tagconsum=Rx=>tagconsumclk<=68)
        & (tagconsum=Tx=>tagconsumclk<=15)
        & (tagconsum=TwoTags=>tagconsumclk<=1)
        & (tagconsum=ThreeCollisions=>tagconsumclk<=1)
    endinvariant

    [] (tagconsum=ThreeCollisions)&(tagconsumclk=1) −> 0.52:(tagconsum'=Next)
        &(tagconsumclk'=0) + 0.48:( tagconsum'=End)&(tagconsumclk'=0);
    [] (tagconsum=Tx)&(i=0)&(tagconsumclk=15) −> 0.52:(tagconsum'=Next)
        &(tagconsumclk'=0) + 0.48:( tagconsum'=End)&(tagconsumclk'=0);
    [] (tagconsum=Next)&(tagconsumclk>=0)&(tagconsumclk<=1) −>
        (tagconsum'=(i<=2)?Rx:End)&(tagconsumclk'=0);
    [] (tagconsum=Tx)&(i!=0)&(tagconsumclk=15) −> 0.96:(tagconsum'=TwoTags)
        &(tagconsumclk'=0) + 0.04:( tagconsum'=ThreeCollisions)&(tagconsumclk'=0);
    [] (tagconsum=Rx)&(i<3)&(tagconsumclk=68) −> (i'=(i+1))&(tagconsum'=Tx)
        &(tagconsumclk'=0);
    [] (tagconsum=TwoTags)&(tagconsumclk=1) −> 0.8:(tagconsum'=End)
        &(tagconsumclk'=0) + 0.2:( tagconsum'=Next)&(tagconsumclk'=0);
endmodule

rewards "cons"
    (tagconsum=Next): 0.2;
    (tagconsum=Rx): 0.5;
    (tagconsum=Tx): 9.2;
    (tagconsum=TwoTags): 0.2;
    (tagconsum=ThreeCollisions): 0.2;
endrewards
```

Fig. 7.18 PRISM code generated by *pState* for model shown in Fig. 7.17

be automatically transferred into assembly code and worst case execution time (WCET) can be calculated in terms of processor cycles. Assembly code generation and calculation of WCET on pCharts is described in [29].

From pCharts in Figs. 7.14 and 7.17, input code for the probabilistic model checker is generated, and from the pCharts in Fig. 7.19, executable code is generated. All generated code is posted on the *pState* web site.

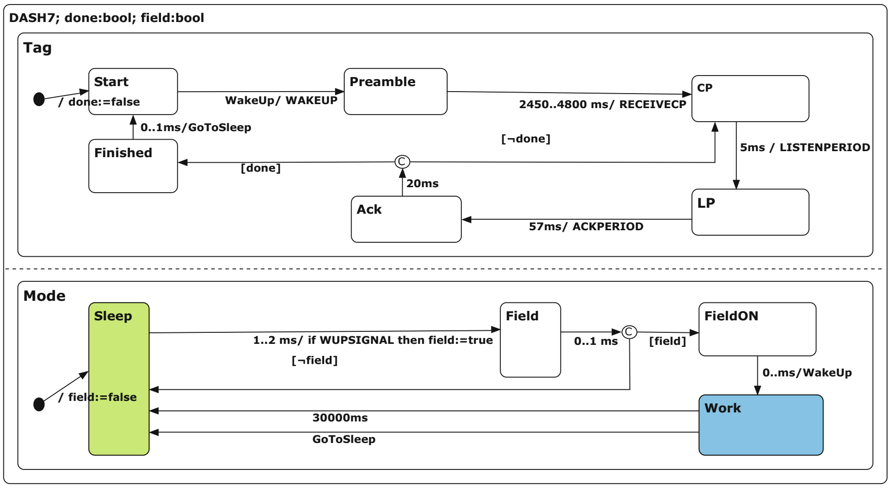


Fig. 7.19 Tag model

7.11 Conclusion

In this chapter, we presented the model based process of system analysis and code generation. From the same model, input code for probabilistic model checker and an executable code are generated. Probabilistic model checker is used for quantitative and qualitative properties analysis. The target for executable code are 8- and 16-bit micro-controllers used in embedded systems. The code is generated in C or assembly language. As a part of assembly executable code generation, we can calculate execution time calculation for each graph transition. Target micro-controllers do not have features like multi-stage pipelines and caches, so execution time in a number of executable *cycles* is actual execution time. The integrated process of model-based analysis and code generation increases an accuracy of analysis and fidelity of generated executable code.

We created a tool, *pState*, for the purpose of *holistic* software design. In addition to executable code generation, the tool is used to verify quantitative properties. This is of practical interest specially for complex embedded systems where not only functional correctness and timing guarantees are relevant, but also quantitative properties, which cannot be analyzed by considering exclusively the software part. The environment has to be considered as well.

In the example of RFID tag working according to DASH-7 ISO/IEC protocol, we show how one tool can be used for system property analysis (collision probability), device property analysis (power consumption), and device code generation.

The goal is an *automated* approach from modelling and analysis to code generation. This can be used to *evaluate* design alternatives and generate *trustworthy* code.

References

1. A. Hinton, M. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, vol. 3920 of *LNCS*, ed. by H. Hermanns, J. Palsberg (Springer, New York, 2006), pp. 441–444
2. D. Harel, Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
3. D. Harel, Statecharts in the making: a personal account. Commun. ACM **52**(3), 67–75 (2009)
4. B. Nokovic, E. Sekerinski, pState: A probabilistic statecharts translator. In *Embedded Computing (MECO), 2013 2nd Mediterranean Conference on*, pp. 29–32, 2013
5. B. Nokovic, E. Sekerinski, Verification and code generation for timed transitions in pCharts. In *Proceedings of the 2014 International C* Conference on Computer Science #38*, C3S2E '14 (ACM, New York, NY, USA, 2014), pp. 3:1–3:10
6. D.N. Jansen, *Extensions of Statecharts with Probability, Time, and Stochastic Timing*. PhD thesis, University of Twente, Enschede, 2003
7. E. Sekerinski, Design verification with state invariants. In *UML 2 Semantics and Applications*, ed. by K. Lano (Wiley, New York, 2009), pp. 317–347.
8. W. Randelshofer, JHotDraw. http://www.randelshofer.ch/oop/jhotdraw/index.html, December 2012
9. R. Eshuis, D.N. Jansen, R. Wieringa, Requirements-level semantics and model checking of object-oriented statecharts. Requir. Eng. **7**(6), 243–263 (2002)
10. D. Harel, A. Naamad, The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**, 293–333 (1996)
11. E. Sekerinski, R. Zurob, iState: A statechart translator. In *UML 2001 - The Unified Modeling Language, 4th International Conference, Lecture Notes in Computer Science 2185*, ed. by M. Gogolla, C. Kobryn, Toronto, Canada, 2001, pp. 376–390
12. E. Mikk, Y. Lakhnech, M. Siegel, G.J. Holzmann, Implementing statecharts in PROMELA/SPIN. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, WIFT '98 (IEEE Computer Society, Washington, DC, USA, 1998), pp. 90–101
13. E. Sekerinski, Verifying statecharts with state invariants. In *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems* (IEEE Computer Society, Washington, DC, USA, 2008), pp. 7–14
14. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Longman Publishing, Boston, MA, 1995)
15. University of Birmingham, Probabilistic symbolic model checker. Website, 2015. http://www.prismmodelchecker.org/
16. M. Fruth, *Formal Methods for the Analysis of Wireless Network Protocols*. PhD thesis, University of Oxford, 2011
17. C. Baier, J.P. Katoen, *Principles of Model Checking* (MIT Press, New York, 2008)
18. G. Norman, D. Parker, J. Sproston, Model checking for probabilistic timed automata. Formal Methods Syst. Des. **43**(2), 164–190 (2013)
19. R. Alur, D.L. Dill, A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
20. C.A.R. Hoare, Communicating sequential processes. Commun. ACM **21**, 666–677 (1978)
21. V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, Automated verification techniques for probabilistic systems. In *Formal Methods for Eternal Networked Software Systems (SFM'11)*, vol. 6659 of *LNCS*, ed. by M. Bernardo, V. Issarny (Springer, New York, 2011), pp. 53–113
22. M. Kwiatkowska, G. Norman, D. Parker, J. Sproston, Performance analysis of probabilistic timed automata using digital clocks. In *Formal Modeling and Analysis of Timed Systems*, vol. 2791 of *Lecture Notes in Computer Science*, ed. by K.G. Larsen, P. Niebert (Springer, Berlin, Heidelberg, 2004), pp. 105–120

23. M. Kwiatkowska, G. Norman, D. Parker, Stochastic games for verification of probabilistic timed automata. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '09, ed. by J. Ouaknine, F.W. Vaandrager (Springer, Berlin, Heidelberg, 2009), pp. 212–227
24. A. Bianco, L. de Alfaro, Model checking of probabalistic and nondeterministic systems. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, ed. by P.S. Thiagarajan (Springer, London, UK, 1995), pp. 499–513
25. N. Wirth, *Compiler construction*. International computer science series (Addison-Wesley, Reading, 1996)
26. IARSystems, IAR visualstate concept guide, 1999
27. DASH7 Alliance, Dash7, 2013. http://www.dash7.org/
28. M. Paun, Posttag PT23 technical specification. Technical report, Lyngsoe Systems, 2006
29. B. Nokovic, E. Sekerinski, Model-based WCET analysis with invariants. In *Proceedings of the 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015), Edinburgh, UK, Sep., 2015*, vol. 72 of *Electronic Communications of the EASST*, ed. by G. Grov, A. Ireland, 2015