Verification and Code Generation for Timed Transitions in pCharts

Bojan Nokovic
McMaster University
Computing and Software Department
1280 Main Street West, Hamilton, Ontario, Canada
nokovib@mcmaster.ca

Emil Sekerinski
McMaster University
Computing and Software Department
1280 Main Street West, Hamilton, Ontario, Canada
emil@mcmaster.ca

ABSTRACT

This paper describes timed transition in pCharts, a variation of statecharts extended with probabilistic transitions, costs/rewards, and state invariants. Timed transitions with nondeterministic and stochastic timing can be used for the specification and analysis of real-time systems. We present a translation scheme for timed transition of pCharts into probabilistic timed automata (PTA) and executable C code, as implemented in our tool pState. To illustrate the development process, we analyze the power consumption of a radio-frequency identification (RFID) tag and generate code for the PIC micro-controller.

Categories and Subject Descriptors

D.1.7 [**Visual Programming**]; D.2.2 [**Design Tools and Techniques**]: State diagrams; D.2.4 [**Software/Program Verification**]: Model checking, Formal methods

General Terms

Design, Verification

Keywords

RFID; Statecharts; pCharts; Probabilistic Model Checker; Invariants; Costs/rewards

1. INTRODUCTION

Visual specification for modelling and code generation has been the subject of intense interest. From a graphical design of a reactive system, executable code can be generated, but these models are commonly insufficient for stating quantitative properties like resource consumption or performance. Such properties can in principle be analyzed by model checkers, for which the system's functionality has to be represented in particular model checker language, usually in the form of guarded commands. Our goal is to create a visual tool for code generation, verification, and quantitative analysis of complex systems.

C3S2E'14, August 03 - 05 2014, Montreal, QC, Canada

http://dx.doi.org/10.1145/2641483.2641522.

Statecharts, a graphical language to describe the behaviour of discrete-state reactive systems extend finite state diagrams by *hierarchy*, *concurrency*, and *communication* [5]. Statecharts are used to specify the behaviour of already built computer systems, as well as the desired behaviour of systems under development. Statecharts are a modelling notation that captures the notion of correctness in terms of the requirements that the system has to meet. Formal methods typically address model correctness as they operate on a purely mathematical formalization. This makes it possible to prevent errors inexpensively at early design stages.

Our extension of classical statecharts with state invariants, probabilistic transitions, timed transition, stochastic timing, state costs and transition costs we call pCharts. Our previous work [15] illustrates how C code is generated and quantitative verification performed on a system specified by pCharts without timed transitions. The algorithm to translate specifications to input code of the probabilistic symbolic model checker PRISM [6] is based on [23].

In this paper, we present the formal syntax of pCharts and an algorithm for the translation of regular and timed transition into executable code and into the input language of PRISM. pCharts without timed transition are translated into a Markov Decision Protocol (MDP) model of PRISM and pCharts with timed transition into a Probabilistic Timed Automata (PTA)[17] model. PTA are finite state automata extended with real-valued clocks and discrete probabilistic choice [2, 10]. Quantitative verification is done over Probabilistic Computational Tree Logic PCTL formulas. MDPs can be used for verification of systems with probabilistic and nondeterministic behaviour, while PTAs can be used to verify systems which in addition to probabilistic and nondeterministic behaviour have real-time constraints [10].

On a PTA model we can analyze properties like (1) the minimum or maximum probability of reaching particular state within given time, which is an *eventually* property, or (2) the maximum expected time to reach some state, or *deadline* property. pCharts can be augmented with quantitative information in the form of *costs* or *rewards* for transition or state. Models with *costs* represent *priced probabilistic timed automata* and can be used to reason about properties like (1) minimum/maximum expected time before some transition will take place, or (2) excepted steps to reach a particular state.

The code generation targets small embedded microcontrollers in which there is no operating system and no support for concurrency.

Statecharts with timed transition constructs (clocks, timed guards, and invariants) have been analyzed with model-checking in [20, 4]. A translation of UML statecharts with timed extension into the parallel composition of flat timed automata of UPPAAL is given in [4]. The flattering algorithm is based on the translation of every hierarchical composite state (XOR and AND) into one flat UPPAAL automaton. PAT (Process Analysis Toolkit) supports the formal verification of hierarchical timed systems specified in the form of Stateflow diagrams [3]. The formalization of UML state machines by an operational semantics presented in [11] is implemented in the vUML tool for state machine model-checking. Timed semantics for StateMate, in which timed events are formalized in terms of clock transition systems over $\mathbb{N}$ is given in [20]. Modelling based on a set of UML diagrams, called MADES UML diagrams, for reactive, time critical embedded systems is presented in [1]. The formal semantics is presented using a metric temporal logic. With a prototypical verification tool, charts are translated into the input language of the Zot [21] model checker.None of these formalisms for hierarchical timed systems support probabilistic transition and quantitative property verification with costs attached to states and transitions.

The original statecharts paper treats time restrictions using implicit timers [5]: the expression *timeout(event, number)* specifies an event that occurs when specified *number* of time units have elapsed from the occurrence of *event*. The notation *timeout(entered state, bound)* is used to indicate that a state comes with a bound, where an *entered state* is the source of transition, and *bound* specifies time units. In UML statecharts [18], a *time event* specifies an *absolute* point in time a point in time or *relative* to some other point in time. Telative and absolute time triggers are specified with the keywords *after* and *at*, respectively, followed by a time value. The main differences to the original statecharts semantics [5] is the introduction of absolute time in UML. Timed transitions in most of todays statechart tools, like *Stateflow* with Simulink, Eclipse *Papyrus*, *Yakindu Statechart Tools*, *IAR Visual State*, are specified according to the UML statecharts notation. None of these tools allows direct specification of stochastic timed transitions or costs. The semantic of pCharts as implemented in *pState* [14] is according to [24], in which the translation scheme is characterized as *event-centric* where the main structure of the code is that of events, i.e. events are procedures. The translation schemes of other tools like IBM Rational Rhapsody [7] can be characterized as *state-centric*, because main structure of the code are classes for states and events are data values that are passed around. As already explored with *iState* [24], the event-centric approach is suitable for those kind of reactive systems where events are processed quickly enough so that no queuing of events is necessary and where blocking of events is undesirable. This semantic is close to [12].

This paper is organized as follows. The next section introduces probabilistic guarded commands, the target of the code generation algorithm. Section 3 presents the translation scheme for regular and timed transitions. In the Section 4 we formally define the pCharts structure, and in 5 we described algorithms for the code generation of regular and timed transitions. In Section 6 the example of an RFID tag illustrates the generation of a PTA model and verification of properties like power consumption or probability to reach a particular state. We also show how from a selected part of this pChart executable code for an embedded system can be generated.

2. PRELIMINARIES

Statements are inductively constructed as follows. Assuming that b is a Boolean expression, xv is a list of unique variables, ev is a list of expressions of the same length as xv, Q, R are statements, pv is a list of real expressions, and QV is a list of statements of the same length as pv, the set *Statement* consists of:

skip	the empty statement, always enabled
stop	the always blocking statement
$xv := ev$	the multiple assignment, always enabled
$g \to R$	guarded command, enabled if g holds and R is enabled
$Q \,[]\, R$	nondeterministic choice between Q and R, enabled if either one is
$Q /\!/ R$	prioritizing choice, Q if Q is enabled, else R
$Q \parallel R$	independent (parallel) composition of Q and R, provided the assigned variables are disjoint, enabled if both Q and R are
$pv : QV$	probabilistic choice among QV with probability according to pv, provided $\sum pv = 1$

Thus a statement is either *blocking* or *enabled*. The independent (or parallel) composition $Q \parallel R$ is a generalization of multiple assignment in the sense that $(x, y := e, f) = (x := e \parallel y := f)$. It is well-defined if the variables assigned by Q and R are disjoint. For the variables accessed by Q and R their initial values are taken. Thus there is no interleaving. The probabilistic choice is more commonly written as $p_1 : Q_1 \oplus \cdots \oplus p_n : Q_n$ or, using comprehension notation, $\oplus\, i \in I \,.\, p_i : Q_i$. As both nondeterministic choice and independent composition are associative, we write simply $Q_1 \,[]\, \cdots \,[]\, Q_n$ and $Q_1 \parallel \cdots \parallel Q_n$ without parenthesis, or, using comprehension notation, $[]\, i \in I \,.\, p_i : Q_i$ and $\parallel i \in I \,.\, p_i : Q_i$. The *conditional statement* is defined in terms of above statements:

$$\begin{array}{lcl} \textsf{if } b \textsf{ then } Q & \widehat{=} & (b \to Q) \,[]\, (\neg b \to \textsf{skip}) \\ \textsf{if } b \textsf{ then } Q \textsf{ else } R & \widehat{=} & (b \to Q) \,[]\, (\neg b \to R) \end{array}$$

Probabilistic guarded commands can be defined by predicate transformers [13]; for our purposes, a simpler definition by relations between the initial state and *distributions* over the final state is sufficient, i.e. as functions of type $\Gamma \to \mathcal{PD}\Gamma$, where $\mathcal{D}\Gamma = \Gamma \to [0,1]$ such that $\sum d = 1$ for all $d \in \mathcal{D}\Gamma$, distribution d is not 0 for finitely many states of Γ, and $\mathcal{P}$ is the powerset operator. Intuitively, a statement first makes an arbitrary nondeterministic choice among distributions and then a probabilistic choice according to that distribution. The independent composition leads to a cross product of the state space, i.e. if $Q : \Gamma \to \mathcal{PD}\Gamma$ and $R : \Delta \to \mathcal{PD}\Delta$ then $Q \parallel R : (\Gamma \times \Delta) \to \mathcal{PD}(\Gamma \times \Delta)$. The following properties of statements will be used, where b, c are Boolean expressions, P, Q, R are statements, xv, yv are lists of variables, and ev, fv are lists expressions of same

length as xv, yv; their proofs are left out for brevity:

$$xv := ev \parallel yv := fv = xv, yv := ev, fv \quad (1)$$
$$b \to c \to Q = (b \wedge c) \to Q \quad (2)$$
$$b \to (P \,[]\, Q) = (b \to P) \,[]\, (b \to Q) \quad (3)$$
$$(P \,[]\, Q) \parallel R = (P \parallel R) \,[]\, (Q \parallel R) \quad (4)$$
$$(b \to Q) \parallel R = b \to (Q \parallel R) \quad (5)$$

Above laws are known for standard (non-probabilistic) statements and continue to hold for probabilistic statements. We also need following laws involving probabilistic choice:

$$(p : P \oplus q : Q) \parallel R = p : (P \parallel R) \oplus q : (Q \parallel R) \quad (6)$$

$$p : (q_1 : Q_1 \oplus q_2 : Q_2) \oplus r : R = p \times q_1 : Q_1 \oplus p \times q_2 : Q_2 \oplus r : R \quad (7)$$

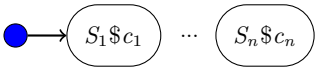
Probabilistic choice can be distributed inside guarded nondeterministic choice provided that one of the guards is true:

$$p : (b_1 \to P_1 \,[]\, b_2 \to P_2) \oplus q : Q = b_1 \to (p : P_1 \oplus q : Q) \,[]\, b_2 \to (p : P_2 \oplus q : Q) \quad \text{if } b_1 \vee b_2 \quad (8)$$

To see why the condition is necessary, assume that $b_1 = b_2 = \mathsf{false}$: the left side blocks with probability p and choses T with probability q, but the right side always blocks.

3. ELEMENTS OF PCHARTS

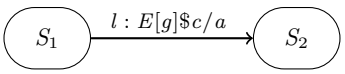
Basic Charts. pCharts consists of a finite number of states and transitions between those state. Upon an event, a system may evolve from one state into another. States are symbolized by (rounded) boxes. We represent the states of a state diagram as a variable of an enumerated set type [22], to which we here allow a cost, a non-negative real number, to be associated, using the notation $\$c$. The expression $in(S)$ tests if the chart is in state S and the statement $goto(S)$ moves the chart to state S:



$r : \{S_1..S_n\}$

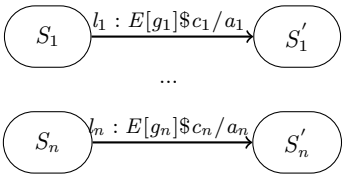
$in(S_i) = r = S_i$
$goto(S_i) = r := S_i$
$cost(S_i) = c_i$

Upon an event, a transition takes only place if in the current state there is a transition on this event. Otherwise, the event is ignored. Suppose only one transition for event E exists. Boolean expression guard $[g]$, cost specifications $\$c$, and action expression $/a$ are optional. Actions are assumed to be instantaneous. The operation $op(E)$ of event E returns a set of pairs $g \to S\ \$c$, each consisting of a guarded command $g \to S$ and cost c:



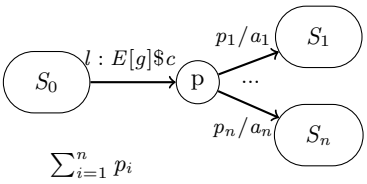
$op(E) =$
$in(S_1) \wedge g \to$
$goto(S_2) \parallel a\ \$c$

In case there are several transitions labelled with E, the one starting from the current state is taken, if any transition is taken at all. States $S_1^{'}, ..S_n^{'}$ do not have to be distinct:



$op(E) =$
$in(S_1) \wedge g_1 \to$
$goto(S_1^{'}) \parallel a_1\ \$c_1\ []$
$\ldots []$
$in(S_n) \wedge g_n \to$
$goto(S_n^{'}) \parallel a_n\ \$c_n$

In case of a probabilistic transition, each alternative consists of a probability $p_i \in [0..1]$, an optional body a_i, where each a_i is a statement that may include broadcasts.

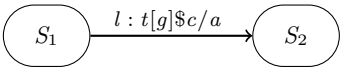


$op(E) =$
$in(S_0) \wedge g \to$
$p_1 : goto(S_1) \parallel a_1 \oplus$
$\ldots \oplus$
$p_n : goto(S_n) \parallel a_n\ \$c$

Each timed transition has a unique label, written l below. We introduce the shorthand $g \xrightarrow{t} S$ for a guarded command executed at time t.

$$t \in n.. \mid n_1..n_2 \mid exp(n) \mid unif(n)$$

In a PTA this involves a clock variable whose value is tested in the guard and whose value is reset in the body of the guarded command:



$op(l) =$
$in(S_1) \wedge g \xrightarrow{t}$
$goto(S_2) \parallel a\ \$c$

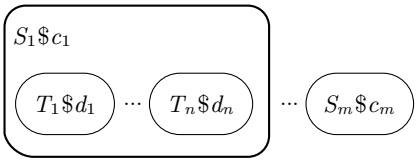
For timed events we allow equivalent notations: $after(t)$ is the same as $t..$, with the meaning that the transition is taken any time after t, and $between(t_1, t_2)$ is the same as $t_1..t_2$, with the meaning that the transition is taken any time between t_1 and t_2. Time is specified in time units, which are milliseconds (ms), seconds (s), hours (h), or days (d); pState normalizes the units to the smallest one used in a chart. As a special case, we write simply t if the transition is to be taken at exactly time t, formally $between(t, t)$.

In the specification of the environment we allow two stochastic transitions: *exponential* and *uniform* as introduced in [8, 9]. In the transition $exp(t)$ the delay is defined by an exponential distribution with an *average* duration of t time units. The timed transition $unif(t_1,\ t_2,\ d)$ indicates a uniformly distributed delay with the given minimum duration of t_1 and maximum duration t_2 time units. The uniform distribution takes an optional parameter d for the step, as the distribution is approximated discreetly.

Stochastic time events can be used only in the specification from which code for model checker is generated, but at this

moment we do not generate executable code from stochastic representation. The pecification of *absolute* timed events, like an event to happen at *May. 1st 2015, Noon* is not supported.

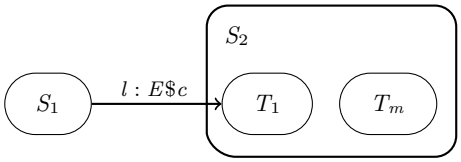
Hierarchy. Composite states can have substates or children. If the system is in a state with substates, it is also in exactly one of those substates. Conversely, if a system is in a substate of a superstate, it is also in that superstate. In statecharts, a superstate with substates is drawn by nesting.



$r : \{S_1 .. S_m\}$,
$s : \{T_1 .. T_n\}$

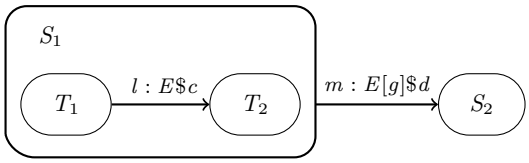
$cost(S_i) = c_i$
$cost(T_j) = d_j$
$goto(S_i) = r := S_i$
$goto(T_j) = s := T_j$

When entering a superstate, the substate to be entered has to be specified as well. In statecharts this is expressed by letting the transition arrow point to a specific substate:



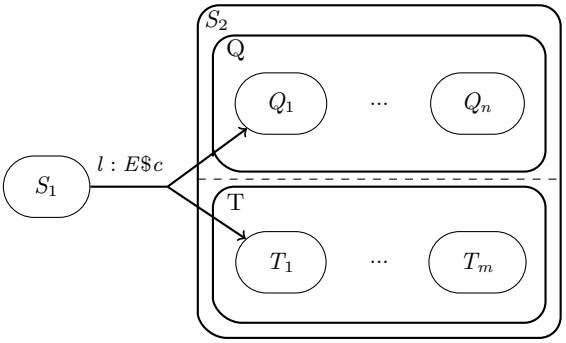
$$op(E) = in(S_1) \rightarrow goto(S_2) \parallel goto(T_1) \ \$c$$

If we have two transitions on the same event E, the transition going out form superstate will have higher priority. If guard g is *true*, transition from S_1 to S_2 will happen, otherwise, transition form T_1 to T_2 will happen. Without condition g on the transition from superstate, the transition inside superstate would never happen.



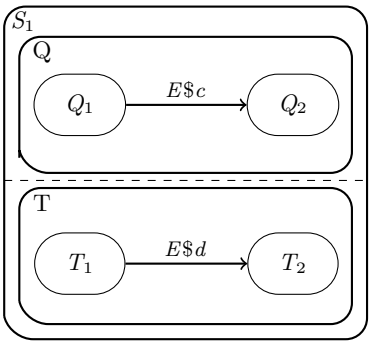
$$op(E) = in(S_1) \wedge g \rightarrow goto(S_2)) \ \$d \ [] \\ in(S_1) \wedge in(T_1) \wedge \neg g \rightarrow goto(T_2) \ \$c$$

Concurrency. Concurrency is expressed by orthogonality: a system can be in two independent states simultaneously. This is drawn by splitting a state with a dashed line into independent substates, each of which consists of a number of states in turn. A state with concurrent substates is entered by a fork into states in each of the concurrent substates. This corresponds to setting the variables for all the concurrent states:



$r : \{S_1, S_2\},\ q : \{Q_1 .. Q_m\},\ t : \{T_1 .. T_n\}$
$op(E) = in(S_1) \rightarrow goto(S_2) \parallel goto(Q_1) \parallel goto(T_1) \ \$c$

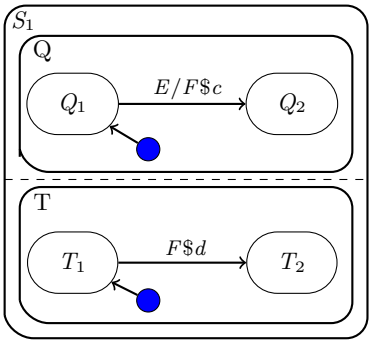
Two concurrent states may have transitions on the same event. In case this event occurs, these transitions are taken simultaneously. Parallel composition of the transitions has implications on the global variables which can occur in the conditions and the actions, a variable can only be assigned by one action:



$$op(E) = in(S_1) \wedge in(Q_1) \wedge in(T_1)) \rightarrow \\ goto(Q_2) \parallel goto(T_2) \ \$(c + d) \ [] \\ in(S_1) \wedge in(Q_1) \wedge \neg in(T_1) \rightarrow goto(Q_2) \ \$c, [] \\ in(S_1) \wedge in(T_1) \wedge \neg in(Q_1) \rightarrow goto(T_2) \ \$d$$

Communication. Communication is possible between concurrent states in three ways [22]: First, concurrent states can communicate by global variables. These can be set in actions and read in actions and conditions, following the rules for variables given earlier. Secondly, the condition or the action of a transition may depend on the current substate of a concurrent state. Thirdly, concurrent states can communicate by broadcasting events. On a broadcast of an event, all concurrent states react simultaneously. Events are either generated internally through a broadcast or externally by the environment. Broadcasting an event F corresponds to calling $op(F)$. In the below, the initial configuration is (Q_1, T_1). On external event E, Q changes from Q_1 to Q_2,

and broadcasting event F changes T from T_1 to T_2. After event E chart is in (Q_2, T_2):



$$op(E) = in(S_1) \wedge in(Q_1) \rightarrow goto(Q_2) \parallel op(F) \ \ \$c$$
$$op(F) = in(S_1) \wedge in(T_1) \rightarrow goto(T_2) \ \ \$d$$

4. PCHART STRUCTURE

A *pChart* is a structure with states, transitions, expressions, types, and statements that are defined in turn.

States. We assume that *Variable* and *Event* are variable and event names, that *Basic*, *AND*, *XOR* are finite, mutually disjoint sets of state names, and let $Composite = AND \cup XOR$ be the set of *composite* states and $State = Basic \cup Composite$ be the set of all states:

$Root \in XOR$	root state
$parent : State - \{Root\} \rightarrow State$	parent of each state except the root state
$var : State \rightarrow (Variable \nrightarrow Type)$	variables declared local to a state
$ev : State \rightarrow \mathcal{P}Event$	events declared local to a state
$inv : State \rightarrow Expr$	invariant attached to a state
$cost : State \rightarrow Expr$	cost of being in a state

All states form a tree that is rooted in *Root*, formally $Root \in parent^*[\{s\}]$ for any $s \in State$, where r^* is the transitive and reflexive closure of relation r and $r[S]$ is the image of the set S under r. We let the relation *children* be the inverse of *parent*, i.e. $children = parent^{-1}$. Basic states don't have children, $children[Basic] = \{\}$. The children of an *AND* state are said to be *concurrent*, the children of an *XOR* state are said to be *exclusive*. The children of an *AND* state must be *XOR* states, $children[AND] \subseteq XOR$. The variables of *Root* are the *global variables*, the events of *Root* are the *global events*, and the invariant of *Root* is the *global invariant*.

Transitions. For the transitions of a chart we assume that *Transition* is a finite set of transitions and *Alternative* is a finite set of probabilistic alternatives:

$source : Transition \rightarrow \mathcal{P}State$	set of source states of a transition
$event : Transition \rightarrow Event \cup Time$	event of a regular or timed transition
$guard : Transition \rightarrow Expr$	guard of a transition
$cost : Transition \rightarrow Expr$	cost of taking transition
$alt : Transition \rightarrow \mathcal{P}Alternative$	set of probabilistic alternatives of a transition
$prob : Alternative \rightarrow Expr$	probability of an alternative
$target : Alternative \rightarrow \mathcal{P}State$	set of target states of an alternative
$body : Alternative \rightarrow Statement$	body of an alternative
$default : XOR \nrightarrow Alternative$	default alternative of an *XOR* state

The sets *source, alt, target* are non-empty. The state *Root* must not be the source or target of any transition, $Root \notin source(t)$ and $Root \notin target(d)$ for any $t \in Transition$ and $d \in alt(t)$. The default transition of an *XOR* state s, if defined, must go to a descendant of s, i.e. $target(default(s)) \subseteq children^+[\{s\}]$, where r^+ is the transitive closure of relation r.

Time is

$$Time \ \ \widehat{=} \ \ between(\mathbb{R}^+_{\geq 0}, \mathbb{R}^\infty_{\geq 0}) \ \mid exp(\mathbb{R}^+_{\geq 0}) \ \mid unif(\mathbb{R}^+_{\geq 0})$$

and timed transition can be *regular* or *stochastic* as illustrated later on.

The *closest common ancestor* $cca(ss)$ of a set ss of states is the state that is a proper ancestor of each state in ss and all other common ancestors are also its ancestor. We write $x\, r\, y$ for the pair (x, y) belonging to relation r.

$$c = cca(ss) \equiv c \in parent^+[ss] \wedge (\forall\, a \in State \,.\, a \in parent^+[ss] \Rightarrow a\ parent^*\ c)$$

The closest common ancestor exists and is unique for any non-empty set of states that does not include the root state. States r, s are *orthogonal*, written $r \perp s$, if their closest common ancestor is an *AND* state and neither is an ancestor of the other. A set ss of states is called orthogonal, written $\perp ss$, if every pair of distinct states of ss is orthogonal. All source states of a transition must be orthogonal, $\perp source(t)$ for all $t \in Transition$ and targets of an alternative must be orthogonal, $\perp target(d)$ for all $d \in Alternative$. The *scope* of a transition is the state closest to the root through which the transition passes.

$$scope(t) \ \ \widehat{=} \ \ cca(source(t) \cup (\bigcup d \in alt(t) \,.\, target(d)))$$

The *path* from state s to a set ss of descendants of s is the set of those states that are descendants of s and ancestors of states in ss, excluding s but including the states of ss.

$$path(s, ss) \ \ \widehat{=} \ \ children^+[\{s\}] \cap parent^*[ss]$$

The states *exited* by a transition are all the states on the path from the scope of the transition to its sources. The

states *explicitly entered* by a transition t are all the states on the path from the scope of the transition to a specific probabilistic alternative; if an alternative targets a descendant of an *AND* state, then other states may be implicitly entered as well. In general, $entered(s, d)$ for state s and alternative d targeting a descendant of s are all states on the path from s to the target of d.

$$\begin{array}{lcl} exited(t) & \widehat{=} & path(scope(t), source(t)) \\ entered(s, d) & \widehat{=} & path(s, target(d)) \end{array}$$

Given a state set ss, the *implicit children* are those children of *AND* states of ss that are not in ss. As children of *AND* states are *XOR* states, all implicit children are *XOR* states. If a chart enters ss, it also enters all its implicit children.

$$imp(ss) \quad \widehat{=} \quad children[ss \cap AND] - ss$$

The *completion* of an alternative d starting at state s and targeting descendants of s is the set of all alternatives that are taken when d is taken: it adds to d the default alternatives of *XOR* explicit targets of d and all default alternatives of implicitly entered states.

$$\begin{array}{lcl} comp(s, d) & \widehat{=} & \{(s, d)\} \cup (\bigcup r \in (target(d) \cap XOR) \cup \\ & & imp(entered(s, d)) \;.\; comp(r, default(r))) \end{array}$$

Certain *XOR* states are required to have a default initial state: a default alternative must be defined for the root state, $Root \in \mathsf{dom}\ default$, and any *XOR* state that is the target of some alternative or that is being implicitly entered as it has an *AND* ancestor that is being entered. Formally this means that *default* must be defined such that $comp(scope(t), d)$ is well-defined for all $t \in Transition$ and $d \in alt(t)$.

Expressions. A *chart expression* is composed from program variables, state tests $\mathsf{in}\, S$, where S is any state except *Root*, and functions *fn* applied to zero or more arguments:

$$Expr ::= Variable \mid \mathsf{in}\, S_1, \ldots, S_m \mid fn(Expr_1, \ldots, Expr_n)$$

A function without arguments must be an integer, Boolean, or real constant. A function can also be one of the unary operators $\neg e$, $-e$, $\lfloor e \rfloor$, $\lceil e \rceil$, one of the binary arithmetic, Boolean, and relational operators $e + e$, $e - e$, $e * f$, $e\ \mathsf{div}\ e$, $e\ \mathsf{mod}\ e$, e/f, $e \wedge f$, $e \vee f$, $e \Rightarrow f$, $e \Leftarrow f$, $e = f$, $e \neq f$, $e < f$, $e \leq f$, $e > f$, $e \geq f$, or the logarithm, minimum, or maximum function, $\log(e_1, e_2)$, $\mathsf{min}\ (e_1, \ldots, e_m)$, $\mathsf{max}\ (e_1, \ldots, e_m)$.

Type. A *chart type* is either an integer subrange, Boolean, or real.

$$Type \quad ::= \quad \mathsf{integer}\,..\,\mathsf{integer} \mid \mathsf{bool} \mid \mathsf{real}$$

The partial function $type : Expr \nrightarrow Type$ determines the type of an expression. The type of a variable is determined by its declaration; the scope rules of languages with nested structures apply here to nested states. If variable v occurs in the body of a transition with scope S, then $decl(v, S)$ is the closest ancestor, or S itself, where v is declared:

$$decl(v, S) \quad \widehat{=} \quad \mathsf{if}\ v \in \mathsf{dom}\ var(S)\ \mathsf{then}\ S\ \mathsf{else}\ parent(S)$$

Thus, if $v \in Variable$ occurs in state S, its type is:

$$type(v) \quad \widehat{=} \quad var(decl(v, S), v)$$

While expressions can be of any type, variables can only be of subrange or Boolean type. An expression e is *well-typed* if $type(e)$ is defined. Transition guards, state invariants, and conditions of conditional statements have to be of Boolean type. Probabilities of alternatives, costs of states, and costs of transitions have to be of type real.

Statements. A *chart statement* is either skip, a multiple assignment, a parallel composition, or a conditional. In addition a chart statement can be a *broadcast* of event $E \in Event$, simply written as E. All assignment statements have to be *type-correct*, i.e. the types of the left and right hand side have to agree, and all broadcast statements have to be *conflict-free*, in a sense to be defined shortly.

Conventions. In charts, if a transition guard $[g]$ is missing, it is assumed to be true. If a transition $/B$ body is missing, it is assumed to be skip. If there is only one probabilistic alternative, its probability of 1 is left out.

5. TRANSLATION

We present two translations of charts to statements, *op* which generates guarded commands for regular events and *top* which generates guarded commands for timed events.

State Model. For representing the configuration (or "state") of a chart, we use a model that makes it easy to express independent updates of concurrent states and state tests of any state in the hierarchy, and can directly be mapped to a programming language [22]. For every *XOR* state s, including *Root*, a variable $lc(s)$, ranging over $uc(c)$ for every child c of s, is declared. We interpret $lc(s)$ and $uc(s)$ to be the state s starting with a lowercase or an uppercase letter. (We assume that these variables and their values are distinct from variables declared in the chart.) This model allows to define the *state test* and *state assignment* for any state s that is a child of an *XOR* state by inspecting and assigning the variable for that state:

$$\begin{array}{lcl} test(s) & \widehat{=} & lc(parent(s)) = uc(s) \\ assign(s) & \widehat{=} & lc(parent(s)) := uc(s) \end{array}$$

Manipulation of configurations is expressed in terms of *test* and *assign*. The predicate $in(ss)$ tests whether the current configuration is in the set ss; similarly $goto(ss)$ sets the current configuration to ss.

$$\begin{array}{lcl} in(ss) & \widehat{=} & \forall\, s \in ss \cap children[XOR] \;.\; test(s) \\ goto(ss) & \widehat{=} & \|\ s \in ss \cap children[XOR] \;.\; assign(s) \end{array}$$

As special cases we have $in(\{\}) = \mathsf{true}$ and $goto(\{\}) = \mathsf{skip}$. The statement $goto(ss)$ is well-defined if the states of ss are not exclusive.

Event Translation. The *trigger* of a transition t is true if the transition guard is true and if the chart is in all source states of the transition. The *effect* of a transition t is a probabilistic choice among its alternatives: each alternative is completed and for each completion, the body of the completion is executed and the system moves to all entered states

of the completion; in addition, clocks for timed events are reset, to be explained later.

$$\begin{array}{lcl} \mathit{trigger}(t) & \widehat{=} & \mathit{in}(\mathit{exited}(t)) \wedge \mathit{guard}(t) \\ \mathit{effect}(t) & \widehat{=} & \oplus\, c \in \mathit{alt}(t)\ .\ \mathit{prob}(c) : \\ & & \quad (\|\ (s,d) \in \mathit{comp}(t,c)\ .\ \mathit{body}(d)\ \| \\ & & \quad\ \mathit{goto}(\mathit{entered}(s,d))\ \| \\ & & \quad\ \mathit{reset}(\mathit{entered}(s,d))) \end{array}$$

The *operation* of an event E is a statement that captures the joint meaning of all transitions in a chart on E. For brevity, let $\mathit{Trans}(E,s)$ stand for the set of transitions on event E with scope s:

$$\mathit{Trans}(E,s) \quad \widehat{=} \quad \{t \mid \mathit{event}(t) = E \wedge \mathit{scope}(t) = s\}$$

The function $\mathit{op}(E)$ recursively visits all transitions on E, starting with those on the outermost scope, *Root*. In case there is a choice between transitions with the same scope, one is selected arbitrarily. In case there is a choice between transitions on different scopes, transitions on outer scopes are given priority. All transitions on the same event in concurrent states are taken in parallel. Of all transitions in an exclusive state, at most one can be taken. Following statecharts, the response to an event on which no transition can take place is to do nothing, i.e. skip, rather than to block. A transition may also broadcast an event, say F, either explicitly or implicitly in one of the alternatives of its completion; any transition taken on F is taken jointly with those on E and if no transition on F can be taken, F behaves as skip. Thus the meaning of broadcasting F is that of $\mathit{op}(F)$. We write $S[F \backslash T]$ for replacing event F by T in statement S:

$$\begin{array}{lcl} \mathit{op}(E) & \widehat{=} & \mathit{scopeop}(E, \mathit{Root}) \\ \mathit{scopeop}(E,s) & \widehat{=} & ([]\ t \in \mathit{Trans}(E,s)\ .\ \mathit{trigger}(t) \rightarrow \\ & & \quad \mathit{effect}(t)[F \backslash \mathit{op}(F)])\ /\!\!/\ \mathit{childop}(E,s) \\ \mathit{childop}(E,s) & \widehat{=} & \mathsf{case}\ s\ \mathsf{of} \\ & & \quad \mathit{XOR} : []\ c \in \mathit{children}[\{s\}] - \\ & & \quad\ \mathit{Basic}\ .\ \mathit{test}(c) \rightarrow \mathit{scopeop}(E,c)\ /\!\!/\ \mathsf{skip} \\ & & \quad \mathit{AND} : \|\ c \in \mathit{children}[\{s\}] - \\ & & \quad\ \mathit{Basic}\ .\ \mathit{scopeop}(E,c) \\ & & \mathsf{end} \end{array}$$

The above definition generalizes to the case when more than one event is broadcast. The function $\mathit{childop}(E,s)$ is defined only if s is an *XOR* or *AND* state, which the mutually recursive definition respects at each call.

For an operation to be *conflict-free*, there must not be conflicting multiple assignments to the same variable. Such a conflict may appear if the body of a transition assigns to, say x, and broadcasts an event that also assignes to x. As chart configurations are modified by assignments to variables, this implies that no event can be transitively broadcast twice. By extension, event broadcasting cannot be cyclic [25].

For pCharts with timed transition, set of timed transitions with scope s is defined as:

$$\mathit{Trans}(s) \quad \widehat{=} \quad \{t \mid \mathit{event}(t) \in \mathit{Time} \wedge \mathit{scope}(t) = s\}$$

While by *op(E)* we generate the code for the single event E, by *top()* we generate the code for all timed transitions of a chart. As timed transitions on outer scopes take priority over timed transition in inner scopes, generation starts with timed transitions with scope *Root* and then recursively descends to transitions with inner scopes, i.e.

$$\begin{array}{lcl} \mathit{top}() & \widehat{=} & \mathit{tscopeop}(\mathit{Root}) \\ \mathit{tscopeop}(s) & \widehat{=} & ([]\ t \in \mathit{Trans}(s)\ .\ \mathit{ttrigger}(t) \rightarrow \\ & & \quad \mathit{teffect}(t)[F \backslash \mathit{op}(F)])\ /\!\!/\ \mathit{tchildop}(s) \\ \mathit{tchildop}(s) & \widehat{=} & \mathsf{case}\ s\ \mathsf{of} \\ & & \quad \mathit{XOR} : []\ c \in \mathit{children}[\{s\}] - \\ & & \quad\ \mathit{Basic}\ .\ \mathit{test}(c) \rightarrow \mathit{tscopeop}(c)\ /\!\!/\ \mathsf{skip} \\ & & \quad \mathit{AND} : \|\ c \in \mathit{children}[\{s\}] - \\ & & \quad\ \mathit{Basic}\ .\ \mathit{tscopeop}(c) \\ & & \mathsf{end} \end{array}$$

where:

$$\mathit{ttrigger}(t) \quad \widehat{=} \quad \mathit{in}(\mathit{exited}(t)) \wedge \mathit{guard}(t) \wedge \mathit{timeout}(t)$$

The scope s of each transition is an XOR state and with each scope we associate a clock variable, $\mathit{clock}(s)$. A timed transition t with scope s is scheduled by $\mathit{clock}(s)$. If $\mathit{event}(t)$ is $\mathit{between}(l,u)$, then $\mathit{timeout}(t)$ is $l \leq \mathit{clock}(\mathit{scope}(t))$ and $\mathit{clock}(\mathit{scope}(t)) \leq u$ becomes a timed invariant of the resulting PTA. We now can define the statement $\mathit{reset}(ss)$ for a set ss of states: for each timed transition t leaving some $s \in ss$, the clock of $\mathit{scope}(s)$ is reset, $\mathit{clock}(\mathit{scope}(s)) := 0$, such that each t will be scheduled correctly.

A timed transition may broadcast an event, say F, but a timed transition can not be broadcasted itself. If event F is broadcasted, it has to be an untimed event since transition taken on F is taken jointly with those on timed event.

PRISM allows only a flat collection of guarded commands of the form $b_1 \rightarrow S_1\ []\ \cdots\ []\ b_m \rightarrow S_m$, where each S_i is of the form $p_1 : A_1 \oplus \cdots \oplus p_m : A_n$ and each A_i is a multiple assignment statement. We call this the *normal form* of an operation. For generating a normal form, first $\mathit{scopeop}(E,s)$ is equivalently expressed by making the guard explicit instead of writing "$/\!\!/$ skip". With abbreviations

$$\begin{array}{lcl} \mathit{TE} & \widehat{=} & []\ t \in \mathit{Trans}(E,s)\ .\ \mathit{trigger}(t) \rightarrow \\ & & \quad \mathit{effect}(t)[F \backslash \mathit{op}(F)] \\ \mathit{TT} & \widehat{=} & \forall\, t \in \mathit{Trans}(E,s)\ .\ \neg \mathit{trigger}(t) \end{array}$$

we have:

$$\mathit{scopeop}(E,s) \quad = \quad \mathit{TE}\ []\ \mathit{TT} \rightarrow \mathit{childop}(E,s)$$

From the definitions we observe for $\mathit{effect}(t)$ in TE that $\mathit{goto}(\mathit{entered}(s,d))$ is a parallel composition of multiple assignments. If $\mathit{body}(d)$ also consists only of multiple assignments (or skip), then we can use (1) to transform $\mathit{effect}(t)$ into a single multiple assignment as needed for the normal form. If $\mathit{body}(d)$ contains conditionals, which by definition are of the form $(b \rightarrow Q)\ []\ (c \rightarrow R)$, then first by (4) and (5) the guard and the choice can be "moved out", and on the "top level" merged by (3) and (2) with other nondeterministic choices of TE. We note that each conditional statement leads to two "top level" choices. If $\mathit{effect}(t)$ contains a broadcast of an event, say F, then that has to be replaced by $\mathit{op}(F)$. We assume that $\mathit{op}(F)$ is in normal form and show how to transform TE to normal form. More specifically, suppose that $\mathit{op}(E)$ and $\mathit{op}(F)$ are of the form:

$$\begin{array}{lcl} \mathit{op}(F) & = & b_1 \rightarrow P_1\ []\ b_2 \rightarrow P_2 \\ \mathit{op}(E) & = & c_1 \rightarrow (p : \mathit{op}(F) \oplus q : Q)\ []\ c_2 \rightarrow R \end{array}$$

We note that any operation is always enabled as $P\ /\!\!/\ Q$ is enabled if either P or Q is, so $P\ /\!\!/$ skip is always enabled.

As $op(F)$ is always enabled $b_1 \vee b_2$ must hold and we can use (8) to transform $op(E)$ to

$$\begin{aligned} op(E) \quad = \quad & c_1 \to (b_1 \to (p : P_1 \oplus q : Q) \,[]\, \\ & b_2 \to (p : P_2 \oplus q : Q)) \,[]\, c_2 \to R \end{aligned}$$

which can then be brought into normal form by (3) and (2). This generalizes to more than two probabilistic alternatives and nondeterministic choices accordingly.

Now we show inductively that the result of $scopeop(E, s)$ can be transformed to normal form, assuming that recursive calls are returning a normal form. Considering $TT \to childop(E, s)$ as above, there are two cases. If s is an *XOR* state, we use (3) and (2) to simplify $childop(E, s)$. That is, assuming $childop(E, s)$ is of the form $b_1 \to P_1 \,[]\, b_2 \to P_2$, where P_1, P_2 are in normal form (and b_1, b_2 are state tests) we obtain:

$$TT \to childop(E, s) \quad = \quad TT \wedge b_1 \to P_1 \,[]\, TT \wedge b_2 \to P_2$$

As P_1, P_2 are in normal form, (3) and (2) can be used again to flatten the whole structure. If s is an *AND* state, we use (4) and (3) to simplify $childop(E, s)$. That is, assuming $childop(E, s)$ is of the form $(b_1 \to P_1 \,[]\, b_2 \to P_2) \parallel (c_1 \to Q_1 \,[]\, c_2 \to Q_2)$, resulting the normal form returned by $scopeop(E, r)$, we obtain:

$$\begin{aligned} TT \to childop(E, s) \quad = \quad & TT \wedge b_1 \wedge c_1 \to (P_1 \parallel Q_1) \,[]\, \\ & TT \wedge b_1 \wedge c_2 \to (P_1 \parallel Q_2) \,[]\, \\ & TT \wedge b_2 \wedge c_1 \to (P_2 \parallel Q_1) \,[]\, \\ & TT \wedge b_2 \wedge c_2 \to (P_2 \parallel Q_2) \end{aligned}$$

Considering now P_1 to be $r_1 : R_1 \oplus r_2 : R_2$ and Q_1 to be $s_1 : S_1 \oplus s_2 : S_2$, where R_1, R_2, S_1, S_2 are multiple assignment statements, we use (6) for "moving out" the probabilistic choice in $P_1 \parallel Q_1$ and then use (7) to flatten the nested probabilistic alternatives:

$$\begin{aligned} P_1 \parallel Q_1 \quad = \quad & r_1 \times s_1 : (R_1 \parallel S_1) \oplus r_1 \times s_2 : (R_1 \parallel S_2) \oplus \\ & r_2 \times s_1 : (R_2 \parallel S_1) \oplus r_2 \times s_2 : (R_2 \parallel S_2) \end{aligned}$$

Repeating this process brings then $TT \to childop(E, s)$ in normal form and therefore $scopeop(E, s)$ in normal form, which completes the induction. The procedure for transforming all operations in normal form consists of repeatedly picking an event that does not contain a broadcast and transforming its operation to normal form by first eliminating conditionals. All occurrences of broadcasts to that event are replaced by its operation. This is repeated as long as events have not been considered.

6. RFID TAG CASE STUDY

In this short case study, we show how the pCharts model of Figure 1 can be used to analyze properties of an RFID tag [19] and to generate code for am embedded system. This model has concurrent states *ElectronicTag* and *Environment*. In *ElectronicTag*, the basic operation of the RFID device is specified. Initially, a tag is in *StandBy* and on *FieldOn* it goes into *Receive*. Local event *FieldOn* is broadcasted by *Environment* on transition from *Off* to *On*. State *Environment* is initially in *Off* and in time between 58s and 62s, goes into *On*. During this transition, the boolean variable *field* is set to *true*, which means that a low frequency (LF) field is present in the environment.

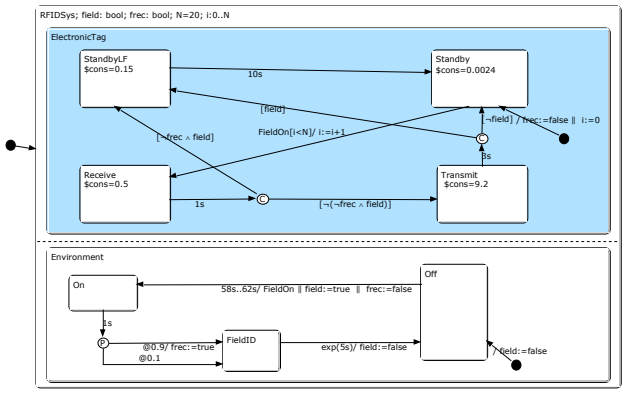


Figure 1: Model of RFID Tag Excitation in pCharts

Whenever the electronic tag is in the range of an LF field, it tries to read the unique field identification number *ID*. This process takes about $1s$, and on average is recognized in 90% of cases. If the field *ID* is recognized, variable *frec* is set to *true*. This is shown by a probabilistic transition from *On* to *FieldID*. We assume that the field will disappear according to an exponential distribution with a scale of 5s, which is specified by transition $exp(5s)$. This means that transition from state *FieldID* to *Off* can take between 1 and 58 *sec*. Theoretically exponential transition will take longer, but we assume that each transition will be taken when the probability of the transition is greater than 99.9999%. We may allow the specification of exponential termination epsilon i.e. $exp(5s, \epsilon)$, which means that if the probability of the transition is greater than $1-\epsilon$, it is consider to be 1. On the transition from *FieldID* to *Off* the variable *field*, which represents presence of the field, is set to *false*.

On the *ElectronicTag* side, in *Receive*, the system field *ID* is read. If *ID* is not recognized ($frec = false$), but a field is still present ($filed = true$), the tag goes into *StandbyLF* or low field standby state, in which it stays the next 10s. This is done to prevent multiple excitation by a pulsating field which cannot be recognized, and to save energy since the consumption in *StandbyLF* is lower than in *Receive*. While in *StandbyLF*, broadcasted event by environment *FieldOn* does not take any effect. If the reading of LF field *ID* is good ($frec = true$), regardless of the status of *field* flag, the tag goes to *Transmit*. After transmission of a preprogrammed number of messages, the tag goes back into the initial state *Standby* if field is not present ($field = false$), or goes to *StandbyLF* if the field is still present ($field = true$). This depends on how fast *Environment* goes from *FieldID* to *Off*, and that is specified by the exponential timed transition $exp(5s)$.

To each state of *ElectronicTag* we assign a *cost* for the power consumption. With our model, we can calculate the average consumption after a number of broadcasts of *FieldOn* event. To count broadcasts we introduce counter variable i and increase it on every transition form *Standby* to *Receive*. In Table 1 we show the minimum and maximum expected costs of a tag cycle (path from initial state *Standby* back to that state). The values are shown for three different probabilities (0.9, 0.8, and 0.1) of a message to be lost, and for an ex-

ponential distribution of the field disappearance of $exp(5s)$. The maximal consumption of one excitation is calculated using formula

$$R\{\text{"}cons\text{"}\}max =? [F\ environment = Off\ \&\ i = 1] \quad (9)$$

which sums the consumption in all *ElectronicTag* states (*Receive, Transmit, Standby, StandbyLF*) when *Environment* reaches *Off* after one tag excitation (process of going form *Standby* to *Receive*).

To validate the model, in addition to counter i which increases every time when event *FieldOn* is generated, we can add temporary *End* states in which both *Environment* and *ElectronicTag* will go at the end of the test. (Without this states PRISM would report a deadlock problem. The reason is the condition on variable i on *FieldOn* event). The verification of formula (9) for run 1 returns 26.7037 and the elapsed time for the model checking process is 135.26 s, on an Intel Core2 Duo CPU 2.00GHz laptop. The minimum expected consumption is 1.2394. The built model has 1136223 states and 1728571 transitions. The generated PRISM PTA code for RFID case study is sown in theextended version of this paper [16]. Based on the calculation of the maximal consumption and information about tag battery capacity, we can calculate the expected tag lifespan, as one of the most important design requirements of active RFID tags. By modifying the consumption in some states, we can verify impact on the overall consumption, which can help with optimizing the product.

Executable code generation. pCharts describe the high level structure and behaviour, rather than all details of the implementation. From the pCharts specification we generate the scheduled timed events and the *software control loop*, while other parts of the code, like oscillator setting, initialization of the registers, evaluation of input, and setting actions on the output are written separately. To handle timed events we use the internal timer that generates interrupt and call a procedure to count time in scheduled timed events. From selected part of a pCharts model we can generate executable code. In our example 1, if we select *ElectronicTag* state (blue shaded state), executable code for embedded system of microcontroler form PIC16Fxx family can be generated. Internal timer is set to generate interrupt every $1ms$, which is used by the scheduler to arrange the next due time procedure. We automatically generate PIC C code framework, which includes all timed events and the software control loop. Part of the code to configure the oscillator, initialize I/O and peripherals (*setupProcessor.c*), and code to define I/O Actions (*actions.h*) are target dependent and are written separately. The file which initialize scheduler data and procedures (*Scheduler.h*) is prewritten and target independent. The generated code is compatible with HI-TECH C Compiler for PIC10/12/16 MCUs. Timed transitions are managed by a scheduler with two procedures, *schedule* and *cancel*. Procedure *schedule(timedproc, tm, prio)* schedules the execution of *timedproc* at time *tm* with priority *prio*. Procedure *cancel(timedproc)* removes *timedproc* from the schedule. Generated executable code and source code of other files an in an extended version of this paper [16]. Our implementation of the code generation to micro-controllers is similar to IAR Visual State for implementing embedded applications based on state machines, but we assume that events are processed fast enough so that we do not need an *event handler*. IAR Visual State can be used for testing and for code generation, but can not perform quantitative verification of the systems.

7. CONCLUSION

We describe timed transition specification in pCharts from which code for the probabilistic timed automata model of the PRISM model checker and executable code for embedded systems can be generated. On the PTA model we can perform formal verification during system specification process, which allows us to detect and isolate possible design faults in earlier phases of software development. On a case study we show how we can specify the impact of the environment, which can be used to optimize device hardware (i.e. power consumption) and software design for a particular environment. The application can be developed in a natural, iterative fashion. The translation of time transitions specified in pChars to code is tested for each transition (*after*, *between*, *exactly*, *stochastic*) separately, as well as on the number of case study examples, but a formal proof of translation correctness remains to done yet.

The experimental pState software development experimental tool enables rapid application development through the use of a *holistic* pCharts design, which includes verification of correctness, quantitative analysis, and code generation.

Currently we use PRISM as the backend model checker, but the pState architecture allows other probabilistic model checkers like Fortuna or MRMC to be added. The general problem of model checkers is sate-space explosion. One of the way to handle this problem is to use approximate or statistical model checkers and estimate the correctness of a design. Some probabilistic statistical model checkers like APMC and Ymer can be also added as backend verification tools.

Acknowledgements. We would like to thank the reviewers for their valuable comments.

8. REFERENCES

[1] L. Baresi, A. Morzenti, A. Motta, and M. Rossi. A logic-based semantics for the verification of multi-diagram UML models. *SIGSOFT Softw. Eng. Notes*, 37(4):1–8, July 2012.

[2] D. Beauquier. On probabilistic timed automata. *Theor. Comput. Sci.*, 292(1):65–84, Jan. 2003.

[3] C. Chen, J. Sun, Y. Liu, J. Dong, and M. Zheng. Formal modeling and validation of stateflow diagrams. *International Journal on Software Tools for Technology Transfer (STTT)*, 14:653–671, 2012.

[4] A. David, M. O. Möller, and W. Yi. Formal verification of UML statecharts with real-time extensions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*, volume 2306 of *LNCS*, pages 218–232. Springer, 2002.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June

Table 1: Min/max Expected Cost vs. Channel Quality

Run	Environment			Electronic Tag				Expected Cost	
	Succ.	Lost.	Exp	Receive	Standby	Transmit	StandbyLF	Min	Max
1	0.9	0.1	5	0.5	0.0024	9.2	0.15	1.2394	26.7037
2	0.8	0.2	5	0.5	0.0024	9.2	0.15	1.2524	23.9642
3	0.1	0.9	5	0.5	0.0024	9.2	0.15	1.3434	4.7871

1987.

[6] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[7] IBM. IBM Rational Rhapsody. `http://www-03.ibm.com/software/products/en/ratirhapfami`, February 2014.

[8] D. N. Jansen. *Extensions of Statecharts with Probability, Time, and Stochastic Timing*. PhD thesis, University of Twente, Enschede, 2003.

[9] D. N. Jansen. More or less true: DCTL for continuous-time MDPs. In *Proceedings of the 11th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'13, pages 137–151, Berlin, Heidelberg, 2013. Springer-Verlag.

[10] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic games for verification of probabilistic timed automata. In J. Ouaknine and F. Vaandrager, editors, *Proc. 7th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'09)*, volume 5813 of *LNCS*, pages 212–227. Springer, 2009.

[11] J. Lilius and I. P. Paltor. Formalising UML state machines for model checking. In *Proceedings of the 2nd international conference on The unified modelling language: beyond the standard*, UML'99, pages 430–444, Berlin, Heidelberg, 1999. Springer-Verlag.

[12] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in PROMELA/SPIN. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, WIFT '98, pages 90–, Washington, DC, USA, 1998. IEEE Computer Society.

[13] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *Association for Computing Machinery Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[14] B. Nokovic. pState Webpage. `pstate.mcmaster.ca`, July 2014.

[15] B. Nokovic and E. Sekerinski. pstate: A probabilistic statecharts translator. In *Embedded Computing (MECO), 2013 2nd Mediterranean Conference on*, pages 29–32, 2013.

[16] B. Nokovic and E. Sekerinski. Verification and Code Generation for Timed Transitions in pCharts - Extended. `http://www.cas.mcmaster.ca/~nokovib/C3S2E2014extended.pdf`, July 2014.

[17] G. Norman, D. Parker, and J. Sproston. Model checking for probabilistic timed automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[18] OMG Unified Modeling LanguageTM (OMG UML), Superstructure, Version 2.2, 2009.

[19] M. Paun. Posttag PT23 technical specification. Technical report, Lyngsoe Systems, 2006. Internal report.

[20] C. Petersohn and L. Urbina. A timed semantics for the statemate implementation of statecharts. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 553–572. Springer Berlin Heidelberg, 1997.

[21] M. Pradella, A. Morzenti, and P. San Pietro. Refining real-time system specifications through bounded model- and satisfiability-checking. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 119–127, 2008.

[22] E. Sekerinski. Graphical design of reactive systems. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, pages 182–197. Springer-Verlag, 1998.

[23] E. Sekerinski. Verifying statecharts with state invariants. In K. Breitman, J. Woodcock, R. Sterritt, and M. Hinchey, editors, *13th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '08, pages 7–14, Belfast, Northern Ireland, March 2008. IEEE Computer Society.

[24] E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language, 4th International Conference*, volume 2185 of *Lecture Notes in Computer Science*, pages 376–390, Toronto, Canada, 2001. Springer-Verlag.

[25] E. Sekerinski and R. Zurob. Translating statecharts to b. In M. Butler, L. Petre, and K. Sere, editors, *Third International Conference on Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 128–144. Springer-Verlag, 2002.

APPENDIX
A. GENERATED PRISM PTA CODE

```
pta

const N=20;
const RFIDSys=0;
const On=0; const FieldID=1; const Off=2;
const EEnd=3; const End=4;
const Receive=0; const Standby=1; const Transmit=2;
const StandbyLF=3;

module rfidformats
    tc1 :[0..58]  init 0;
    root :[0..1]  init RFIDSys;
    environment :[0..3]  init Off;
    environmentclk : clock;
    electronictag :[0..4]  init Standby;
    electronictagclk  : clock;
    i :[0.. N] init 0;
    field :  bool init false;
    frec :  bool init false;

    invariant
        (environment=On=>environmentclk<=1)
        & (environment=FieldID=>environmentclk<=5)
        & (environment=Off=>environmentclk<=58)
        & (electronictag=Receive=>electronictagclk<=1)
        & (electronictag=Transmit=>
          electronictagclk <=3)
        & (electronictag=StandbyLF=>
          electronictagclk <=10)
    endinvariant

    [] (electronictag=Transmit)&(electronictagclk=3)->
       (electronictag'=field?StandbyLF:Standby)&
       (electronictagclk'=0);
    [] (electronictag=StandbyLF)&
       (electronictagclk=10) ->
       (electronictag'=Standby)&(electronictagclk'=0);
    [] (environment=FieldID)&(tc1>=58) ->
       (environment'=Off)&(environmentclk'=0)&
       (field'=false);
    [] (environment=FieldID)&(tc1<58)&
       (environmentclk>=1) ->
       0.81873:(environment'=FieldID)&
       (environmentclk'=0)&(tc1'=(tc1+1)) +
       0.18127:(environment'=Off)&(field'=false);
    [] (environment=Off)&(environmentclk>=58)&
       (environmentclk<=62) &(electronictag=Standby)&
       (i<N) ->
       (i'=(i+1))&(field'=true)&(frec'=false)&
       (environment'=On)&(environmentclk'=0)&
       (electronictag'=Receive)&(electronictagclk'=0);
    [] (environment=Off)&(environmentclk>=58)&
       (environmentclk<=62) &
       (electronictag!=Standby)&(i<N) ->
       (field'=true)&(frec'=false)&
       (environment'=On)&(environmentclk'=0);
    [] (environment=On)&(environmentclk=1) ->
       0.9:(environment'=FieldID)&(frec'=true) +
       0.1:(environment'=FieldID)&(environmentclk'=0);
    [] (electronictag=Receive)&(electronictagclk=1) ->
       (electronictag'=!((!(frec)&field))?
       Transmit:StandbyLF)&(electronictagclk'=0);
    [] (electronictag=Standby)&(i=N) ->
       (electronictag'=End);
    [] (environment=Off)&(i=N) ->
       (environment'=EEnd);
endmodule

rewards "cons"
    (electronictag=Receive): 0.5;
    (electronictag=Standby): 0.0024;
    (electronictag=Transmit): 9.2;
    (electronictag=StandbyLF): 0.15;
endrewards
```

B. GENERATED CODE FOR PIC16XX

```
#include <pic.h>
#include <stdio.h>
#include <stdlib.h>
#include <htc.h>

#include "scheduler.h"
#include "actions.h"

/* Global variable declaration */
enum electronictag_status
        {StandbyLF, Receive, Standby, Transmit} root;

bit frec = 0;
bit field = 0;

void exactly0 (unsigned int);
void exactly1 (unsigned int);
void exactly2 (unsigned int);
void exactly3 (unsigned int);

// Configuration bits:
__CONFIG(WAKECNT & FCMDIS & IESODIS &
        BORDIS & UNPROTECT & MCLREN &
        PWRTEN & WDTDIS & INTIO);

/* Main Program */
int main(void){

    /* Configure the oscillator,
       initialize I/O and Peripherals */
    InitDevice();

    /* initialize data */
    n = 0;
    tr = 0;
    ir = 1;

    root = Standby;
    frec=false;
    timer_running = 1;

    while(1) {
        /* Timed event trigger polling */
        if (run) {
            run = 0;
            timedEvent[event] (tm[event]);
        }

        /* I/O Actions */
    }

}

void FieldOn(unsigned int t){
    if ((root == Standby)) {
        root = Receive;
        schedule(exactly2, 1000, 1);
    }
}

void exactly1(unsigned int t){
    if ((root == StandbyLF)) {
        root = Standby;
        frec=false;
    }
}

void exactly0(unsigned int t){
    if ((root == State)) {
        if(field){
                root=StandbyLF;
                schedule(exactly1, 20000, 1);
        }
        else{
                if (!(field)) {
                        root=Standby;
                }
        }
    }
}

void exactly2(unsigned int t){
    if ((root == Receive)) {
        if(!(frec)){
                root=State;
                schedule(exactly0, 1000, 1);
        }
        else{
                if (frec) {
                        root=Transmit;
                        schedule(exactly3, 4000, 1);
                }
        }
    }
}

void exactly3(unsigned int t){
    if ((root == Transmit)) {
        if(field){
                root=StandbyLF;
                schedule(exactly1, 20000, 1);
        }
        else{
                if (!(field)) {
                        root=Standby;
                }
        }
    }
}
```

C. PROCESSOR SETUP

```
/*
 * File: setupProcessor.c
 *
 * Created by Bojan Nokovic on 13-08-03.
 *
 * Copyright (c) 2013
 *       McMaster University, Hamilton, Canada
 *       All rights reserved.
 *
 */
#include <htc.h>
#include <stdio.h>
#include <stdlib.h>
#include <pic16f636.h>

/* System frequency */
#ifndef _XTAL_FREQ
#define _XTAL_FREQ 1695000
#endif

/* Initialize registers */
void InitDevice(void) {

        //OPTION = 0b10000110; //Enable TMR0 with
            1:128 prescaler
        OPTION = 0b00000110; // Enable internal pull up
                             // Pull-ups on PORTA/
                                PORTB are disabled
                             // Interrupt on falling edge
                                of INT pin
                             // Internal instruction cycle
                                clock (FOSC/4)
                             // Increment on high-to-low
                                transition on T0CKI pin
                             // Prescaler is assigned to
                                the Timer0 module
                             // TMR0 rate = 1:128

        INTCON = 0x00;   // All interrupts disabled.
        IOCA = 0x00;     // Interrupt on change for
            PORTA disabled.

        TRISA = 0b00110101; // PORTA directions: 1=
            input, 0=output
        TRISC = 0x01;    // RC0 - input, all other
            outputs

        PORTC = 0b00000000;
        PORTA = 0x00;

        /* Configure timer 0 */
        T0CS = 0;        // Timer mode for Timer0
        PSA = 0;         // Assign the prescaler to
            Timer0
        TMR0 = 249;   // 256 - TIMER_counts_ms; 1/
            Fosc/4 * TIMER_counts_ms ~ 1ms

        GIE = 1;         // Global interrupt enable
        T0IE = 1;        // Enable Timer0 Overflow
            interrupt
}
```

D. SCHEDULER

```
/*
 *  File: tTimer.h
 *
 *  Created by Bojan Nokovic on 13-08-03.
 *
 *  Copyright (c) 2013
 *       McMaster University, Hamilton, Canada
 *       All rights reserved.
 *
 */
#ifndef SCHEDULER_H
#define SCHEDULER_H

/* For 1 ms delay @ F=1.695MHz  Tcy= 1/(F/4)=2.36us
    */
// const unsigned char TIMER_counts_ms = 141;
// for F=32KHz (0..7);
const unsigned char TIMER_counts_ms = 7;

unsigned int timer_ticks;
char timer_running;

int test = 0;
int  b=0;

void InitDevice(void);
void Tick(void);
void schedule(void (*myTimedEvent)(int), int, int);
void cancel(void (*myTimedEvent)(int));

char timer_running;

void (*timedEvent[MAX_TIMED_EVENTS])(unsigned int
    ); // array of event function pointers
char ir;
char tr;
unsigned int tm[MAX_TIMED_EVENTS]; // time
int pr[MAX_TIMED_EVENTS];     // priority
int n;                        // number of timed
    event to schedule
char run;
int event;

/*
 * Tick is called every ms.
 * It decrease time in scheduled time event, when it
     reach zero,
 * event is enabled.
 */
void Tick(void) {
    int i = 0;
    if (timer_running) {
        for (i = 0; i < n; i++) {
            // search for due event
            if (!--tm[i]) {
                // there is tmed event to be executed
                run = 1;
                event = i;
                // if slef-loop, we have to take care
                    about
                // cumulative drift
            }
        }
    }
}

/*
```

```
 * Timer overflow interrupt
 */
static void interrupt
isr(void)
{
    // Timer interrupt
    if (T0IF) {
        // Increase internal timer tick
        Tick();
        TMR0 = 256 - TIMER_counts_ms; // 1/Fosc/4 *
            TIMER_counts_ms ~ 1ms
        T0IF = 0;
    }
}

/*
 * Add event to data structure
 *  myTimedEvent - function pointer
 *  t - time
 *  p - priotity
 */
void schedule(void (*myTimedEvent)(int), int t, int p) {
    tm[n] = t;
    pr[n] = p;
    timedEvent[n] = myTimedEvent;
    n = n + 1;
}

/*
 * Cancel timed event
 * myTimedEvent - function pointer
 */
void cancel(void (*myTimedEvent)(int)) {
    int i = 0;
    while (timedEvent[i] != myTimedEvent && (i <
        MAX_TIMED_EVENTS)) {
        i++; // search for myTimedEvent
    }
    if (i < n) {
        // Swap with last
        n = n - 1;
        timedEvent[i] = timedEvent[n];
        tm[i] = tm[n];
        pr[i] = pr[n];
    }
}

#endif /* Scheduler.h */
```