From Verification to Implementation: A Model Translation Tool and a Pacemaker Case Study

Miroslav Pajic*, Zhihao Jiang†, Insup Lee†, Oleg Sokolsky† and Rahul Mangharam*†
*Department of Electrical and Systems Engineering
University of Pennsylvania
†Department of Computer and Information Science
University of Pennsylvania
{pajic, zhihaoj, lee, sokolsky, rahulm}@seas.upenn.edu

***Abstract*—Model-Driven Design (MDD) of cyber-physical systems advocates for design procedures that start with formal modeling of the real-time system, followed by the model's verification at an early stage. The verified model must then be translated to a more detailed model for simulation-based testing and finally translated into executable code in a physical implementation. As later stages build on the same core model, it is essential that models used earlier in the pipeline are valid approximations of the more detailed models developed downstream. The focus of this effort is on the design and development of a model translation tool, UPP2SF, and how it integrates system modeling, verification, model-based WCET analysis, simulation, code generation and testing into an MDD-based framework. UPP2SF facilitates automatic conversion of verified timed automata-based models (in UPPAAL) to models that may be simulated and tested (in Simulink/Stateflow). We describe the design rules to ensure the conversion is correct, efficient and applicable to a large class of models. We show how the tool enables MDD of an implantable cardiac pacemaker. We demonstrate that UPP2SF preserves behaviors of the pacemaker model from UPPAAL to Stateflow. The resultant Stateflow chart is automatically converted into C and tested on a hardware platform for a set of requirements.**

I. Introduction

Model-Driven Development (MDD) of real-time and control systems can be conducted in four steps: 1) modeling a plant, 2) design and verification of a controller for the plant, 3) simulating the closed-loop system as a basis for testing, and 4) synthesizing the controller code for platform testing and deployment. These models, when used with model checking and simulation tools, can lead to rapid prototyping, software testing, and verification. In the case of safety-critical systems, the methodology advocates for design procedures that start with formal modeling of the real-time system, followed by the model's verification at an early stage. This allows specification, requirements, and modeling errors with the core model to be found early in the design pipeline. The verified model must then be translated to a more detailed model for simulation-based testing and finally translated into executable code. As later stages of MDD build on the same core model, it is essential that models used earlier in the pipeline are valid approximations of the more detailed models developed downstream.

This research was partially supported by NSF research grants MRI 0923518, CNS 0931239, CNS-1035715 and CNS-0834524.

Timed automata [1] are standard modeling formalism for real-time systems that enables efficient system verification. Due to the limitations of the verification process (e.g., restricted model size), some parts of the models used for verification represent over-approximations of the more 'realistic' models. For example, for verification of Cyber-Physical Systems (CPS) that feature a tight coupling between the real-time controller and (usually) continuous physical environment, it is necessary to model the closed-loop system as a whole. In addition to the plant and controller models, this includes a model of interaction between the controller and the environment. Although in the general case this interaction can be modeled using hybrid systems, the complexity of this approach usually renders it out of reach of current verification tools [2]. Thus, CPS designers often strive to describe the interaction using a set of timing related properties. This results in a combination of model checking of a more abstract timed automata model with a simulation-based analysis of a detailed model with continuous-time dynamics of the environment.

The focus of this effort is on the design and development of a model translation tool, UPP2SF, and how it enables an MDD-based framework. UPP2SF facilitates automatic conversion of verified models (in UPPAAL) to models that may be simulated and tested (in Simulink/Stateflow). This allows for automatic end-to-end model translation across multiple levels of abstraction to generated code. We describe the design rules to ensure the conversion is correct, efficient and applicable to a large class of models. We show how the tool enables MDD of an implantable cardiac pacemaker. The detailed case study highlights the design process (see Fig. 1) from (a) a timed automata model of the pacemaker software, the model verification and model-based worst case execution time estimation; (b) automatic translation of the model to Stateflow using the developed UPP2SF tool; (c) testing of the Stateflow model; and (d) automatic code generation, test generation and testing of timing related errors on the hardware platform. We focus on the pacemaker design as there is a strong need for verification and testing of medical device

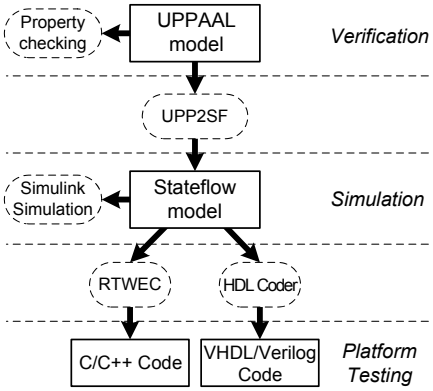


Figure 1. MDD-based framework: From UPPAAL to Stateflow to generated code; This covers model verification, simulation-based testing and platform testing.

software [3]. From 1990-2000, firmware issues resulted in over 200,000 implantable device recalls [4].

UPPAAL [5], [6] is a standard tool for modeling and verification of real-time systems, based on networks of timed automata. As it does not support simulation of continuous-time dynamics, there is a need to translate models verified in UPPAAL into a tool such as Simulink/Stateflow which allows for such simulation. Although there exists some work on code generation from timed-automata models (e.g., [7], [8]), there are only a few tools [8], [9] with limited capabilities in generating C code from UPPAAL models.

Simulink is a commercially available tool that is used for modeling and simulation of CPS, while its toolbox, Stateflow, supports design and simulation of state machines and control logic. Simulink provides full support for C, C++, VHDL and Verilog code generation. However, Simulink has had a very limited success with model verification [10]. Thus, one of our goals is to provide a tool for translation of UPPAAL models to Stateflow. This allows the use of Simulink toolboxes for simulation and code synthesis for controllers verified in UPPAAL, and is a step towards a modular code synthesis. Although there exist different tools for translation of Stateflow models into various frameworks (e.g., [11]), to the best of our knowledge, this is the first tool for translation of UPPAAL models into Stateflow.

The paper is organized as follows: in Sec. II and III we start with an overview of UPPAAL and Stateflow, and show that for a large class of UPPAAL models an execution trace can be obtained by evaluating transitions only at integer time points.[1] In Sec. IV, we present the model translation procedure. Sec. V presents the pacemaker case study, where the pacemaker model was developed and verified in UPPAAL, translated into Stateflow by UPP2SF (Sec. VI), implemented on an RTOS (Sec. VII) and tested (Sec. VIII). We show how the Stateflow chart generated by the UPP2SF simplifies code synthesis and implementation on an RTOS, and testing for specified properties. This allows for worst case execution time (WCET) estimation at the UPPAAL model level, thus, considerably simplifying the design process at an early stage.

[1]Unlike existing work (e.g., [12]), we do not consider the verification of a digitized model, which uses *integral* model of time.

II. System Modeling in UPPAAL

UPPAAL supports networks of timed automata. Each automaton is a state machine, equipped with special real-valued variables called *clocks*. Clocks spontaneously increase their values with time, at the same fixed rate. Locations (i.e., states) in automata have invariants that are predicates over clocks. A location in an automaton can be active as long as its invariant is satisfied. Transitions in automata have *guards* that are predicates over clocks and variables. A transition can be taken only if its guard is true. Because clock values increase, an initially false guard can eventually become true, allowing us to model time-dependent behaviors, such as delays and timeouts. When a transition is taken, an associated *action* is executed, which can update variable values and reset clocks.

Automata in the network execute concurrently. They can communicate via shared variables, as well as via events over synchronous channels. If c is a channel, $c?$ represents receiving an event from c, while $c!$ stands for sending an event on c. In the general case, an edge from location l_1 to location l_2 can be described in a form $l_1 \xrightarrow{g,\tau,r} l_2$, if there is no synchronization over channels (τ denotes an 'empty' action), or $l_1 \xrightarrow{g,c*,r} l_2$. Here, $c*$ denotes a synchronization label over channel c (i.e., $* \in \{!, ?\}$), g represents a guard for the edge and r denotes the reset operations performed when the transition occurs.

A. Semantics of Networks of Timed Automata

To define the semantics of a network of timed-automata we introduce the following terms from [6]. With C we denote the set of all clocks. A clock valuation is a function $u : C \to \mathbb{R}^+$, and we use $\mathbb{R}^C$ to denote the set of all clock valuations. A simple valuation is the function $u_0(x) = 0$, for all $x \in C$. For valuation u, $u+d$ denotes the valuation where $(u + d)(x) = u(x) + d$, for $x \in C$. Furthermore, let $B(C)$ denote the set of conjunctions over simple clock conditions of the form $x \bowtie n$ or $x - y \bowtie n$, where $n \in \mathbb{N}_0$, $x, y \in C$, and $\bowtie \in \{\leq, \geq, =, <, >\}$. Finally, we use K to denote the set of all channels and $A = \{\alpha?|\alpha \in K\} \cup \{\alpha!|\alpha \in K\} \cup \{\tau\}$ the set of all actions.

Definition 1 ([6]). *An automaton $\mathcal{A}$ is a tuple (L, l_0, A, C, E, I), where L denotes the set of locations in the timed automaton, l_0 is the initial location, A is a set of of actions, C a set of clocks, and $E \subseteq L \times A \times B(C) \times 2^C \times L$ denotes the set of edges (between locations, with an action*

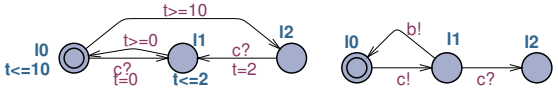


Figure 2. UPPAAL model example; (a) P0 automaton; (b) P1 automaton.

($a \in A$), a guard ($g \in B(C)$) and a subset of clocks to be reset), while $I : L \to B(C)$ assigns invariants to locations.

A network of n timed automata is obtained by composing automata $\mathcal{A}_i = (L_i, l_i^0, A, C, E_i, I_i)$, $i \in \{1, ..., n\}$. In this case, a location vector is defined as $\bar{l} = (l_1, l_2, ..., l_n)$. In addition, an invariant for location vector $\bar{l}$ is defined as $I(\bar{l}) = \wedge_i I_i(l_i)$. To denote the vector where i^{th} element of vector $\bar{l}$ (i.e., l_i) is substituted with l_i' we use notation $\bar{l}[l_i'/l_i]$. If clock valuation u satisfies the invariants at locations from l, we abuse the notation and write $u \in I(l)$. Similarly, if valuation u satisfies condition $g \in B(C)$, we write $u \in g$. Finally, $r(u)$ denotes the clock valuation obtained from u when all clocks from the set $r \subseteq C$ are reset to zero.

Definition 2 ([6]). *Let $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2..., \mathcal{A}_n\}$ be a network of n timed automata, and let $\bar{l}^0 = (l_1^0, l_2^0, ..., l_n^0)$ be the initial location vector. The semantics is defined as a transition system $\langle \mathcal{S}, s_0, \to \rangle$, where $\mathcal{S} = (L_1 \times L_2 \times ... \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\to \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation defined by:*
1) $(\bar{l}, u) \to (\bar{l}, u + d)$ if $\forall d', 0 \le d' \le d \Rightarrow u + d' \in I(\bar{l})$.
2) $(\bar{l}, u) \to (\bar{l}[l_i'/l_i], u')$ if there exists $l_i \xrightarrow{g,\tau,r} l_i'$ s.t. $u \in g$, $u' = r(u)$ and $u' \in I(\bar{l}[l_i'/l_i])$.
3) $(\bar{l}, u) \to (\bar{l}[l_j'/l_j, l_i'/l_i], u')$ if exist $l_i \xrightarrow{g_i,c?,r_i} l_i'$ and $l_j \xrightarrow{g_j,c!,r_j} l_j'$, s.t. $u \in (g_i \wedge g_j)$, $u' = (r_i \cup r_j)(u)$, $u' \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$.

For semantics $\langle \mathcal{S}, s_0, \to \rangle$, a sequence $\mathcal{R}$ defined as $\mathcal{R} := (\bar{l}_0, u_0) \to (\bar{l}_1, u_1) \to ... \to (\bar{l}_i, u_i) \to ...$, is called a *run*, and we use notation $u_k^{\mathcal{R}} := u_k$, $\bar{l}_k^{\mathcal{R}} := \bar{l}_k$, for all $k \ge 0$. For example, a run for the UPPAAL model from Fig. 2 is:

$$\left(\begin{bmatrix} P0.l0 \\ P1.l0 \end{bmatrix}, \begin{bmatrix} t=0 \\ t1=0 \end{bmatrix}\right) \to \left(\begin{bmatrix} P0.l0 \\ P1.l0 \end{bmatrix}, \begin{bmatrix} t=10 \\ t1=10 \end{bmatrix}\right) \to \left(\begin{bmatrix} P0.l2 \\ P1.l0 \end{bmatrix}, \begin{bmatrix} t=10 \\ t1=10 \end{bmatrix}\right) \to \left(\begin{bmatrix} P0.l2 \\ P1.l0 \end{bmatrix}, \begin{bmatrix} t=11.7 \\ t1=11.7 \end{bmatrix}\right) \to \left(\begin{bmatrix} P0.l1 \\ P1.l1 \end{bmatrix}, \begin{bmatrix} t=2 \\ t1=11.7 \end{bmatrix}\right) ...$$

B. UPPAAL Extensions of Timed-Automata

UPPAAL extends the standard framework of timed-automata [1], as it allows use of bounded integer variables, and reset operations that can update variable values and reset clocks to integer values (possibly non-zero). Variables can be used in guard conditions, and guards can be described as conjunction of clock conditions and simple conditions over variables. UPPAAL also extends timed-automata with *committed* and *urgent* locations where time is not allowed to pass (i.e., no delay is allowed) [6]. *Committed locations* are more restrictive and they are usually used to model atomic sequences of actions in UPPAAL. In a network of timed automata, if some automata are in committed locations then only transitions outgoing from the committed locations are allowed. UPPAAL also introduces *broadcast channels*, where one sender can synchronize with multiple receivers.

We focus on a large class of UPPAAL models without clock conditions of the form $x > E$, where x is a clock and E an expression. The restriction, while not limiting in real control system models, guarantees that all invariants and guards are expressed as intersections of left semi-closed intervals. Thus, we refer to this class as *Class LSC*. Unless otherwise specified, in the rest of the paper we consider only this type of models.

Theorem 1. *For each UPPAAL model from the Class LSC, and for every run $\mathcal{R}$ of the model, there exists a run $\mathcal{R}'$ such that:*
1) $\bar{l}_k^{\mathcal{R}} = \bar{l}_k^{\mathcal{R}'}, \forall k \ge 0$ (untimed runs of $\mathcal{R}$ and $\mathcal{R}'$ are equal),
2) for all $k \ge 0$, $0 \le u_k^{\mathcal{R}} - u_k^{\mathcal{R}'} < 1$,
3) for each clock $x \in C$ and for all $k \ge 0$, clock valuation $u_k^{\mathcal{R}'}(x) \in \mathbb{N}_0$ (all events of $\mathcal{R}'$ occur at integer time points).

The theorem allows for the use of discrete-time based tools for simulation of UPPAAL models, as it shows that it is enough to evaluate all guards and invariants at the integer time points in order to find execution traces for the models. The theorem is the basis for the translation procedure used in UPP2SF. Due to space limitations, the proof of the theorem is omitted. It can be obtained from the report [13].

III. Brief Overview of Stateflow

A Stateflow chart employs a concept of finite state machines extended with additional features, including support for different variable types and events that trigger actions in a part or the whole chart. In this section, we describe a small subset of the Stateflow features used in the translation procedure. A detailed description can be found in [14], [11].

A state in a Stateflow chart can be active or inactive, and the activity dynamically changes based on events and conditions. There is a hierarchy between states and a state can contain other states (referred to as substates). A decomposition of a chart (state) determines if its states (substates) are *exclusive* or *parallel* states. Within a chart (or a state), no two exclusive states can be active at the same time, while any number of parallel states can be simultaneously activate.

Unlike in UPPAAL, transitions between states in Stateflow are taken as soon as enabled. They are described as:

$$Event[condition]\{condition\_action\}/\{transition\_action\} \quad (1)$$

where each part of the transition is optional. $Event$ specifies the event that enables the transition, if the $condition$ (if specified) is valid. The $condition$ is described using basic logical operations on simple conditions over design variables and Stateflow operators. Actions in $condition\_action$ and $transition\_action$ include event broadcasting and operations on data variables. There is a subtle difference between these two types of actions. The former are executed if the condition is satisfied, while the outgoing state is still active. The latter actions are executed after the transition occurs, but before the incoming state is activated [14].

A Stateflow chart runs on a single thread and it is executed only when an event occurs. All actions that occur during an execution triggered by an event are **atomic** to that event. After all activities that take place based on the event are finished, the execution returns to its prior activity (i.e., activity before receiving the event). All parallel

states within a chart (or a state) are assigned with a unique execution order. Furthermore, all outgoing transitions from a state have different execution indices. Thus, the execution of a Stateflow chart is fully deterministic – active states are scheduled, and state transitions are evaluated in the execution order (starting from the lowest).

1) Notion of Time in Stateflow Charts: Stateflow temporal logic can be used to control execution of a discrete-time chart in terms of time.[2] It defines time periods using absolute-time operators based on the chart's simulation time, or event-based operators that use the number of event occurrences. Absolute-time logic defines operators $after$, $before$:

$$after(n, \text{sec}) = \begin{cases} 0, \text{ if } t < n \\ 1, \text{ if } t \geq n \end{cases} \quad (2)$$

$$before(n, \text{sec}) = not(after(n, \text{sec})) \quad (3)$$

where t denotes the time that has elapsed since the activation of the associated state (i.e., from the last transition to the state - including self-transitions). The value for time t can be obtained using the operator $temporalCount(\text{sec})$. Finally, for event-based temporal logic, the aforementioned temporal operators are used for event counting. For example, $after(n, clk)$ returns 1 if the event clk has occurred more than $(n-1)$ times after the state has been activated.

IV. Model Translation Procedure

We now describe the translation procedure, along with the features used in the pacemaker model – broadcast channels, committed states and local clocks. We first describe the UPP2SF procedure for translation of UPPAAL edges without synchronization, followed by the procedure used to preserve semantics of broadcast channels. Finally, we show how UPP2SF maps urgent and committed locations. A detailed description of the full translation procedure is provided in [13].

A. Overview of UPP2SF

Consider an UPPAAL model that contains automata $P_0, ..., P_n$. The translation results in a two-level model (see Fig. 3). The top level Stateflow chart corresponds to the UPPAAL network of automata, with the predefined global variables. Since all automata in UPPAAL are simultaneously active, the chart is a collection of parallel states, and for each automaton P_i a parallel state with the same name P_i and a unique execution order[3] is created in the chart. We will refer to the parallel states in the Stateflow chart as the *parent* states. In an UPPAAL automaton, only a single location can be active at a time. Thus, each of the parent states is a collection of exclusive states, extracted from locations in the UPPAAL automaton (e.g., Fig. 7 and Fig. 8 present the pacemaker model in UPPAAL and the extracted Stateflow chart). Furthermore, for each transition between locations in the UPPAAL automaton, a transition between the corresponding Stateflow states is introduced.

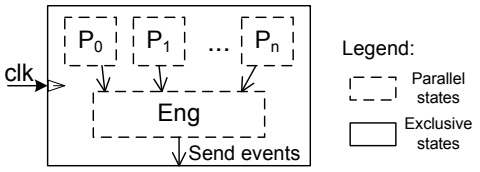


Figure 3. Structure of Stateflow charts obtained using UPP2SF.

Input event clk is added to the chart to guarantee that the extracted chart is executed at each integer time step. Also, a signal generator block is added to the parent Simulink model to create the event at every integer-time point. Hence, to measure time we employ the event-based temporal logic that utilizes the number of clk event appearances, rather than the chart's simulation time (e.g., instead of $after(n, sec)$, we use $after(n, clk)$). The execution of the chart from the moment the chart is triggered by a clk event, until processing of the event has been finished, is referred to as clk *execution*. In the general case, the chart can (re)activate itself (or some of its states) by transmitting local events. Processing of the events triggered during a clk *execution* is considered a part of the clk *execution*. Since processing of events is atomic in Stateflow, no time elapses during a clk *execution*, regardless how many additional event broadcasts have occurred.

Based on Stateflow semantics, at most one transition can occur per parent state upon a chart activation (i.e., within a clk execution), unless the chart is reactivated by sending a local event. On the other hand, there is no upper bound on the number of transitions in an UPPAAL automaton, while time does not pass. In some cases more than one transition has to occur within a single UPPAAL automaton. For example, in the automaton from Fig. 2(a), after transition $P0.l2 \rightarrow P0.l1$ occurs, time t is reset to 2. Due to the invariant at $P0.l1$, no time elapses at the state because the transition $P0.l1 \rightarrow P0.l0$ has to be immediately taken.

To ensure the UPPAAL model semantics are not violated during the translation, we add a parallel state Eng to the chart to serve as the control execution engine. The state reactivates the chart when necessary, and is also used for event broadcasting. With this approach, a single activation of the chart triggers all transitions enabled at that integer time point. This simplifies implementation of the code extracted from the Stateflow chart, because the generated procedure should be invoked only once at every integer time step.

B. Translating UPPAAL Edges Without Synchronization

Consider an UPPAAL edge $l_i \xrightarrow{g,\tau,r} l_j$. Since the guard can be split into a conjunction of data and clock conditions, from the semantics of timed-automata (Def. 2), an equivalent Stateflow transition between the states l_i and l_j has the form:

$$[G_C(I(l_i) \wedge g) \wedge G_V(g) \wedge G_C(r, I(l_j))]/\{R_V(r); R_C(r); R_S(r);\} \quad (4)$$

[2]Although 'temporal logic' has a different meaning in the real-time literature, we use the term in this context, as it is part of Stateflow terminology.

[3]Different assignments of the execution orders (for the parallel states) could result in different runs. Since each of these runs is a run of the initial UPPAAL model, current version of the UPP2SF assigns the execution orders in the order in which UPPAAL automata are parsed.

where:
1) $G_C(h)$ ($G_V(h)$) maps the clock (data) conditions from an UPPAAL condition h into an equivalent Stateflow condition,
2) $R_C(r)$ ($R_V(r)$) translates the clock (data) resets from r into an equivalent Stateflow assignment (see Section IV-B1),
3) $G_C(r, I(l_j))$ maps the condition that the clock valuation after the reset satisfies the invariant at the 'new' location l_j,
4) $R_S(r)$ controls execution of the chart (Section IV-B2).

Data reset operations, $R_V(r)$, and data guard conditions, $G_V(g)$, are directly mapped using the same Stateflow operations. Mapping of resets and guards that use only local clocks is described below (for full procedure see [13]).

1) Mapping Clock Conditions and Resets: Stateflow temporal logic always resets time when a transition occurs. This forces us to explicitly model each local clock x by introducing the accounting variable n_x. To enable the correct expression of clock constraints used in the UPPAAL model, as part of each transition the variable is updated to maintain the clock value from the moment of the state's activation. This is done using $R_C(r)$ defined for each local clock x as:

$$[R_C(r)](x) := \begin{cases} n_x = n_x + temporalCount(clk), & \text{if } x \notin r \\ n_x = r(x), & \text{if } x \in r \end{cases} \tag{5}$$

Here, if clock x is reset to n as a part of the UPPAAL transition (even if n denotes an expression and not only a constant), variable n_x is set to n on the Stateflow transition (i.e., $n_x = n$ is added to data reset). However, if the clock is not reset in the transition ($x \notin r$), then the accounting variable has to be increased by the value $temporalCount(clk)$.

As described in Section II, each clock condition h is specified as $h = h_1 \wedge ... \wedge h_M$, where h_i is a basic clock condition. Therefore, an equivalent clock condition in Stateflow, $G_C(h)$, can be expressed as $G_C(h) = G_C(h_1) \wedge ... \wedge G_C(h_M)$. The transformation of the basic clock conditions presented in Fig. 4 employs event-based temporal logic operators, while taking into account the values of the accounting variables. Note that since we consider UPPAAL automata where clocks progress at the same rate, for all clocks x, y the difference between the clock values while a state is active is equal to the clock differential upon entering the state. Thus,
$G_C(x - y \bowtie n) := (n_x - n_y \bowtie n)$.

Finally, the requirement that the new clock valuation satisfies the invariant at the (new) location l_j (i.e., $I(l_j)$) is equivalent to the condition that both the reset and non-reset clock values satisfy $I(l_j)$. Therefore, for $I(l_j) = h_1 \wedge ... \wedge h_k$ it follows that $G_C(r, I(l_j)) = G_C(r, h_1) \wedge ... \wedge G_C(r, h_k)$ where:

$$G_C(r, x \bowtie n) := \begin{cases} r(x) \bowtie n, & x \in r \\ (n_x + temporalCount(clk)) \bowtie n, & x \notin r \end{cases} \tag{6}$$

$$G_C(r,\ x - y \bowtie n) := \tag{7}$$
$$\begin{cases} r(x) - r(y) \bowtie n, & \text{if } x, y \in r \\ r(x) - (n_y + temporalCount(clk)) \bowtie n, & \text{if } x \in r, y \notin r \\ (n_x + temporalCount(clk)) - r(y) \bowtie n, & \text{if } x \notin r, y \in r \\ n_x - n_y \bowtie n & \text{if } x, y \notin r \end{cases}$$

UPPAAL condition ($x, y \in C, n \in \mathbf{N}_0$)	Stateflow condition (n_x, n_y clock accounting variables)
$x \leq n$	$before(n - n_x, clk) \|\| (temporalCount(clk) == n - n_x)$
$x < n$	$before(n - n_x, clk)$
$x = n$	$(temporalCount(clk) == n - n_x)$
$x \geq n$	$after(n - n_x, clk)$
$x - y \bowtie n$	$n_x - n_y \bowtie n$

Figure 4. Mapping UPPAAL conditions over clocks into Stateflow – only clock conditions that specify left-closed intervals are considered.

The expression for $G_C(r, I(l_j))$ can be significantly simplified. If $r(x)$ resets the clock x to a constant (which is the prevailing case in UPPAAL), conditions from Eq. (6), (7) can be evaluated during the translation and can be replaced with fixed terms 0 or 1 (i.e., $false$ or $true$).

2) Control of the Chart's Execution: To allow that more than a single transition per parent state can occur during a clk execution, in the obtained Stateflow chart UPP2SF introduces the state Eng, which is executed last among the parent states. Eng is designed to reactivate the chart when transitions occur (Fig. 3). To achieve this additional event tt and flag act are defined in the chart, and as a part of each transition act is set to 1. This is done within $R_S(r)$ from Eq. (4), defined as $R_S(r) := (act = 1;)$. In addition, Eng contains a single state and it broadcasts event tt to the chart if act has been set to 1, using a self-transition (see Fig. 8) of the form:

$$[act == 1]/\{act = 0; send(tt);\} \tag{8}$$

C. Translating Broadcast Channels

Events in Stateflow are a good semantic match for broadcast channels in UPPAAL. Therefore, for each broadcast channel c, UPP2SF defines a Stateflow event c assigned with a unique positive integer $ID(c)$. Consider an UPPAAL edge of the form $l_i \xrightarrow{g,c!,r} l_j$ from automaton P, where c is a broadcast channel. To preserve UPPAAL semantics (Def. 2), the translation of the edge into Stateflow, along with all edges with receiving over channels c, has to satisfy the following requirements:
R1) After the transition occurs, no other 'non-receiving' transition should be taken until all of the transitions conditioned with the event's receiving are taken.
R2) After the event is broadcast, the incoming state l_j has to be activated, so that enabled transitions from l_j can occur at the same (simulation) time. For example, consider automaton $P1$ from Fig. 2(b). After $l1$ is reached, if without any delay $b!$ occurs, the transition between $l1$ and $l0$ has to be taken.
R3) After the event is broadcast, no transition should occur in the parent state P while receiving the event. This ensures that a parent state does not synchronize with itself.

In report [13] we show that, due to Stateflow semantics, events can not be broadcast from the transitions within the parent states derived from UPPAAL automata. Broadcasting events as part of condition actions might result in infinite cycle behavior. Similarly, broadcasting events as a part of transition actions would result in the chart activation where

UPPAAL edge	Stateflow transition
$l_i \xrightarrow{g,\tau,r} l_j$	$[(sent == 0) \wedge G_{C,V}(l_i, g, r, l_j)]/\{R_{C,V}(r); R_S(r);\}$
$l_i \xrightarrow{g_j,c!,r_j} l_j$	$[(sent == 0) \wedge G_{C,V}(l_i, g, r, l_j)]/\{R_{C,V}(r); sent = ID(c);\}$
$l_i \xrightarrow{g_i,c?,r_i} l_j$	$c[(sent \sim= -ExO(P)) \wedge G_{C,V}(l_i, g, r, l_j)]/\{R_{C,V}(r);\}$

Figure 5. Mapping UPPAAL edges from automaton P into Stateflow transitions; $G_{C,V}(l_i, g, r, l_j)$ and $R_{C,V}(r)$ denote the terms $G_C(I(l_i) \wedge g) \wedge G_V(g) \wedge G_C(r, I(l_j))$ and $R_V(r); R_C(r);$ from Eq. (4).

the incoming state (i.e., l_j) is not active, which violates **R2**. Thus, we use a centralized approach where the Eng state sends events and controls execution of the chart. To achieve this we employ additional variable $sent$ that can have the following values:

$$sent = \begin{cases} ID(c), & \text{event } c \text{ is scheduled for broadcast} \\ 0, & \text{no event scheduled for broadcast} \\ -ExO(P), & \text{an event is broadcast and only} \\ & \text{receiving edges should be activated} \end{cases} \quad (9)$$

where $ExO(P) > 0$ is a constant that denotes the execution order of the Stateflow parent state P (note that $ID(c) > 0$).[4]

Fig. 5 shows translation of UPPAAL edges into Stateflow. Action $c!$ is mapped into $sent = ID(C)$ assignment, which disables all 'non-receiving' transitions due to their condition $(sent == 0)$. Similarly, condition $(sent \sim= -ExO(P))$ disables all 'receiving' transitions in the parent state P (satisfying **R3**). For example, consider the model from Fig. 2(b). Since events are broadcast outside of parent states (i.e., from Eng), after the state $P1.l1$ is reached, sending event c from outside of the parent state $P1$ would enable transition $P1.l1 \rightarrow P1.l2$ (if the condition is not used), which violates the requirement **R3**.

Finally, the Eng state is used to broadcast events, by adding for each event c the following self-transition in the state:

$$[(sent == ID(c))]\{sent = -ExO(P); send(c);\}^5 \quad (10)$$

In addition, to reset $sent$, and to ensure that all previously disabled transitions are reevaluated by reactivating the chart after the event is processed (i.e., after all parent states are re-executed), UPP2SF adds the following self-transition in the state Eng:

$$[(sent < 0)]\{sent = 0; send(tt);\} \quad (11)$$

For the Eng state to operate correctly, the transitions (10), (11) have precedence (i.e., lower execution order) over the transition from Eq. (8). An example of Eng state is shown in Fig. 8. In the general case, UPPAAL allows more than one automaton to transmit over a broadcast channel. This requires a simple extension of the procedure and is described in [13].

[4]Stateflow requires that each parent state has a different execution order, i.e., $ExO(P_i) \neq ExO(P_j)$ if $P_i \neq P_j$.

[5]Although the events are sent as a part of condition actions, infinite cycle does not occur, since $sent$ (used in the transition guard) changes its value before broadcasts. This disables the transition in the next activation.

D. Translating Urgent and Committed States

UPP2SF also preserves semantics of urgent and committed states. Adding $act = 1$ to transitions to an urgent state, triggers broadcast of the event tt to reactivate the chart. This enables evaluation of outgoing transitions from the urgent state. However, if these transitions are disabled (e.g., due to event receiving), broadcasting tt before the end of clk executions ensures that all states will be activated at least one more time. Therefore, in the derived Stateflow chart no time passes in the states extracted from urgent states in the UPPAAL model.

If some automata in UPPAAL are in committed locations, then only transitions outgoing from one of the committed locations are allowed. Thus, to deal with committed locations we introduce a new 'control' variable $comm$ that always contains the number of active committed states. For all transitions incoming to a committed state expression $comm = comm+1; act = 1;$ is added to the reset operations (i.e., $R_S(r)$ from Eq. (4)). Similarly, for all outgoing transitions from a committed state $comm = comm - 1; act = 1;$ is added to the reset. To disable transitions from non-committed states when there exists an active committed state, guard condition $(comm == 0)$ is added to all 'non-receiving' transitions outgoing from a committed state. Note that setting act to 1 reactivates the chart to ensure that all transitions are reevaluated, including transitions that have been disabled due to $(comm == 0)$ condition.

E. Optimization

Stateflow chart obtained using the aforementioned rules can be significantly simplified. For example, clock guards and invariants are conjunctions of the basic clock conditions, and thus specify fixed left-closed intervals if only conditions from Fig. 4 are used where n is a constant. These intervals can be expressed in Stateflow with maximum two terms from Fig. 4 (e.g., invariant $t \leq n$ and guard $t \geq n$ can be combined into a single Stateflow condition $temporalCount(clk) == n - n_t$). In addition, it is possible to remove clock resets defined in Eq. (5) from transitions incoming to a state, if on all paths from the state there exists a clock reset (i.e., reset of the corresponding accounting variable of the form $n_x = r(x)$) before the clock is used in a transition guard.

Similarly, due to Eq. (11) there is no need to reactivate the chart with the $act = 1$ reset on transitions to a committed/urgent state that are conditioned with event receiving. The same holds if outgoing transitions from a new state are disabled (which is a common case) at the time of activation. For example, in the buffer from Fig. 7(d), for $dL\_a > 0$, after l_1 is entered the clock guard disables the outgoing transition. Thus, the assignment $act = 1$ does not have to be added to the Stateflow transitions from $l1$ to $l0$. The assignment can be removed from all edges incoming to a state if: 1) the state is not urgent or committed; 2) all clocks used in the state's invariant and outgoing guards from the

state are reset on the incoming transition; 3) all outgoing guards and the state's invariant are clock conditions (without data conditions), specifying a fixed time interval that is not a point.

Finally, transitions between two states with same conditions and transition actions can be combined into a single transition.

V. Pacemaker Case Study

In the remaining of the paper, on a pacemaker case study we describe how the UPP2SF enables an MDD-based framework. We start with an UPPAAL description of a pacemaker before we present how the translation tool allows for automatic code generation and implementation of the implantable pacemaker.

The primary function of an implantable pacemaker is to maintain an adequate heart rate and ensure safe and efficient cardiac output. The pacemaker paces the heart when the intrinsic heart rate is below a lower rate limit, it does not pace the heart above an upper rate limit and maintains synchrony between the atrial and ventricular activity. To interact with the patient, the leads of a pacemaker that can both sense and pace are implanted inside the patient's heart. In this work we describe the most commonly used mode of pacemaker, the DDD mode that paces both the atrium and ventricle, senses both, and sensing either activates or inhibits further pacing [15]. We developed a basic DDD dual chamber model according to [16]. In [17] we present a thorough pacemaker modeling and verification in UPPAAL.

A. Pacemaker UPPAAL Design

The five basic timing cycles of the DDD mode pacemaker are shown in Fig. 6, where AP and AS (VP and VS) denote atrial (ventricular) pacing and sensing events. In our pacemaker model we have designed the following software components, shown in Fig. 7, where each automaton only uses its own local clock.

1. Lowest Rate Interval (LRI) automaton (Fig. 7(b)) models the LRI requirement which is the basic timing cycle. This component keeps the heart rate above a minimum value by delivering atrial pace events (AP). The LRI timer is reset after a ventricular event (VP or VS). For DDD mode, if no atrial event has been sensed (AS) before the timer runs out, the pacing event will be delivered from the atrial lead (AP).

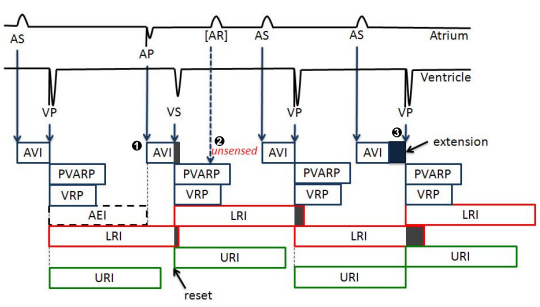


Figure 6. Pacemaker timing cycles; Notation: VP - ventricular pace, VS - ventricular sense, AP - atrial pace, AS - atrial sense [18].

2. Atrio-Ventricular Interval (AVI) automaton from Fig. 7(c) models the AVI requirement. This component mimics the intrinsic AV delay to synchronize the atrial and ventricular events. The timer is started by a sensed or paced atrial event (AP or AS) and can be terminated by a sensed ventricular event (VS). If no ventricular event is sensed before the timer times out, the pacemaker generates a ventricular pace (VP) if the Upper Rate Limit is not violated. Guarantees for this are provided by the URI component.

3. Upper Rate Interval (URI) automaton in Fig. 7(e) limits the ventricular pacing rate by enforcing a lower bound on the times between consecutive ventricle events.

4. PVARP and VRP automata are used to filter noise and early events which could otherwise cause undesired pacemaker behavior. The Post Ventricular Atrial Refractory Period (PVARP) and the Ventricular Refractory Period (VRP) automata generate atrial (AS) and ventricular (VS) sensing events from the buffered atrial and ventricular inputs (AinB and VinB, respectively). The PVARP automaton (Fig. 7(f)) models the blocking interval after each ventricular event (VP or VS) where the atrial sensing (AS) cannot occur. The VRP automaton (Fig. 7(g)) models the blocking interval for ventricular events. The intervals follow ventricular events and no ventricular sensing should occur during the interval.

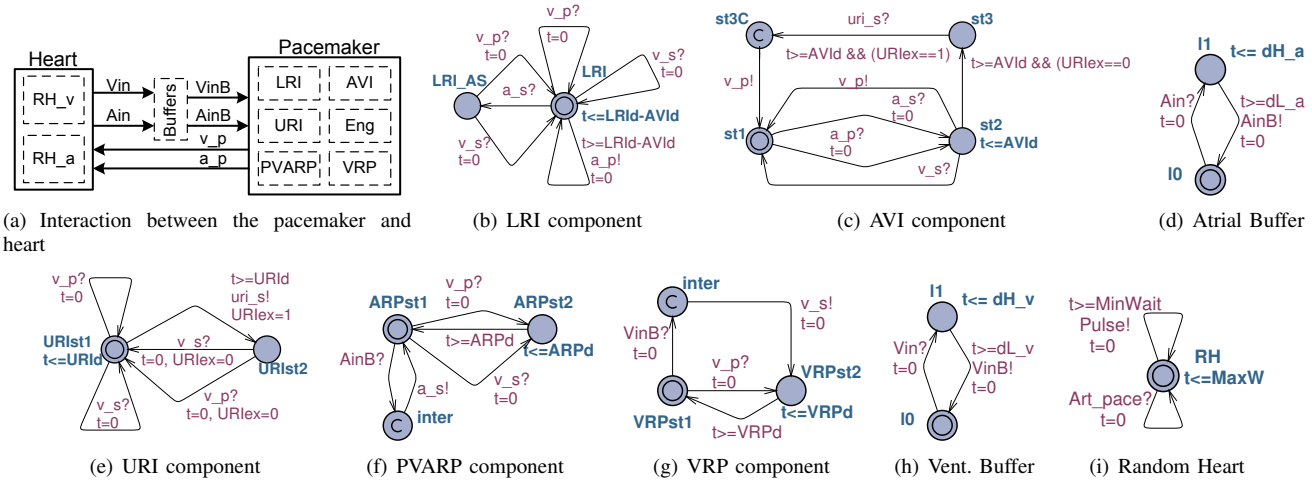
5. Inputs buffers (Fig. 6(d),(h)) are used to model delays imposed by processing inputs Ain and Vin from the heart. For example, these delays can be introduced by the design of the analog interface between the heart and device.

B. Pacemaker Stateflow Design

From the model shown in Fig. 7, using the UPP2SF tool we obtained the pacemaker Stateflow chart presented in Fig. 8. The chart contains 9 parallel states, one for each automaton from the UPPAAL model, and the Eng state for control of the chart's execution. For closed-loop verification in UPPAAL we modeled both the heart and pacemaker, and thus the obtained chart contains both models of the controller (i.e., pacemaker) and the environment (i.e., the heart). Therefore, it is necessary to decouple the pacemaker from the heart model.

Fig. 7(a) presents the interaction between the pacemaker and the heart. Since the interaction corresponds to synchronization over broadcast channels, the pacemaker model can be easily extracted from the chart shown in Fig. 8. This is done by removing the parent states that model the heart and buffers (RH_a, RH_b, ASbuf, VSbuf), and by defining $AinB$ and $VinB$ as input events. Also, the Eng state has to be modified to remove the transitions used to broadcast these input events. In our case we removed the transitions that broadcast $AinB$ or $BinB$ (highlighted in red in Fig. 8).

Stateflow does not allow utilization of output events to condition internal transitions. Hence, it is necessary to define additional output events from the chart, and in our case for



(a) Interaction between the pacemaker and heart (b) LRI component (c) AVI component (d) Atrial Buffer

(e) URI component (f) PVARP component (g) VRP component (h) Vent. Buffer (i) Random Heart

Figure 7. UPPAAL model of a DDD pacemaker - each automaton uses its own local clock; Two Random Heart templates were instantiated for Atrium (Pulse:=Ain, Art_pace:=a_p) and Ventricle (Pulse:=Vin, Art_pace:=v_p).

local events a_p and v_p two output events (AP and VP) were defined. These events are broadcast as a part of the same transitions used to broadcast a_p and v_p, respectively. In addition, to deal with some implementation issues (details are provided in Section VII), for each output `Event` an empty C function `sendHW_Event` is added using Simulink features for integrating custom C code. The function does not affect simulations of the chart in Simulink, but allows for the correct output generation from the synthesized code. For example, the *Eng* transition highlighted with dotted green rectangle was modified to

$$[(sent == 3)]\{sent = -1; send(VP); sendHW\_VP(); send(v\_p);\}$$

VI. Stateflow Chart Validation

We validated the Simulink chart by extending the procedure for testing real-time constraints described in [19]. Therefore, we first describe the set of timing requirements used to extract the test vectors, followed by the simulation results. In [17] we presented the UPPAAL model verification of these properties.

A. Testing Requirements

We classify two types of real-time constraints for pacemaker: *behavioral constraints* that describe time intervals that end when the required input is applied, and *performance constraints* describing intervals that end when the required output is produced. For example, a behavioral constraint is that a certain input E_1 must occur within time interval $[t_1, t_2)$ and as a result it should produce output E_2. Similarly, a performance constraint is a requirement that output has to occur within time interval $[t_1, t_2)$.

As shown in Fig. 6, pacemakers exert a highly repetitive behavior. Thus, we focus on a set of requirements that need to be satisfied in each time interval between consecutive ventricular and/or atrial events. To specify constraints for DDD pacemaker it is necessary to consider two time axes, t_v and t_a that measure the time since the last ventricular (VP or VS) and the last atrial event (AP or AS), respectively. Using the pacemaker specification [16], [15] we defined a set of real-time pacemaker requirements (performance constraints are denoted with **P** and behavioral with **B**):

1. *Pacing in the atrium*:
P1.1. AP cannot occur during the interval $t_v \in [0, LRI_d - AVI_d)$;
B1.1. If AS does not occur within interval $t_v \in [0, LRI_d - AVI_d)$, an AP should occur at $t_v = LRI_d - AVI_d$;
B1.2. If AS occurs at $t_v \in [0, LRI_d - AVI_d)$, AP should not be applied in the atrium within the interval $t_v \in [0, LRI_d - AVI_d)$.

2. *Pacing in Ventricle*:
P2.1. VP cannot occur during the interval $t_a \in (0, AVI_d)$;
P2.2. VP cannot be generated within $t_v \in (0, URI_{def})$;[6]
B2.1. If VS does not occur in intervals $t_a \in (0, AVI_d)$ and $t_v \geq URI_d$, VP should occur at $t_a = AVI_d$;
B2.2. If VS occurs at $t_a \in (0, AVI_d)$, no VP should be generated within the interval $t_a \in (0, AVI_d)$.

3. *Atrial Sensing* (requirements for ARP):
P3.1. AS cannot occur within the interval $t_v \in (0, ARP_d]$;
B3.1. If atrial input (Ain) occurs within interval $t_v \in (0, ARP_d)$, it should be disregarded (no AS is generated within $t_v \in (0, ARP_d)$);
B3.2 If Ain occurs at $t_v \geq ARP_d$, AS is to be created at t_v.

4. *Ventricular Sensing* (requirements for VRP):
P4.1. A ventricular sense (VS) cannot be generated within interval $t_v \in (0, VRP_d)$;
B4.1. If a ventricular input (Vin) occurs at time $t_v \in (0, VRP_d)$ it should be ignored (no VS is generated within $t_v \in (0, VRP_d)$);
B4.2 If Vin occurs at $t_v \geq VRP_d$, VS is to be created at t_v.

Using the above requirements, the testing procedure

[6]The requirement specifies a lower bound on intervals between consecutive events in ventricle – the requirement for URI component.

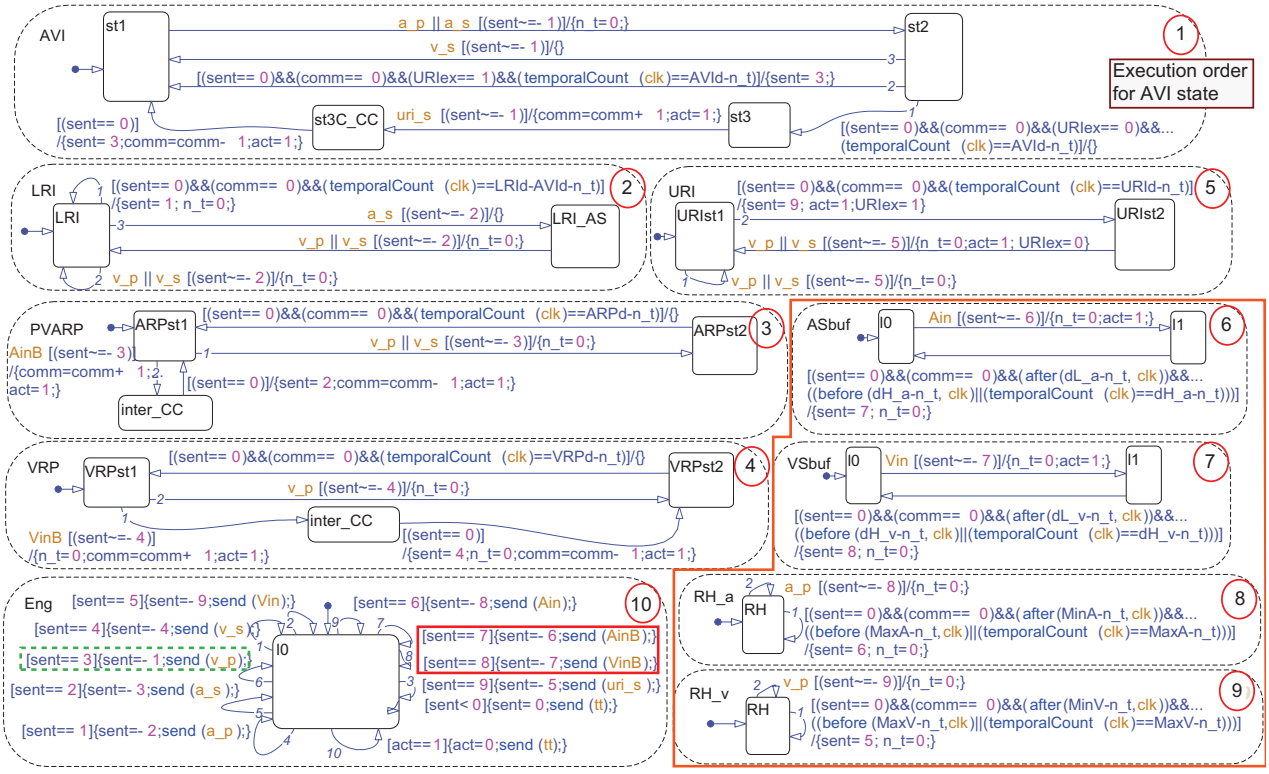


Figure 8. Pacemaker Stateflow chart extracted using UPP2SF from the UPPAAL model in Fig. 7; The heart and buffer models are highlighted.

from [19] was employed to specify a set of tests used for validation. For each performance constraint we create a test to check if the appropriate output has been generated within the required interval. Testing behavioral constraints is more complex. For each interval boundary we generate two tests. For closed boundaries we apply inputs that are exactly at the boundary point and tests points that are outside the interval, at the distance ϵ from the boundary point (Fig. 9). For open boundaries we generate inputs that are inside and outside the interval, at the distance ϵ from the boundary point.

The aforementioned set of requirements is referred to as the *ideal* pacemaker requirements. As specified in [16], each of the intervals is assigned with a certain level of tolerance. Thus, we define 'realistic' requirements, where each of the real-time constraints is modified to incorporate tolerance. For example, we modify constraints **P1.1** and **B1.1** to:
P1.1: AP cannot occur during $t_v \in [0, LRI_d - AVI_d - \Delta_{ap})$;
B1.1: If AS does not occur within interval $t_v \in [0, LRI_d - AVI_d - \Delta_{ap})$, AP should occur at t_v, $t_v \in [LRI_d - AVI_d, LRI_d - AVI_d + \Delta_{ap}]$;

In the above formulations Δ_{ap} defines the tolerance for the atrial pacing requirements. Similarly, we (re)define all of the remaining requirements using the tolerances: Δ_{vp} - tolerance for ventricular pacing; Δ_{as} - tolerance for atrial sensing; Δ_{vs} - tolerance for ventricular sensing.

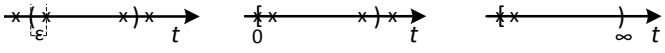


Figure 9. Test points for behavioral real-time constraints.

A set of parameters values, along with their tolerances, is specified in [16] for each of the aforementioned intervals.

B. Testing in Simulink

To perform evaluation of the Stateflow chart we used the nominal values in clinical settings [17]: $LRI_d = 1000ms, AVI_d = 150ms, URI_d = 400ms, VRP_d = 150ms, ARP_d = 200ms$. For testing in Simulink we considered only tests for the *ideal* system specifications. Since the chart was activated every 1ms, and transitions in Stateflow are instantaneous (all transition actions are atomic) we used $\epsilon = 0.5ms$ for simulations. All 13 'ideal' real-time constraints (and thus, the constraints with tolerances) were satisfied in Simulink. This was expected since all actions within a clk execution are atomic to the event and no simulation time elapses during them. In addition, the chart exhibited the same behaviors as the initial UPPAAL model. For example, for the aforementioned model parameters, when no inputs were applied the chart generated AP and VP pulses at the same time points as the UPPAAL model (i.e., AP were generated at $t_i^{ap} = (850 + 1000(i-1))ms,\ i = 1, 2, ...$, and VP at $t_i^{vp} = (1000i)ms,\ i = 1, 2, ...$). Similarly, no time was spent in committed states $st3C\_CC$ in AVI, and $inter\_CC$ states in PVARP and VRP parallel states. To illustrate a more complex behavior: as in UPPAAL, when AVI component was in $st3$ state when the transition $URIst1 \rightarrow URIst2$ occurred (causing broadcast of uri_s event), no time elapsed in the $URIst2$, before the transition $URIst2 \rightarrow URIst1$ conditioned with v_p took place.

VII. Pacemaker Implementation

We generated C code from the pacemaker Stateflow chart using the Simulink Real-Time Workshop Embedded Coder (RTWEC). The code was generated for the general embedded real-time target and as a result we obtained the main procedure, `rt_OneStep`, which processes the three input events, $VinB$, $AinB$ and clk. To ensure that the model semantics is preserved (modulo the execution time), clk events should be created every 1ms, followed by the procedure's activation. This makes it suitable for implementation on top of a real-time operating system (RTOS).

1) Code Structure: The structure of the code is straightforward. The current state of the procedure and all variables defined in the chart are maintained in the structure `rtDWork`. The structure also maintains counter values for all temporal logic operators. As time in temporal logic is measured by the number of events since the state's activation, for each of the parallel states an additional counter is defined. Finally, `rtDWork` contains a structure (List. 1, Fig. 10(a)) that for each parent state specifies if it is active, along with which of its substates is active. For example, `is_active_AVI` is 1 if AVI state is active, while `is_AVI` specifies which of its exclusive substates is active.

The structure of `rt_OneStep` is shown in List. 2, Fig. 10(a). After detecting the active input events, for each active input event, starting from events will lower indices, an execution of the chart procedure `c1_ChartName` is invoked. The variable $\_sfEvent\_$ is used to denote the event that is processed during the chart execution. As in Stateflow, starting from input events with lower indices, the events are processed in a prespecified order using `c1_ChartName` function. After all events are processed the procedure updates the outputs and event states in the prespecified order. This means that although we broadcast output events and the local events corresponding to them (e.g., VP and v_p) as a part of same transitions, the outputs will be actually updated at the end of `rtOneStep`. This can cause a couple of problems. First, ordering of the generated output events can be changed from the order of the corresponding local events. Note that this does not affect simulations in Simulink, since all actions within a clk *execution* are atomic from perspective of the rest of the Simulink model. The second problem is that with this approach, for each output event only a single output trigger can be generated at the end of a clk *execution*. Thus, if an output event is broadcast more than once within a single clk *execution*, the corresponding output events will be actually generated one by one, at the end of the consecutive clk executions (i.e., separated by the duration of clk period).

These issues are resolved using the aforementioned `SendHW_EventName` functions.[7] Using Simulink features for integrating custom C code with Stateflow charts in Simulink, we define empty C functions for each output event (e.g., for VP we define `SendWH_VP`). When the code is implemented on a particular hardware platform, the user needs to define these functions. For example, the simplest implementation would include toggling a particular CPU pin every time the function is invoked.

At the beginning of the chart procedure (List. 3, Fig. 10(a)) all counters associated with the event (stored in $\_sfEvent\_$) are increased. Since the pacemaker code uses only clk event in temporal logic operators, the five counters will be incremented only when clk is processed. After this, the functions associated with each of the parallel states are called in the order specified by the execution order.

List. 4 from Fig. 10(a) presents a pseudo-code for processing each of the parallel states. If the state is active, all transitions outgoing from its active substate are evaluated in the pre-specified execution order. The first enabled transition is taken and associated transition actions are executed. In the generated code only Eng state, which is executed last, is used to broadcast events as part of its transition actions. As shown in List. 5, Fig. 10(a), broadcasting an event associates the (current event) variable $\_sfEvent\_$ with the event, before it reactivates the chart (by calling `c1_ChartName()`).

A. Platform Implementation

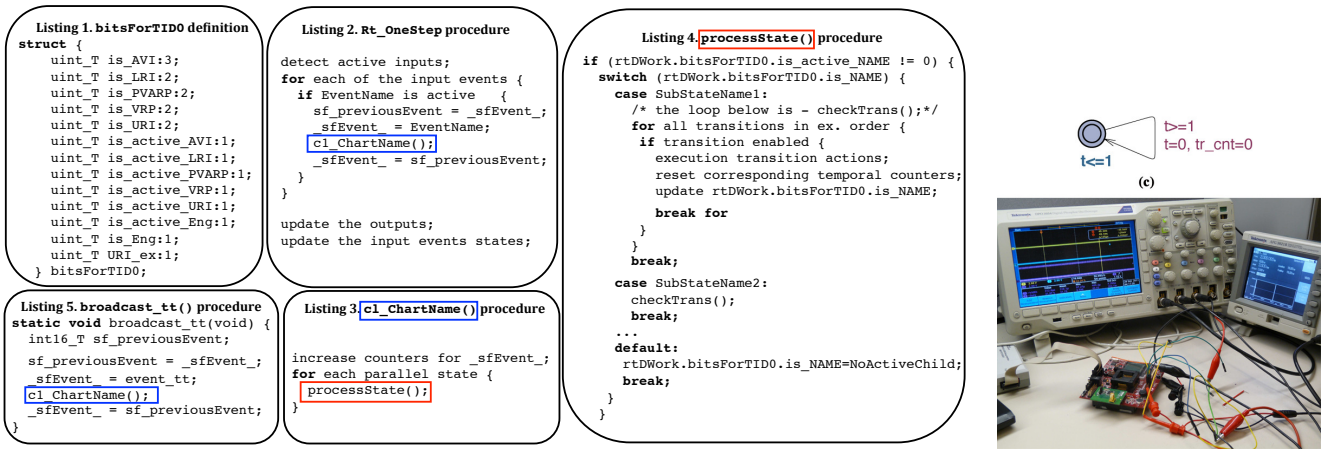
The pacemaker code has been executed on nanoRK [20], a fixed-priority preemptive RTOS that runs on a variety of resource constrained platforms (e.g., 8-bit Atmel-AVR, 16-bit TI-MSP430). We tested the implementation on the TI MSP-EXP430F5438 Experimenter Board interfaced with a signal generator that provides inputs for the pacemaker code (Fig. 10(b)). The compiled (without optimization) pacemaker procedure uses 2536 B for code and additional 180 B for data. To interface the code with the environment, each of the inputs (Ain, Vin) triggers an interrupt routine used to set the appropriate event for `rt_OneStep` function.

The pacemaker code was run as a task with period $1ms$. Table I shows measured execution times for the pacemaker tasks, for two different CPU frequencies. The measurements from Table I can be mapped to CPU utilization for the pacemaker task. With the average utilization of 9.2% for an 8MHz CPU, we can run multiple tasks on the RTOS.

CPU frequency	Average ex. time	Minimal ex. time	Maximal ex. time	Standard deviation
4MHz, OL	$176.1\mu s$	$167.6\mu s$	$462.9\mu s$	$14.2\mu s$
4MHz	$180.9\mu s$	$167.6\mu s$	$738.2\mu s$	$17.3\mu s$
8MHz, OL	$89.5\mu s$	$84.7\mu s$	$234.6\mu s$	$7.2\mu s$
8MHz	$92.0\mu s$	$84.9\mu s$	$370.4\mu s$	$13.7\mu s$

Table I
Execution times for the pacemaker procedure; OL denotes open-loop, without inputs from the signal generator.

[7] These issues do not present a problem for the pacemaker design from Fig. 8, since only a single AP or a single VP can be broadcast within one clk *execution*. However, in this paper we describe the general approach that allows utilization of the UPP2SF for all types of UPPAAL models.



(a) (b)

Figure 10. a) Structure of the pacemaker code obtained from the Stateflow chart shown in Fig. 8. b) Hardware setup, with MSP430F5438 experimenters board. c) Transition monitor - TrMonitor automaton.

B. Worst Case Execution Time Estimation in UPPAAL

Correctness of the generated code relies on the assumption that execution of the code completes before the next external activation. To make sure that it does, we need to estimate the WCET of the code execution, taking into account that the `c1_ChartName` (i.e., the chart) may be internally activated multiple times. We propose an approach that does not require translation from UPPAAL to Stateflow. Rather it uses the initial UPPAAL model to calculate an upper bound on the maximal number of internal activations N_i within an external activation (i.e., per clk execution).

To determine the bound for N_i we note that the chart is reactivated with event broadcasts and some transitions. Therefore, we extend the model with the following accounting features:

- Global variable tr_cnt and the automaton `TrMonitor` (Fig. 10(c)) that resets the variable at integer time points,
- In the controller part of the UPPAAL model (automata AVI, LRI, URI, PVARP, ARP in the pacemaker model) reset operation $tr\_cnt = tr\_cnt + 1$ is added to all edges with transmissions over a broadcast channel, or edges that would be translated into Stateflow transitions with $act = 1$ reset,
- Reset $tr\_cnt = tr\_cnt + 1$ to the edges with transmissions over broadcast channels that present inputs to the controller,
- Introduce UPPAAL temporal formula $A[\,]tr\_cnt \leq \tilde{N}_i$.

With the above changes the variable tr_cnt bounds the number of internal activations of the chart. Thus, if the above proposition is satisfied, the value $\tilde{N}_i + 1$ provides an upper bound for the number of chart executions within a single clk execution (1 is due to external activation). For the pacemaker UPPAAL model from Fig. 7 we added the reset operation to 8 transitions. We proved that the formula holds for $\tilde{N}_i = 5$.

VIII. Testing of the Physical Implementation

We validated the physical implementation using the procedure from Section VI-A. Unlike validation of the Stateflow chart, for physical testing we considered two types of tests. For the *ideal* system specifications we used $\epsilon \leq 80\mu s$, since $84.9\mu s$ was the chart's minimal execution time (Table I). Similarly, since all the predefined tolerances are $\pm 4ms$, for the second set of tests we used $4ms < \epsilon \leq 4.08ms$.

Table II presents testing results for the pacemaker implementation executed on the MSP430 experimenters board. When the tolerances are not taken into account some of the properties that were verified in UPPAAL and validated in Simulink were violated during the tests. The reason is that the UPPAAL semantics uses an unrealistic assumption that the machine executing the code is infinitely fast (i.e., no time elapses during transitions) and the system's reaction to synchronization is instantaneous. In the general case, the execution delays can cause violation of the UPPAAL semantics in the obtained physical implementation, which is the main reason for violation of some of the verified safety properties. However, when interval tolerances are taken into account, all properties were satisfied, as shown in Table II.

For example, consider the property **B4.2**. Fig. 11 presents one of the oscilloscope screenshots obtained during the testing. The signals shown are Ain (top), AS (middle) and clk (bottom). As shown, Ain appeared right after the first clk occurrence. It sets the appropriate flag in the interrupt routine, but the processing of the corresponding event occurred with the next clk. The event processing takes approximately $232\mu s$ before AS is generated. This, along with the time (up to $1ms$) between Ain and the

Requirement	P1.1	B1.1	B1.2	P2.1	P2.2	B2.1	B2.2
Ideal	Pass	Fail	Pass	Pass	Pass	Fail	Fail
With tolerance	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Requirement	P3.1	B3.1	B3.2	P4.1	B4.1	B4.2	
Ideal	Pass	Fail	Fail	Pass	Fail	Fail	
With tolerance	Pass	Pass	Pass	Pass	Pass	Pass	

Table II
Results of the tests performed on the setup from Fig. 10(b).

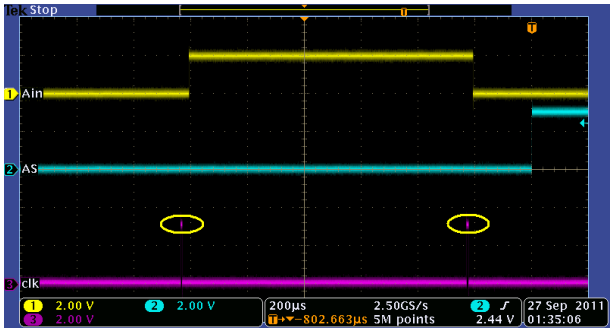


Figure 11. A test screen shot for property **B4.2**; clk pulses are highlighted.

following clk, results in delay of up to $1.232ms$. Thus, *ideal* requirement **B4.2** is violated. However, since the delay is within the tolerance bound, the requirement is satisfied when the tolerances are taken into account.

IX. Discussion

We have presented the design of UPP2SF, a model translation tool from UPPAAL to Simulink/Stateflow. The tool enables an MDD-based framework, with the model verification in UPPAAL, simulation and testing in Stateflow and automated code generation to C, C++, VHDL and Verilog. This has been demonstrated on a case study of an implantable pacemaker. We now discuss some open issues.

1) Decoupling the Controller and the Environment: In Section V-B we have described the method used to decouple models of the heart and pacemaker. However, our implementation introduces some problems regarding processing of input signals. By introducing an interrupt routine that sets a flag if the input occurs, we effectively synchronize asynchronous input signals. This has a twofold effect on the implemented code. First, each input signal will be processed at most once even if it appears more than one time between consecutive task's activations. This is not a problem with the pacemaker, since in the initial UPPAAL model, due to the buffers, all inputs after the first input in a cycle are disregarded until at least dL_a (or dL_v) time. Second, it introduces a latency of up to the task's period (in our case 1ms) before the input signals are processed. This problem occurs even if the code has been generated using Times or any other tool, since the number of clocks used in models is usually greater than the number of timers that CPU provides. A solution that for verification relaxes upper bounds on the buffer induced delays is proposed in [13].

2) Correctness of the UPP2SF Procedure: In this work we have shown that UPP2SF preserves behavior of the pacemaker model, and that the obtained Stateflow chart manifests the same behavior as the UPPAAL model. Although we developed UPP2SF using the 'correct by construction' approach, there is no formal proof of correctness for the derived procedure. Since the Stateflow semantics is informally defined (although there exist some attempts to derive formal semantics, e.g., [21]), establishing correctness of the translation procedure is not possible. Therefore, as a part of our ongoing work we use conformance testing to show that the translation procedure preserves behaviors on a set of representative examples.

References

[1] R. Alur, "Timed Automata," in *Computer Aided Verification*, 1999, vol. 1633, pp. 688–688.

[2] R. Alur *et al.*, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995.

[3] I. Lee *et al.*, "High-Confidence Medical Device Software and Systems," *IEEE Computer*, vol. 39, no. 4, pp. 33–38, 2006.

[4] "List of Device Recalls, U.S. Food and Drug Admin., (last visited Jul. 19, 2010)."

[5] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 134–152, 1997.

[6] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," in *Formal Methods for the Design of Real-Time Systems*, 2004, vol. 3185, pp. 33–35.

[7] K. Altisen and S. Tripakis, "Implementation of timed automata: An issue of semantics or modeling?" in *Formal Modeling and Analysis of Timed Systems*, 2005, vol. 3829, pp. 273–288.

[8] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: A tool for schedulability analysis and code generation of real-time systems," in *Formal Modeling and Analysis of Timed Systems*, 2004, vol. 2791, pp. 60–72.

[9] M. Hendriks, "Translating UPPAAL to Not Quite C," Computing Science Institute, Tech. Rep. CSI-R0108, 2001.

[10] F. Leitner and S. Leue, "Simulink Design Verifier vs. SPIN - a Comparative Case Study," in *FMICS'08: ERCIM Workshop on Formal Methods for Industrial Critical Systems*, 2008.

[11] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, "Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre," in *EMSOFT'04: ACM conf. on Embedded software*, 2004, pp. 259–268.

[12] T. Henzinger, Z. Manna, and A. Pnueli, "What good are digital clocks?" in *Automata, Languages and Programming*, 1992, vol. 623, pp. 545–558.

[13] M. Pajic, I. Lee, R. Mangharam, and O. Sokolsky, "UPP2SF: Translating UPPAAL models to Simulink," University of Pennsylvania, Tech. Rep., Oct 2011.

[14] "Matlab R2011a Documentation → Stateflow," http://www.mathworks.com/help/toolbox/stateflow.

[15] S. Barold, R. Stroobandt, and A. Sinnaeve, *Cardiac Pacemakers: Step by Step*. Blackwell Futura, 2004.

[16] "PACEMAKER System Specification," Boston Scientific, 2007.

[17] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam, "Modeling and Verification of a Dual Chamber Implantable Pacemaker," in *TACAS'12: 18th Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2012.

[18] Z. Jiang, M. Pajic, and R. Mangharam, "Cyber-physical modeling of implantable cardiac medical devices," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 122–137, 2012.

[19] D. Clarke and I. Lee, "Testing real-time constraints in a process algebraic setting," in *Proceedings of the International Conference on Software Engineering*, 1995, pp. 51–60.

[20] "nano-RK Sensor RTOS. http://nanork.org."

[21] G. Hamon and J. Rushby, "An Operational Semantics for Stateflow," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 447–456, 2007.