Verification and Code Generation for Timed Transitions in pCharts

Bojan Nokovic
McMaster University
Computing and Software Department
1280 Main Street West, Hamilton, Ontario, Canada
nokovicb@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]: State diagrams; 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CSC2E'14, August 03 - 05 2014, Montreal, QC, Canada
Copyright ©2014 ACM 978-1-4503-2712-1/14/08 $15.00.
http://dx.doi.org/10.1145/2641483.2641522.
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 flattening 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 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. Relative 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, JAR Visual Statechart Tools, 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, ev is a list of unique variables, ev is a list of expressions of the same length as ev, 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
- ev := ev the multiple assignment, always enabled
- g → 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 || 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 || R is a generalization of multiple assignment in the sense that (x, y := e, f) = (x := e, 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 interleave. The probabilistic choice is more commonly written as p1 : Q1 \( || \cdots || \) pn : Qn, or, using comprehension notation, \( \{ i \in I : p_i : Q_i \} \). As both nondeterministic choice and independent composition are associative, we write simply \( Q_1 \{ \cdots \} \{ Q_n \} \) and \( Q_1 \{ \cdots \} \{ Q_n \} \) without parenthesis, or, using comprehension notation, \( \{ i \in I : p_i : Q_i \} \) and \( \{ i \in I : p_i : Q_i \} \). The conditional statement is defined in terms of above statements:

- if b then Q \( \equiv (b \rightarrow Q) \{ \neg b \rightarrow \text{skip} \} \)
- if b then Q else R \( \equiv (b \rightarrow Q) \{ \neg b \rightarrow R \} \)

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 \( \Gamma \rightarrow PD\Gamma \), where \( PD = \Gamma \rightarrow [0, 1] \) such that \( \sum d = 1 \) for all \( d \in PD \), distribution d is not 0 for finitely many states of \( \Gamma \), and \( P \) 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 \( Q : \Gamma \rightarrow PD\Gamma \) and \( R : \Delta \rightarrow PD\Delta \) then \( Q \parallel R : (\Gamma \times \Delta) \rightarrow PD(\Gamma \times \Delta) \). The following properties of statements will be used, where b, c are Boolean expressions, \( P, Q, R \) are statements, \( x, y \) are lists of variables, and ev, fv are lists expressions of same
length as $xv, yv$; their proofs are left out for brevity:

$$xv := ev \parallel yv := fv = xv, yv := ev, fv$$

(1)

$$b \rightarrow c \rightarrow Q = (b \land c) \rightarrow Q$$

(2)

$$b \rightarrow (P \parallel Q) = (b \rightarrow P) \parallel (b \rightarrow Q)$$

(3)

$$(P \parallel Q) \parallel R = (P \parallel R) \parallel (Q \parallel R)$$

(4)

$$(b \rightarrow Q) \parallel R = b \rightarrow (Q \parallel R)$$

(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:

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

(6)

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

(7)

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

$$p : (b_1 \rightarrow P_1 \parallel b_2 \rightarrow P_2) \oplus q : Q = b_1 \rightarrow (p : P_1 \oplus q : Q) \parallel b_2 \rightarrow (p : P_2 \oplus q : Q)$$

if $b_1 \lor b_2$

(8)

To see why the condition is necessary, assume that $b_1 = b_2 = \text{false}$: the left side blocks with probability $p$ and chosen $T$ 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 $\text{in}(S)$ tests if the chart is in state $S$ and the statement $\text{goto}(S)$ moves the chart to state $S$:

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 $S'_1, \ldots S'_n$ do not have to be distinct:

$$\begin{align*}
S_i \xrightarrow{\text{in}(S_i) \land q_i \rightarrow goto(S'_i) \parallel a_i \$c_i} \\
\ldots \\
S_n \xrightarrow{\text{in}(S_n) \land q_n \rightarrow goto(S'_n) \parallel a_n \$c_n}
\end{align*}$$

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.

$$\begin{align*}
\sum_{i=1}^{n} p_i \\
\xrightarrow{\text{in}(S_i) \land q \rightarrow goto(S'_i) \parallel a_i \oplus \ldots \oplus a_n \$c}
\end{align*}$$

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

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

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:

For timed events we allow equivalent notations: after($t$) is the same as $\ldots$, 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 $\text{exp}(t)$ the delay is defined by an exponential distribution with an average duration of $t$ time units. The timed transition $\text{unif}(t_1, t_2, d)$ indicates a uniformly distributed delay with the given minimum duration of $t_2$ and maximum duration $t_2$ time units. The uniform distribution takes an optional parameter $d$ for the step, as the distribution is approximated discretely.

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.

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:

\[
\begin{align*}
S_1 & \xrightarrow{l : E \& c}_{S_2} T_1 & \quad T_2 & \quad T_m \\
S_1 & \xrightarrow{r : \{S_1.. S_m\}} S_m & \xrightarrow{s : \{T_1.. T_n\}}
\end{align*}
\]

\[
\begin{align*}
\text{cost}(S_i) &= c_i \\
\text{cost}(T_j) &= d_j \\
goto(S_i) &= r := S_i \\
goto(T_j) &= s := T_j
\end{align*}
\]

This corresponds to setting the variables for all the concurrent states:

\[
\begin{align*}
r : \{S_1, S_2\} & \text{, } q : \{Q_1.. Q_m\} \text{, } t : \{T_1.. T_n\} \\
op(E) &= \text{in}(S_1) \rightarrow \text{goto}(S_2) \parallel \text{goto}(T_1) \text{, } c
\end{align*}
\]

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:

\[
\begin{align*}
\text{cost}(S_i) &= c_i \\
\text{cost}(T_j) &= d_j \\
goto(S_i) &= \text{in}(S_1) \rightarrow \text{goto}(S_2) \parallel \text{goto}(T_1) \text{, } c \\
goto(T_j) &= \text{in}(S_1) \rightarrow \text{goto}(T_2) \text{, } d
\end{align*}
\]

\[
\begin{align*}
\text{cost}(S_i) &= c_i \\
\text{cost}(T_j) &= d_j \\
goto(S_i) &= \text{in}(S_1) \rightarrow \text{goto}(Q_2) \parallel \text{goto}(T_2) \text{, } c + d \\
goto(T_j) &= \text{in}(S_1) \rightarrow \text{goto}(Q_2) \text{, } c \\
\text{cost}(S_i) &= \text{in}(S_1) \rightarrow \text{goto}(T_2) \text{, } d
\end{align*}
\]

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 \( op(\cdot) \). 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 $(Q_2, T_2)$:

$$op(E) = in(S_1) \land in(Q_1) \rightarrow goto(Q_2) \parallel op(F) ~$c
$$
$$op(F) = in(S_1) \land 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
- $parent : State \rightarrow \{ Root \} \rightarrow State$
- $var : State \rightarrow ( Variable \rightarrow Type )$
- $ev : State \rightarrow \mathcal{P} \ Event$
- $inv : State \rightarrow Expr$
- $cost : State \rightarrow Expr$

All states form a tree that is rooted in $Root$, formally $Root \in parent^+(ss)$ 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:

- $source : Transition \rightarrow \mathcal{P} \ State$
- $event : Transition \rightarrow Event \cup Time$
- $guard : Transition \rightarrow Expr$
- $cost : Transition \rightarrow Expr$
- $alt : Transition \rightarrow \mathcal{P} \ Alternative$
- $prob : Alternative \rightarrow Expr$
- $target : Alternative \rightarrow \mathcal{P} \ State$
- $body : Alternative \rightarrow Statement$
- $default : XOR \rightarrow Alternative$

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

$$Time \triangleq \text{between}(\mathbb{R}_+^{\geq 0}, \mathbb{R}_+^\infty) \mid \text{exp}(\mathbb{R}_+^\infty) \mid \text{unif}(\mathbb{R}_+^\infty)$$

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 $x \sim y$ for the pair $(x, y)$ belonging to relation $r$.

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

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.

$$scope(t) \triangleq cca(source(t)) \cup (\bigcup d \in alt(t) . target(d))$$

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 $ss$.

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

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 \( d \).

\[
\begin{align*}
\text{entered}(s, d) & \equiv \text{path}(\text{scope}(t), \text{source}(t)) \\
\text{path}(s, \text{target}(d))
\end{align*}
\]

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.

\[
\text{imp}(ss) \equiv \text{children}[ss \cap \text{AND}] - ss
\]

The completion of an alternative \( d \) starting at state \( s \) and targeting descendents 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.

\[
\text{comp}(s, d) \equiv \{ (s, d) \} \cup (\bigcup r \in (\text{target}(d) \cap \text{XOR}) \cup \text{imp}(\text{entered}(s, d)) \cup \text{comp}(r, \text{default}(r)))
\]

Certain XOR states are required to have a default initial state: a default alternative must be defined for the root state, Root \( \in \text{dom default} \), 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 \( \text{comp}(\text{scope}(t), d) \) is well-defined for all \( t \in \text{Transition} \) and \( d \in \text{alt}(t) \).

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

\[
\text{Expr} ::= \text{Variable} \mid \text{in} \, (	ext{Expr}_1, \ldots, \text{Expr}_m) \mid \text{fn} \, (\text{Expr}_1, \ldots, \text{Expr}_n)
\]

A function without arguments must be an integer, Boolean, or real constant. A function can also be one of the unary operators \( +, -e, +e, e \times f, e \div f, e \mod e, e / f, e \land f, e \lor f, e \Rightarrow f, e \Leftrightarrow f, e = f, e \neq f, e < f, e \leq f, e > f, e \geq f \), or the logarithm, minimum, or maximum function, \( \log(e_1, e_2) \), \( \min(e_1, \ldots, e_m) \), \( \max(e_1, \ldots, e_m) \).

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

\[
\text{Type} ::= \text{integer} \cdot \text{integer} \mid \text{bool} \mid \text{real}
\]

The partial function \( \text{type} : \text{Expr} \rightarrow \text{Type} \) 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 \( \text{decl}(v, S) \) is the closest ancestor, or \( S \) itself, where \( v \) is declared:

\[
\text{decl}(v, S) \equiv \text{if } v \in \text{dom var}(S) \text{ then } S \text{ else } \text{parent}(S)
\]

Thus, if \( v \in \text{Variable} \) occurs in state \( S \), its type is:

\[
\text{type}(v) \equiv \text{var}(\text{decl}(v, S), v)
\]

While expressions can be of any type, variables can only be of subrange or Boolean type. An expression \( e \) is well-typed if \( \text{type}(e) \) 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 \( \text{skip} \), a multiple assignment, a parallel composition, or a conditional. In addition a chart statement can be a \( \text{broadcast} \) of event \( E \in \text{Event} \), simply written as \( E \). All assignment statements have to be \( \text{type-correct} \), 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 \( \text{true} \). If a transition \( /B \) body is missing, it is assumed to be \( \text{skip} \). If there is only one probabilistic alternative, its probability of 1 is left out.

5. TRANSLATION

We present two translations of charts to statements, \( \text{op} \) which generates guarded commands for regular events and \( \text{top} \) 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 \( \text{Root} \), a variable \( \text{lc}(s) \), ranging over \( \text{uc}(c) \) for every child \( c \) of \( s \), is declared. We interpret \( \text{lc}(s) \) and \( \text{uc}(s) \) 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:

\[
\begin{align*}
\text{test}(s) & \equiv \text{lc(parent}(s)) = \text{uc}(s) \\
\text{assign}(s) & \equiv \text{lc(parent}(s)) := \text{uc}(s)
\end{align*}
\]

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

\[
\begin{align*}
\text{in}(ss) & \equiv \forall s \in ss \cap \text{children}[\text{XOR}] . \text{test}(s) \\
\text{goto}(ss) & \equiv \exists s \in ss \cap \text{children}[\text{XOR}] . \text{assign}(s)
\end{align*}
\]

As special cases we have \( \text{in}(\{\}) = \text{true} \) and \( \text{goto}(\{\}) = \text{skip} \). The statement \( \text{goto}(ss) \) 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.

\[
\begin{align*}
\text{trigger}(t) & \equiv \text{in}(\text{exited}(t)) \land \text{guard}(t) \\
\text{effect}(t) & \equiv \oplus \in \text{alt}(t). \text{prob}(c) : \\
& \quad \text{(} (s, d) \in \text{comp}(t, c) \cdot \text{body}(d) \| \\
& \quad \quad \text{goto}([\text{entered}(s, d)]) \| \\
& \quad \quad \text{reset}([\text{entered}(s, d)]) )
\end{align*}
\]

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 \( \text{Trans}(E, s) \) stand for the set of transitions on event \( E \) with scope \( s \):

\[
\text{Trans}(E, s) \equiv \{ t \mid \text{event}(t) = E \land \text{scope}(t) = s \}
\]

The function \( \text{op}(E) \) recursively visits all transitions on \( E \), starting with those on the outermost scope, \( E \). 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 state-charts, the response to an event on which no transition can take place is to do nothing, i.e. \text{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 \text{skip}. Thus the meaning of broadcasting \( F \) is that of \text{op}(F).

We write \( S[F \setminus T] \) for replacing event \( F \) by \( T \) in statement \( S \):

\[
\begin{align*}
\text{op}(E) & \equiv \text{scopeop}(E, \text{Root}) \\
\text{scopeop}(E, s) & \equiv ([t \in \text{Trans}(E, s) \cdot \text{trigger}(t) \rightarrow \\
& \quad \quad \text{effect}(t)[F \setminus \text{op}(F)]) \// \text{childop}(E, s) \\
\text{childop}(E, s) & \equiv \begin{cases} \\
\text{case} \; \text{of} \\
\text{XOR} : \mid c \in \text{children}[\{s\}] - \\
\text{Basic} \cdot \text{test}(c) \rightarrow \text{scopeop}(E, c) \// \text{skip} \\
\text{AND} : \mid c \in \text{children}[\{s\}] - \\
\text{Basic} \cdot \text{scopeop}(E, c) \\
\end{cases}
\end{align*}
\]

The above definition generalizes to the case when more than one event is broadcast. The function \( \text{childop}(E, s) \) is defined only if \( s \) is an \( \text{XOR} \) or \( \text{AND} \) 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 assigns 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:

\[
\text{Trans}(s) \equiv \{ t \mid \text{event}(t) \in \text{Time} \land \text{scope}(t) = s \}
\]

While by \( \text{op}(E) \) we generate the code for the single event \( E \), by \( \text{top}(E) \) 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 \( \text{Root} \) and then recursively descends to transitions with inner scopes, i.e.

\[
\begin{align*}
\text{top}() & \equiv \text{tscopeop}(\text{Root}) \\
\text{tscopeop}(s) & \equiv ([t \in \text{Trans}(s) \cdot \text{trigger}(t) \rightarrow \\
& \quad \quad \text{effect}(t)[F \setminus \text{op}(F)]) \// \text{childop}(s) \\
\text{tchildop}(s) & \equiv \begin{cases} \\
\text{case} \; \text{of} \\
\text{XOR} : \mid c \in \text{children}[\{s\}] - \\
\text{Basic} \cdot \text{test}(c) \rightarrow \text{scopeop}(c) \// \text{skip} \\
\text{AND} : \mid c \in \text{children}[\{s\}] - \\
\text{Basic} \cdot \text{scopeop}(c) \\
\end{cases}
\end{align*}
\]

The scope \( s \) of each transition is an XOR state and with each scope we associate a clock variable, \( \text{clock}(s) \). A timed transition \( t \) with scope \( s \) is scheduled by \( \text{clock}(s) \). If \( \text{event}(t) \) is between \((l, u) \), then \( \text{timeout}(t) \) is \( l \leq \text{clock}(\text{scope}(t)) \land \text{clock}(\text{scope}(t)) \leq u \) becomes a timed invariant of the resulting PTA. We now can define the statement \( \text{reset}(ss) \) for a set \( s \) of states: for each transition \( t \) leaving some \( s \in ss \), the clock of \text{scope}(s) \) is reset, \( \text{clock}([\text{scope}(s)]) := 0 \), 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 \( \{p_1 \oplus \cdots \oplus p_n : A_n \land \text{each } A_i \text{ is a multiple assignment statement} \). We call this the normal form of an operation. For generating a normal form, first \( \text{scopeop}(E, s) \) is equivalently expressed by making the guard explicit instead of writing \”// skip\”. With abbreviations

\[
\begin{align*}
\text{TE} & \equiv [t \in \text{Trans}(E, s) \cdot \text{trigger}(t) \rightarrow \\
& \quad \quad \text{effect}(t)[F \setminus \text{op}(F)] \\
\text{TT} & \equiv \forall t \in \text{Trans}(E, s) \cdot \neg\text{trigger}(t)
\end{align*}
\]

we have:

\[
\begin{align*}
\text{scopeop}(E, s) = \text{TE} \| \text{TT} \rightarrow \text{childop}(E, s)
\end{align*}
\]

From the definitions we observe for \( \text{effect}(t) \) in \( TE \) that \( \text{goto}([\text{entered}(s, d)]) \) is a parallel composition of multiple assignments. If \( \text{body}(d) \) also consists only of multiple assignments (or \text{skip}), then we can use (1) to transform \( \text{effect}(t) \) into a single multiple assignment as needed for the normal form. If \( \text{body}(d) \) 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 \( \text{effect}(t) \) contains a broadcast of an event, say \( F \), then that has to be replaced by \( \text{op}(F) \). We assume that \( \text{op}(F) \) is in normal form and show how to transform \( TE \) to normal form. More specifically, suppose that \( \text{op}(E) \) and \( \text{op}(F) \) are of the form:

\[
\begin{align*}
\text{op}(F) = b_1 \rightarrow P_1 \| b_2 \rightarrow P_2 \\
\text{op}(E) = c_1 \rightarrow (p : \text{op}(F) \| Q) \| c_2 \rightarrow R
\end{align*}
\]

We note that any operation is always enabled as \( P \| Q \) is enabled if either \( P \) or \( Q \) is, so \( P \| \text{skip} \) is always enabled.
As $op(F)$ is always enabled $b_1 \lor b_2$ must hold and we can use (8) to transform $op(E)$ to

$$op(E) = c_1 \rightarrow (b_1 \rightarrow (p : P_1 \oplus q : Q_1) \mid b_2 \rightarrow (p : P_2 \oplus q : Q_2)) \mid c_2 \rightarrow R$$

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 $TT \rightarrow childop(E,s)$ 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 $b_1 \rightarrow P_1 \mid b_2 \rightarrow P_2$, where $P_1, P_2$ are in normal form (and $b_1, b_2$ are state tests) we obtain:

$$TT \rightarrow childop(E,s) = TT \land b_1 \rightarrow P_1 \mid TT \land b_2 \rightarrow P_2$$

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 $(b_1 \rightarrow P_1 \mid b_2 \rightarrow P_2) \mid (c_1 \rightarrow Q_1 \mid c_2 \rightarrow Q_2)$, resulting the normal form returned by $scopeop(E,r)$, we obtain:

$$TT \rightarrow childop(E,s) = TT \land b_1 \land c_1 \rightarrow (P_1 \mid Q_1) \mid TT \land b_1 \land c_2 \rightarrow (P_1 \mid Q_2) \mid TT \land b_2 \land c_1 \rightarrow (P_2 \mid Q_1) \mid TT \land b_2 \land c_2 \rightarrow (P_2 \mid Q_2)$$

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 $P_1 \mid Q_1$ and then use (7) to flatten the nested probabilistic alternatives:

$$P_1 \mid Q_1 = r_1 \times s_1 : (R_1 \mid S_1) \oplus r_2 \times s_2 : (R_2 \mid S_2) \oplus r_1 \times s_1 : (R_1 \mid S_1) \oplus r_2 \times s_2 : (R_2 \mid S_2)$$

Repeating this process brings then $TT \rightarrow childop(E,s)$ 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 conditional statements. 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 an 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.

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 fresh 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 (fresh = false), but a field is still present (field = 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 (fresh = 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

$$R["cons"]_{\max} = 7 \cdot F_{\text{environment}} = \text{Off} \& i = 1$$

(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 state 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 the extended 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 microcontroller 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 PIC16Fxx 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 pCharts 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 MRM to be added. The general problem of model checkers is state-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

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

<table>
<thead>
<tr>
<th>Run</th>
<th>Environment</th>
<th>Electronic Tag</th>
<th>Expected Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Succ.</td>
<td>Lost.</td>
<td>Exp</td>
</tr>
<tr>
<td>1</td>
<td>0.9</td>
<td>0.1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
<td>0.2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>0.9</td>
<td>5</td>
</tr>
</tbody>
</table>

1987.


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;
electronicntag :[0..4] init Standby;
electronicntagclk : 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)
  & (electronicntag=Receive=>electronicntagclk<=1)
  & (electronicntag=Transmit=>
    electronicntagclk <=3)
  & (electronicntag=StandbyLF=>
    electronicntagclk <=10)
  & (environment=FieldID)&(tc1 < 58) +
    0.81873: (environment=FieldID)
  & (environment=FieldID)&(tc1 > 58)
  & (environment=Off)&(field=false)
  & (environment=Off)&(environmentclk=0);

endinvariant

[] ( electronicntag=Transmit)&(i=N) ->
  ( electronicntag=End);

[] (environment=Off)&(i=N) ->
  (environment='EEnd);

endmodule

rewards "cons"
  ( electronicntag =Receive): 0.5;
  ( electronicntag =Standby): 0.0024;
  ( electronicntag =Transmit): 9.2;
  ( electronicntag =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) {
      timedEvent[event] (tm[event]);
    }

    /* I/O Actions */

  }

  void FieldOn(unsigned int t){
    if ((root == Standby)) {
      root = Standby;
    }
  }

  void exactly1(unsigned int t){
    if ((root == Receive)) {
      root = Standby;
    }
  }

  void exactly2(unsigned int t){
    if ((root == Transmit)) {
      if (field) {
        root = Standby;
      }
      else {
        root = Transmit;
      }
    }
  }

  void exactly3(unsigned int t){
    if ((root == StandbyLF)) {
      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>

//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

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
#define SCHEDULER

/* 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, int));
void cancel(void (*myTimedEvent)(int, int, int, int));

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;

/*
 * 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 timed event to be executed
                run = 1;
                event = i;
                // if self−loop, we have to take care
                // about cumulative drift
            }
        }
    }
}

int event;
* Timer overflow interrupt

/∗
static void interrupt
isr (void)
{
    // Timer interrupt
    if (T0IF) {
        // Increase internal timer tick
        Tick();
        T0IF = 0;
        T0IF = 0;
    }
}
/∗

/* Add event to data structure
myTimedEvent − function pointer
* t − time
* p − priority
*/
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++;
    }
    if (i < n) {
        // Swap with last
        n = n - 1;
        timedEvent[i] = timedEvent[n];
        tm[i] = tm[n];
        pr[i] = pr[n];
    }
}
#endif /* Scheduler.h */