2014

Safety-Critical Medical Device Development Using the UPP2SF Model

Miroslav Pajic
University of Pennsylvania, pajic@seas.upenn.edu

Zhihao Jiang
University of Pennsylvania, zhihaoj@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Rahul Mangharam
University of Pennsylvania, rahulm@seas.upenn.edu

Recommended Citation

Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam, "Safety-Critical Medical Device Development Using the UPP2SF Model", *ACM Transactions on Embedded Computing Systems (TECS)* 13(4s). January 2014. http://dx.doi.org/10.1145/2584651

Special Issue on Real-Time and Embedded Technology and Applications, Domain-Specific Multicore Computing, Cross-Layer Dependable Embedded Systems, and Application of Concurrency to System Design (ACSD'13).

Safety-Critical Medical Device Development Using the UPP2SF Model

Abstract

Software-based control of life-critical embedded systems has become increasingly complex, and to a large extent has come to determine the safety of the human being. For example, implantable cardiac pacemakers have over 80,000 lines of code which are responsible for maintaining the heart within safe operating limits. As firmware-related recalls accounted for over 41% of the 600,000 devices recalled in the last decade, there is a need for rigorous model-driven design tools to generate verified code from verified software models. To this effect we have developed the UPP2SF model-translation tool, which facilitates automatic conversion of verified models (in UPPAAL) to models that may be simulated and tested (in Simulink/Stateflow). We describe the translation rules that ensure correct model conversion, applicable to a large class of models. We demonstrate how UPP2SF is used in the model-driven design of a pacemaker whose model is (a) designed and verified in UPPAAL (using timed automata), (b) automatically translated to Stateflow for simulation-based testing, and then (c) automatically generated into modular code for hardware-level integration testing of timing-related errors. In addition, we show how UPP2SF may be used for worst-case execution time estimation early in the design stage. Using UPP2SF, we demonstrate the value of integrated end-to-end modeling, verification, code-generation and testing process for complex software-controlled embedded systems.

Keywords

design, verification, measurement, model-based development, model translation, medical devices validation and verification, real-time embedded systems

Disciplines

Computer Engineering | Computer Sciences

Comments

Special Issue on Real-Time and Embedded Technology and Applications, Domain-Specific Multicore Computing, Cross-Layer Dependable Embedded Systems, and Application of Concurrency to System Design (ACSD'13).

Safety-critical Medical Device Development using the UPP2SF Model Translation Tool

MIROSLAV PAJIC, University of Pennsylvania
ZHIHAO JIANG, University of Pennsylvania
INSUP LEE, University of Pennsylvania
OLEG SOKOLSKY, University of Pennsylvania
RAHUL MANGHARAM, University of Pennsylvania

Software-based control of life-critical embedded systems has become increasingly complex, and to a large extent has come to determine the safety of the human being. For example, implantable cardiac pacemakers have over 80,000 lines of code which are responsible for maintaining the heart within safe operating limits. As firmware-related recalls accounted for over 41% of the 600,000 devices recalled in the last decade, there is a need for rigorous model-driven design tools to generate verified code from verified software models. To this effect we have developed the UPP2SF model-translation tool, which facilitates automatic conversion of verified models (in UPPAAL) to models that may be simulated and tested (in Simulink/Stateflow). We describe the translation rules that ensure correct model conversion, applicable to a large class of models. We demonstrate how UPP2SF is used in the model-driven design of a pacemaker whose model is (a) designed and verified in UPPAAL (using timed automata), (b) automatically translated to Stateflow for simulation-based testing, and then (c) automatically generated into modular code for hardware-level integration testing of timing-related errors. In addition, we show how UPP2SF may be used for worst-case execution time estimation early in the design stage. Using UPP2SF, we demonstrate the value of integrated end-to-end modeling, verification, code-generation and testing process for complex software-controlled embedded systems.

Categories and Subject Descriptors: C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems; J.3 [**LIFE AND MEDICAL SCIENCES**]: Health; D.2.10 [**Design**]: Methodologies

General Terms: Design, Verification, Measurement

Additional Key Words and Phrases: Model-based development, model translation, medical devices validation and verification, real-time embedded systems

1. INTRODUCTION

During the last decade, Model-Driven Development (MDD) has been widely used for design of real-time and cyber-physical systems (CPS). In the case of safety-critical systems, this methodology advocates for design procedures that start with formal modeling of the system, followed by the model's verification at an early stage. Since the initial input to the system designers is usually a set of informal specifications, this approach enables early detection of the specification and modeling errors.

For real-time systems, timed-automata [Alur 1999] are a commonly used modeling formalism, allowing designers to exhaustively explore the possible behaviors of the system and prove its safety. Design of CPS is more complex, since these systems feature a tight coupling between the real-time discrete controller and (usually) continuous physical environment. For the systems' verification, it is necessary to provide the model of the closed-loop system, which also takes into account the interaction between the controller and the environment. Although in the general case this interaction can be modeled as a hybrid system, the complexity of this approach usually renders it out of reach of current verification tools [Alur et al. 1995]. Thus, in the initial design stage, timed-

Authors' addresses: M. Pajic and R. Mangharam, Department of Electrical & Systems Engineering, University of Pennsylvania, 200 S. 33rd Street, Philadelphia, PA 19104; email: {pajic,rahulm}@seas.upenn.edu; Z. Jiang, I. Lee and O. Sokolsky, Department of Computer & Information Science, University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104; email: {zhihaoj, lee, sokolsky}@cis.upenn.edu;

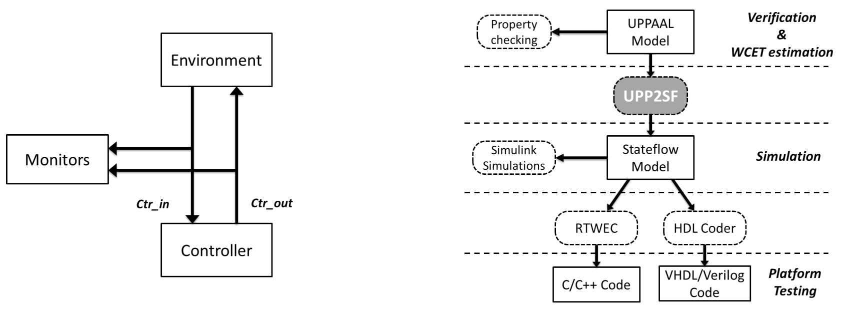


Fig. 1. (a) The standard model architecture used for verification; (b) MDD-based framework: From UPPAAL to Stateflow to generated code – covering model verification, simulation-based testing and platform testing.

automata models that abstract away complex system dynamics, replacing it with timing constraints, are commonly used for verification of CPS (e.g., [Pajic et al. 2012c]).

Since parts of the verified models represent over-approximations of the realistic models, in the later stages of MDD, detailed models of the environment and its interaction with the controller are developed. These models enable high-fidelity system simulation with real system dynamics, and are used to validate the initial assumptions used in the verification stage. During the validation procedure, it is necessary to ensure equivalence between the controller models used for verification and simulation. Therefore, it is essential to provide guarantees that the properties verified in the early stage are still satisfied, as the system model gets complex and more refined.

Finally, the verified model must be translated into executable code for physical implementation, which is then validated using different testing procedures that have been built on the initial system's specification. However, since the verification is performed on a closed-loop system model, the verified model usually has the structure as the one shown in Figure 1(a), meaning that, besides the controller model, it contains model of the environment (which includes the interface controller $\leftrightarrow$ environment). Hence, to obtain valid controller code it is necessary to decouple the controller code from the code synthesized for the whole closed-loop model. In scenarios with complex interfaces between the controller and environment, this can be a very tedious and error prone process. Consequently, to prevent introduction of errors during the decoupling it is necessary to provide a procedure for *modular code synthesis*.

In this work, we present a model-translation tool, UPP2SF, and show how it enables a model-driven design framework for safety-critical real-time embedded systems (Figure 1(b)). UPP2SF facilitates automatic conversion of verified models (in UPPAAL) to models that may be simulated and tested (in Simulink/Stateflow). By using Simulink support for code generation, this allows for automatic end-to-end model translation across multiple levels of abstraction to modular code synthesis. Consequently, the proposed framework ensures that successive models are consistent through the development process: (a) a timed-automata based UPPAAL modeling of the controller software, the model verification and model-based worst case execution time (WCET) estimation; (b) automatic translation of the model to Stateflow using the developed UPP2SF tool; (c) testing of the Stateflow model; and (d) automatic modular code generation, test generation and testing of timing related errors on the hardware platform.

UPPAAL [Larsen et al. 1997; Behrmann et al. 2004] is a standard free tool for modeling and verification of real-time systems, based on networks of timed automata. Thus, it does not support simulation of continuous-time dynamics. Although there exists some work on code generation from timed-automata models (e.g., [Altisen and Tripakis 2005; Amnell et al. 2004]), there are only few tools [Amnell et al. 2004; Hendriks

2001] with limited capabilities in generating C code from UPPAAL models (see Section 2.2). Furthermore, none of these tools supports modular code generation from UPPAAL models. On the other hand, Simulink is a commercially available tool, 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 limited success with model verification [Leitner and Leue 2008; Schürenberg 2012], and (more importantly), due to Simulink's deterministic execution semantics, it currently does not support verification of non-deterministic systems; this is a significant limitation in closed-loop CPS, where behavior of the environment is usually nondeterministic.

In this paper, we demonstrate the use of the framework on a detailed case study for the development of a safety-critical system medical system – an *implantable cardiac pacemaker*. We focus on the pacemaker case study as there is a need to develop a methodology for the design and analysis of safety critical closed-loop medical device software and systems [Lee et al. 2006]. From 1990-2000, firmware issues resulted in over 200,000 implantable device recalls [rec 2010]. From 1996-2006, the percentage of software-related causes in medical device recalls have grown from 10% to 21% [rec 2010]. Furthermore, during the period January-June of 2010, at least six recalls of defective devices, issued by the US Food and Drug Administration (FDA), were caused by some sort of software issues [Sandler et al. 2010]. These recalls were categorized as *Class I*, which means that there is a "reasonable probability that use of these products will cause serious adverse health consequences or death."

We start the case study from informal pacemaker requirements [Boston Scientific 2007], which we formalize, and use for verification by designing an appropriate set of monitors in UPPAAL. We also show how a WCET estimation can be done in the early stage in UPPAAL, before the verified model is translated to Simulink. Furthermore, we illustrate the approach used toward modular code synthesis – extracting the controller's Stateflow model by decoupling the controller (i.e., pacemaker) from environment (i.e., the heart). Finally, we extend the method for testing real-time requirements from [Clarke and Lee 1995] to validate the physical implementation of the device.

The paper is organized as follows: in Sec. 2 and 4, we present brief overviews of UPPAAL and Stateflow. In Sec. 3, we show that for a large class of UPPAAAL models, a run can be obtained by evaluating transitions at integer time-points only. Sec. 5 presents the model translation procedure, while in Sec. 6 we show that the UPP2SF-derived chart has the execution trace that corresponds to one of the maximal progress assumption runs of the initial UPPAAL model. Sec. 7 introduces the pacemaker case study and presents a set of formal pacemaker specifications, which we verified in UPPAAL using the developed pacemaker model (Sec. 7.2). Finally, we present the obtained Stateflow closed-loop system model and the procedure used to decouple the controller and environment (Sec. 8), followed by device implementation on an RTOS (Sec. 9).

1.0.1. Notation. We use the standard notation where $\mathbb{Z}, \mathbb{N}, \mathbb{N}_0$ denote the sets of integers, natural numbers and nonnegative integers, respectively, while $\mathbb{R}^+$ is the set of nonnegative reals. Finally, $\lfloor z \rfloor$ denotes the largest integer that is not larger than z.

2. A BRIEF OVERVIEW OF UPPAAL

In this section, we present an overview of the UPPAAL tool, including some of the UPPAAL extensions [Larsen et al. 1997; Behrmann et al. 2004; Bengtsson and Yi 2004] of the timed-automata formalism from [Alur 1999]. Also, to illustrate the need for a tool like UPP2SF we survey existing tools for code synthesis from UPPAAL models.

2.1. UPPAAL Modeling of Real-time Systems

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 to integer values (possibly non-zero).

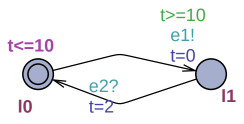
Automata in the network execute concurrently. They can communicate via shared variables, as well as via events over synchronous channels. If c is a channel, $c?$ represents receiving an event from c, while $c!$ stands for sending an event on c. In the general case, an edge from location l_1 to location 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.

2.1.1. Timed-Automata Semantics in UPPAAL. We denote with C the set of all clocks and with V the set of all data (i.e., boolean and integer) variables. 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 $x \in C$. Similarly, a data valuation is a function $v : V \to \mathbb{Z}$, while $\mathbb{Z}^V$ denotes the set of all data valuations. A valuation w, denoted by $w = (u, v)$, is a function $w : C \times V \to \mathbb{R}^+ \times \mathbb{Z}$ such that $w(x, i) = (u(x), v(i))$, for clock valuation u and data valuation v. Also, for a valuation $w = (u, v)$, $w+d$ denotes the valuation where $(w + d)(x) = (u + d)(x) = u(x) + d$ for $x \in C$, and $(w + d)(i) = v(i)$ for $i \in \mathbb{Z}$. In the rest of the paper, a clock valuation u that satisfies that for all $x \in C$, $u(x) \in \mathbb{N}_0$ will be referred to as *integer clock valuation*, while a valuation $w = (u, v)$, where u is an integer clock valuation will be referred to as an *integer valuation*.

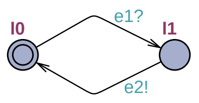
Furthermore, let $B(C, V)$ denote the set of conjunctions over simple clock and variable conditions of the form $x \bowtie n$, $x - y \bowtie n$ or $i - j \bowtie k$, where $n \in \mathbb{N}$, $x, y \in C$, $i, j \in V$, $k \in \mathbb{Z}$ and $\bowtie \in \{\leq, \geq, =, <, >\}$.[1] Similarly, $B(C)$ denotes the set of all conjunctions over the clock variable conditions. Thus, a guard can be defined as an element of $B(C, V)$.[2] Reset operations are used to manipulate clocks and data variables. They have the form $x = n$ or $i = c_1 * j + c_2$, where $x \in C$, $i, j \in V$, $c_1, c_2 \in \mathbb{Z}$ and $n \in \mathbb{N}_0$. We use R to denote the set of all possible reset operations, and for a reset operation $r \in R$ and valuation w, $r(w)$ is the valuation obtained from w where all clocks and data variables specified in r are assigned to the values obtained from the appropriate expressions. Finally, we use K to denote the set of all channels, and $A = \{\alpha?|\alpha \in K\} \cup \{\alpha!|\alpha \in K\} \cup \{\tau\}$ to denote the set of all actions. Here, τ denotes an 'empty' action – without synchronization.

Definition 2.1. An automaton $\mathcal{A}$ is a tuple (L, l_0, A, C, V, 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, V is a set of data variables, and $E \subseteq L \times A \times B(C, V) \times R \times L$ denotes the set of edges, while I assigns invariants to locations (i.e., $I : L \to B(C)$ is a mapping of each location to a constraint over some clocks).

[1]In the latest UPPAAL versions n can also denote an expression over integer variables. Since all results from this section are valid in the latter case, we use the simplified notation where n is a constant integer.
[2]The default guard is *true*.



(a) P0 automaton



(b) P1 automaton

$$\left(\begin{bmatrix} P0.l0 \\ P1.l0 \end{bmatrix}, \begin{bmatrix} P0.t{=}0 \\ P1.t{=}0 \end{bmatrix}\right) \rightarrow \left(\begin{bmatrix} P0.l0 \\ P1.l0 \end{bmatrix}, \begin{bmatrix} P0.t{=}10 \\ P1.t{=}10 \end{bmatrix}\right) \rightarrow \left(\begin{bmatrix} P0.l1 \\ P1.l1 \end{bmatrix}, \begin{bmatrix} P0.t{=}0 \\ P1.t{=}10 \end{bmatrix}\right) \rightarrow \left(\begin{bmatrix} P0.l1 \\ P1.l1 \end{bmatrix}, \begin{bmatrix} P0.t{=}2.4 \\ P1.t{=}12.4 \end{bmatrix}\right) \rightarrow \left(\begin{bmatrix} P0.l0 \\ P1.l0 \end{bmatrix}, \begin{bmatrix} P0.t{=}2 \\ P1.t{=}12.4 \end{bmatrix}\right) \rightarrow \cdots$$

(c) A run of the model

Fig. 2. An UPPAAL model example.

If a clock valuation u satisfies the invariants at location l, we abuse the notation and write $u \in I(l)$. Similarly, we denote with $w \in I(l)$, if $w = (u, v)$ and $u \in I(l)$. Also, if a valuation w satisfies a condition $g \in B(C, V)$ we write $w \in g$.

A network of n timed automata is obtained by composing $\mathcal{A}_i = (L_i, l_i^0, C, A, V, E_i, I_i)$, $i \in \{1, ..., n\}$. In this case, a location vector is defined as $\bar{l} = (l_1, l_2, ..., l_n)$. In addition, the 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 the notation $\bar{l}[l'_i/l_i]$.

Definition 2.2. 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 of the network is defined as a transition system $\langle \mathcal{S}, s_0, \rightarrow \rangle$, where $\mathcal{S} = (L_1 \times L_2 \times ... \times L_n) \times (\mathbb{R}^C \times \mathbb{Z}^V)$ is the set of states, $s_0 = (\bar{l}_0, w_0)$ is the initial state, where $w_0 = (u_0, v_0)$ and v_0 is any initial data valuation, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation defined by:

(1) $(\bar{l}, w) \rightarrow (\bar{l}, w + d)$ if for all d' such that $0 \le d' \le d$, it follows that $w + d' \in I(\bar{l})$;
(2) $(\bar{l}, w) \rightarrow (\bar{l}[l'_i/l_i], w')$ if exists $l_i \xrightarrow{g, \tau, r} l'_i$ such that $w \in g$, $w' = r(w)$ and $w' \in I(\bar{l}[l'_i/l_i])$;
(3) $(\bar{l}, w) \rightarrow (\bar{l}[l'_j/l_j, l'_i/l_i], w')$ if there exist $l_i \xrightarrow{g_i, c?, r_i} l'_i$ and $l_j \xrightarrow{g_j, c!, r_j} l'_j$ such that $w \in (g_i \wedge g_j)$, $w' = (r_i \cup r_j)(w)$ and $w' \in I(\bar{l}[l'_j/l_j, l'_i/l_i])$.

Since the first type of transitions is the result of time-passing, unless otherwise stated, in the rest of the paper when we use the term UPPAAL transition we will refer to either case (2) or (3) of the above definition.

To illustrate the above definition, consider the model from Figure 2, where both automata have separate local clocks t. Location $P0.l0$ has invariant $t \le 10$, while the edge $P0.l0 \rightarrow P0.l1$ has the guard condition $t \ge 10$, reset action $t = 0$, and transmission over channel $e1$. Hence, synchronization over the channel $e1$ ensures that the transitions $P0.l0 \rightarrow P0.l1$ and $P1.l0 \rightarrow P1.l1$ occur simultaneously. Finally, note that clocks do not have to be reset to zero (e.g., as on the edge $P0.l1 \rightarrow P0.l0$).

For semantics $\langle \mathcal{S}, s_0, \rightarrow \rangle$, a sequence $\mathcal{R} := (\bar{l}_0, w_0) \rightarrow (\bar{l}_1, w_1) \rightarrow ... \rightarrow (\bar{l}_i, w_i) \rightarrow ...$, is called a *run*, and we use notation $w_k^{\mathcal{R}} = w_k$, $\bar{l}_k^{\mathcal{R}} = \bar{l}_k$, for all $k \ge 0$. An example run for the model from Figure 2 is shown in Figure 2(c). To simplify the notation, we assume that in each run $\mathcal{R}$ no two consecutive transitions are result of time-elapsing (case (1) of Def. 2.2), since these transitions can be merged into a single transition of that type.

2.1.2. Additional UPPAAL Extensions of the Timed-Automata Formalism. Beside integer variables and synchronization channels, UPPAAL extends timed-automata with *committed* and *urgent* locations where time is not allowed to pass (i.e., no delay is allowed). *Committed locations* are more restrictive and they are usually used to model atomic sequences of actions. 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 (e.g., zero, one, or more than one). Furthermore, *urgent channels* can be used for synchronization, to specify that if a transition with synchronization over an urgent channel is enabled, then the transition should occur without any delay.

2.2. Code Synthesis from UPPAAL Models

There are only few tools that can be used for automatic implementation of timed-automata models designed in UPPAAL (e.g., [Amnell et al. 2004; Hendriks 2001]). A commonly used tool is Times [Amnell et al. 2004] that supports code generation for general platforms, extended with task support for Lego Mindstorm$^{\text{TM}}$platform.

The code synthesized using Times has a very simple structure, where all transitions are stored in an array. Each transition is represented with four fields: an activity flag, source and destination location ids, and a synchronization id. The transitions are evaluated in automatically generated `check_trans` function, and for the code to operate correctly the values for all clocks should be updated in a separate procedure, triggered by the system timers. Since `check_trans` performs a single evaluation for each transition (in the order specified by the array structure), to ensure that no transitions are missed the `check_trans` function has to be continuously invoked within an infinite loop, unless the code is executed on a LEGO Mindstorm RCX brick running brickOS. Thus, in the general case, the code generated with Times completely utilizes the CPU, disallowing instantiation of any other tasks. As we will show later in the paper, this is not the case for the code that is synthesized using our development framework.

Due to this array-based structure, with the code obtained using Times it is not straightforward to decouple the controller (in our case the pacemaker) and the environment (e.g., the heart) in scenarios described in Figure 1(a). For example, to facilitate the decoupling [Kim et al. 2011] propose a modification of the initial UPPAAL model by specifying the interaction between the controller and the environment using boolean shared variables. Although the solution preserves behaviors of the initial model, as pointed out by the authors, this type of manual modifications is effectively one of the most error prone aspects of the model-based development. Hence, to avoid this type of errors, it is necessary to provide modular code synthesis from verified UPPAAL models. We will later show that UPP2SF resolves this issue, since the obtained code has a modular structure (instead of maintaining an array of transitions).

Finally, the code generated from Times models does not guarantee that some aspects of the UPPAAL timed-automata semantics will be preserved [Ayoub et al. 2010]. For example, it does not ensure that the requirement for committed states is satisfied.

3. EXTRACTING RUNS FROM UPPAAL MODELS

To develop the UPP2SF model translation tool, we consider the problem of extracting runs for UPPAAL models. 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 modeling of real control system, guarantees that all invariants and guards are expressed as intersections of left semi-closed (LSC) intervals. Thus, we refer to this class as *Class LSC*, and in this work we consider only this type of models.

To obtain a run of an UPPAAL model it is sufficient to simulate the model only at integer time points [Pajic et al. 2012a], which allows for the use of discrete-time based tools for model simulation. Since the execution of UPPAAL models is non-deterministic, a transition in UPPAAL can occur at any point at which it is enabled. However, at each time instance it may not be straightforward to determine whether a currently enabled transition will still be enabled at the next integer-time point. Therefore, to simplify the translation procedure we consider runs of UPPAAL models that are obtained using the maximal progress assumption (MPA).

Definition 3.1. For semantics $\langle \mathcal{S}, s_0, \rightarrow \rangle$, a run $\mathcal{R} := (\bar{l}_0, w_0) \rightarrow ... \rightarrow (\bar{l}_i, w_i) \rightarrow ...$, satisfies the maximal progress assumption (MPA) if for all $k \geq 0$ such that

(1) $\bar{l}_{k+1} = \bar{l}_k$ and $w_{k+1} = w_k + d$ for some $d > 0$, and

(2) $(\bar{l}_{k+1}, w_{k+1}) \to (\bar{l}_{k+2}, w_{k+2})$ satisfies either case (2) or (3) of Def. 2.2,

there does not exist a d', $0 \leq d' < d$, for which there exist a transition $(\bar{l}_k, w_k + d') \to (\bar{l}_k[l'_j/l_j], w')$ or a transition $(\bar{l}_k, w_k + d') \to (\bar{l}_k[l'_p/l_p, l'_q/l_q], w')$ for the given semantics.

A run that satisfies the MPA will be referred to as an *MPA run*. Note that from the definition, if some transitions in an MPA run are enabled, one of them should occur.

THEOREM 3.2. *Consider an UPPAAL model from the Class LSC. For every MPA run $\mathcal{R}$ of the model and for all $k \geq 0$, the clock valuation $u_k^{\mathcal{R}}$ satisfies that for each clock x, $u_k^{\mathcal{R}}(x) \in \mathbb{N}_0$ (i.e., all transitions of $\mathcal{R}$ occur at integer time points).*

PROOF. Lets assume that the theorem does not holds – i.e., there exists an MPA run $\mathcal{R}$ for which there exists $k \geq 0$ and a clock x such that $u_k^{\mathcal{R}}(x) \notin \mathbb{N}_0$. By k_0 we denote the first (i.e., lowest) such k. Since all clocks are initialized to zero, $k_0 > 0$ and $w_{k_0-1} = (u_{k_0-1}, v_{k_0-1})$ is an integer valuation. Thus, consider the part $(\bar{l}_{k_0-1}, w_{k_0-1}) \to (\bar{l}_{k_0}, w_{k_0}) \to (\bar{l}_{k_0+1}, w_{k_0+1})$ of the run $\mathcal{R}$. From Def. 2.2 one of the following cases is valid:

(1) $(\bar{l}_{k_0-1}, w_{k_0-1}) \to (\bar{l}_{k_0}, w_{k_0})$ satisfies case (1) of Def. 2.2 (i.e., time passing). Then from the definition of a *run*, the transition $(\bar{l}_{k_0}, w_{k_0}) \to (\bar{l}_{k_0+1}, w_{k_0+1})$ is described by either case (2) or (3) from Def. 2.2. In the former case $\bar{l}_{k_0+1} = \bar{l}_{k_0}[l'_i/l_i]$, meaning that there exists $l_i \xrightarrow{g,\tau,r} l'_i$ such that $w_{k_0} = (u_{k_0}, v_{k_0}) \in g$, $w_{k_0+1} = r(w_{k_0})$, and $w_{k_0+1} \in I(\bar{l}_{k_0+1})$. Consider the valuation $w'_{k_0} = \lfloor w_{k_0} \rfloor = (\lfloor u_{k_0} \rfloor, v_{k_0})$. Since $w_{k_0} \in g$ then w'_{k_0} also satisfies all variable conditions from g. For each clock x, if $u_{k_0}(x)$ satisfies the clock guard conditions of the form $x \bowtie n$, where $n \in \mathbb{N}_0$ and $\bowtie \in \{\leq, \geq, =, <\}$, then $u_{k_0}(x)$ belongs to an intersection of left-closed intervals with integer boundaries. Thus, $\lfloor u_{k_0}(x) \rfloor$ belongs to the same intersection of the intervals, meaning that $u'_{k_0} = \lfloor u_{k_0} \rfloor$ satisfies this type of clock constraints from g. Furthermore, if for some clocks x and y, valuation u_{k_0} satisfies constraints of the form $x - y \bowtie n$, it follows that $u_{k_0}(x) - u_{k_0}(y) \bowtie n$. From $u_{k_0} = u_{k_0-1} + d$ (where d is the elapsed time), we have: $u'_{k_0}(x) - u'_{k_0}(y) = \lfloor u_{k_0}(x) \rfloor - \lfloor u_{k_0}(y) \rfloor = u_{k_0-1}(x) + \lfloor d \rfloor - (u_{k_0-1}(y) + \lfloor d \rfloor) = u_{k_0-1}(x) - u_{k_0-1}(y) = u_{k_0-1}(x) + d - u_{k_0-1}(y) - d = u_{k_0}(x) - u_{k_0}(y)$. Thus, $u'_{k_0}(x) - u'_{k_0}(y) \bowtie n$ is true, and all clock constraints of this form are also satisfied, implying that $w'_{k_0} \in g$. Similarly, for $w'_{k_0+1} = r(w'_{k_0}) = (u'_{k_0+1}, v'_{k_0+1})$, from $w'_{k_0} = \lfloor w_{k_0} \rfloor$ and $w_{k_0+1} = r(w_{k_0})$ it follows that $v'_{k_0+1} = v_{k_0+1}$ and $u'_{k_0+1} = \lfloor u_{k_0+1} \rfloor$ (all reset clocks are equal, since clocks are reset to integer values). Therefore, as for the guard, it can be shown that $w_{k_0+1} \in I(\bar{l}_{k_0+1})$ implies that $w'_{k_0+1} \in I(\bar{l}_{k_0+1})$, which proves that the transition $(\bar{l}_{k_0}, w'_{k_0}) \to (\bar{l}_{k_0+1}, w'_{k_0+1})$ was also enabled. Since $w'_{k_0}(x) < w_{k_0}(x)$ it follows that in this case $\mathcal{R}$ is not an MPA run, which violates our initial assumption.
A similar proof can be used in the latter case when $(\bar{l}_{k_0}, w_{k_0}) \to (\bar{l}_{k_0+1}, w_{k_0})$ satisfies case (3) of Def. 2.2, by showing the existence of a transition enabled for $\lfloor u_{k_0} \rfloor$. Since $\lfloor u_k(x) \rfloor < u_k(x)$, the transition $(\bar{l}_{k_0}, w_{k_0}) \to (\bar{l}_{k_0+1}, w_{k_0+1})$ cannot be in an MPA run.

(2) $(\bar{l}_{k_0-1}, w_{k_0-1}) \to (\bar{l}_{k_0}, w_{k_0})$ satisfies either case (2) or (3) of Def. 2.2. Then, there exists a transition with reset r such that $w_{k_0} = r(w_{k_0-1})$, or a synchronized transition with resets r_i, r_j such that $w_{k_0} = (r_i \cup r_j)(w_{k_0-1})$. However, since w_{k_0-1} is an integer valuation, in both cases $u_{k_0}(x) \in \mathbb{N}$ because no clock can be reset to a non-integer value. This conflicts our initial assumption, and thus concludes the proof.

□

The theorem presents the basis for the translation procedure. It allows us to obtain an MPA run by evaluating transitions from active locations in each automaton only at integer time points. Note that each automaton may be evaluated more than once, as more than one transition could occur within a single automaton at any integer time.

Theorem 3.2 can be easily extended for UPPAAL models with committed and urgent locations, and urgent and broadcast channels. For example, urgent locations can be modeled by adding an extra clock x_u that is reset to zero on all incoming edges to urgent locations, and adding condition $x_u \leq 0$ to invariants in all urgent locations. Yet, this does not affect the proof of Theorem 3.2, and thus the theorem is still valid. In addition, semantics for UPPAAL models that employ urgent channels is similar to the semantics from Def. 2.2, with an additional condition in case (1) of the definition. In this case $(\bar{l}, w) \to (\bar{l}, w + d)$, if for all $d', 0 \leq d' < d$ it holds that $w + d' \in I(\bar{l})$, and for any urgent channel c there does not exist $l_i \xrightarrow{g_i, c?, r_i} l'_i$ and $l_j \xrightarrow{g_j, c!, r_j} l'_j$ such that $w + d' \in (g_i \wedge g_j)$ and $(r_i \cup r_j)(w + d') \in I(\bar{l}[l'_j/l_j, l'_i/l_i])$. Similarly, broadcast channels semantics does not require that exactly one transition with receiving channel occurs simultaneously with a transition that contains a transmission over the channel – with broadcast channels none, one or more than one 'receiving' transitions could occur. Thus, Theorem 3.2 is also satisfied even if urgent and broadcast channels are used.

4. BRIEF OVERVIEW OF STATEFLOW

A Stateflow chart (i.e., model) employs a concept of finite state machines extended with additional features, including support for different data types and events that trigger actions in a part or the whole chart. Here, we present a small subset of the Stateflow features used in the translation procedure. Detailed descriptions of other features can be found in [sta 2012; Scaife et al. 2004; Hamon and Rushby 2007; Hamon 2005].

A state in a Stateflow chart can be active or inactive, and the activity dynamically changes based on events and conditions. States can be defined hierarchically – i.e., a state can contain other states (referred to as *substates*). A decomposition of a chart (or a 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 (but executed sequentially).

Unlike in UPPAAL, transitions between states in Stateflow are taken as soon as enabled. They are described in the form (where each part of the description is optional)

$$Event[condition]\{condition\_actions\}/\{transition\_actions\} \tag{1}$$

$Event$ identifies the event that enables the transition (which is enabled by default if $Event$ is not stated), if the $condition$ (if specified, by default it is true) is valid. The $condition$ is described using basic logical operations on conditions over chart variables and Stateflow operators. Actions in $condition\_actions$ and $transition\_actions$ include event broadcasting and operations on data variables. The Stateflow semantics specifies that when a transition from a state s_i to state s_j occurs, then $condition\_action$ are executed first, before the state s_i becomes inactive. This is followed by the execution of $transition\_actions$, and finally activation of the state s_j (i.e., during the execution of $transition\_actions$ none of the states is active).

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 (and similarly, all parallel substates in 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

– Stateflow semantics specifies that active states are scheduled, and state transitions are evaluated in the execution order (starting from the lowest execution index).[3]

4.0.1. Notion of Time in Stateflow. Stateflow temporal logic can be used to control execution of a discrete-time chart in terms of time. It defines time periods using absolute-time operators based on the simulation time, or event-based operators that use the number of event occurrences. Absolute-time logic defines operators $after$, $before$ as

$$after(n,\text{sec}) = \begin{cases} 0, & \text{if } t < n \\ 1, & \text{if } t \geq n \end{cases}, \qquad before(n,\text{sec}) = not(after(n,\text{sec})) \tag{2}$$

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 $temporalCount(\text{sec})$. Similarly, event-based temporal logic operators are used for event counting – e.g., $after(n, clk)$ returns 1 if the event clk has occurred more than $(n-1)$ times after the state has been activated.

5. UPP2SF: MODEL TRANSLATION PROCEDURE

In this section, we present an overview of the UPP2SF translation procedure. We also describe the translation rules for UPPAAL models with urgent and broadcast channels, urgent and committed locations, and local clocks, as these functionalities are used for the pacemaker modeling. For the full UPP2SF description refer to [Pajic et al. 2012b].

5.1. Overview of UPP2SF

Consider an UPPAAL model with automata $P_1, ..., P_n$. The UPP2SF translation procedure would produce a two-level Stateflow chart as in Figure 3, with parallel states $P_1, ..., P_n$ (referred to as the *parent* states) derived from the automata, parallel states $Gc\_x_1, ..., Gc\_x_m$ (referred to as *clock states*) that model all global clocks $x_1, ..., x_m$ from the UPPAAL model,[4] and the state Eng that is used as the chart's control execution engine. In addition, the chart has predefined global data variables (and constants) with appropriate variable ranges and initial values obtained from the UPPAAL model. Since all automata in UPPAAL are simultaneously active, the obtained Stateflow chart is a collection of parallel states with unique execution orders. Also, in every UPPAAL automaton exactly one location is active at a time. Thus, each of the parent states is a collection of exclusive states, extracted from locations in the UPPAAL automaton.

To ensure that the extracted chart is simulated at integer time points, input trigger event clk is added to the chart and a signal generator block is added to the parent Simulink model. We call a clk *execution* the execution of the chart from the moment the chart is triggered by a clk event, until processing of the event has been finished. Since our goal is to derive a Stateflow chart whose execution is one of the MPA runs of the initial UPPAAL model, it is possible that more than one transition within the model (and even within a single automaton) occur at any time point. Therefore, the chart can (re)activate itself by transmitting local (within the scope of the chart) events from the additional parallel state Eng, which is executed last of all chart's parallel states. Processing of the events triggered during a clk *execution* is considered a part of the clk *execution*. Since event processing is atomic in Stateflow, no time elapses (in Simulink) during a clk *execution* regardless how many additional event broadcasts have occurred. With this approach, a single activation of the chart triggers all transitions enabled at that integer time point, effectively extracting an MPA execution trace of the model.

[3]The user can specify the execution index for each transition and state – default values are assigned by the order of instantiation. Parallel states (or transitions from a state) must have different execution indices.
[4]Note that if no global clocks are used in the UPPAAL model, the obtained Stateflow chart would not contain parallel global clocks states Gc_x_j.

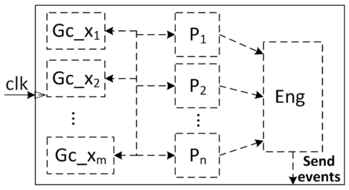


Fig. 3. Structure of Stateflow charts derived by UPP2SF. Parent states $P_1, ..., P_n$ are derived from automata, while the *clock* states $Gc\_x_1, ..., Gc\_x_m$ model all global clocks $x_1, ..., x_m$ from the UPPAAL model. The state Eng is used to control execution of the chart.

Finally, any UPPAAL edge from location l_{k_i} to l_{k_j} in any automaton P_k, which does not use global clocks and synchronization over binary channels, is mapped into a Stateflow transition $P_k.l_{k_i} \rightarrow P_k.l_{k_j}$ between the corresponding substates in the parent state P_k. In the rest of the section we provide a description of the edge translation procedure.

Remark 5.1. In the general case, the edge $l_{k_i} \xrightarrow{g,\alpha,r} l_{k_j}$ (where $\alpha \in \{\tau, c!, c?\}$ and c is a binary or broadcast channel) is mapped into a more complex structure in Stateflow between the substates $P_k.l_{k_i}$ and $P_k.l_{k_j}$. If the edge uses global clocks then a junction J_{ij} and transition $P_k.l_{k_i} \rightarrow P_k.J_{ij}$ are introduced to update global clock values used in the edge's guard and invariants. Also, if α does not use a binary channel, a single transition is introduced from J_{ij} to $P_k.l_{k_j}$. However, if α uses a binary channel, to preserve the semantics three edges and a junction are added between J_{ij} and $P_k.l_{k_j}$.

5.2. Mapping UPPAAL Edges Without Synchronization

Consider an UPPAAL edge $l_{k_i} \xrightarrow{g,\tau,r} l_{k_j}$ in automaton P_k. The guard g can be split into a conjunction of data and clock conditions, and thus during the translation UPP2SF introduces a Stateflow transition $P_k.l_{k_i} \rightarrow P_k.l_{k_j}$ of the form:

$$\underbrace{[G_C(I(l_{k_i}) \wedge g) \wedge G_V(g) \wedge G_C(r, I(l_{k_j}))]}_{G_{C,V}(l_{k_i}, l_{k_j}, g, r)} / \{\underbrace{R_V(r); R_C(r);}_{R_{C,V}(r)} R_S(r);\} \qquad (3)$$

where:

(1) $G_C(h)$ ($G_V(h)$) translates the clock (data) conditions from UPPAAL condition h into an equivalent Stateflow condition,
(2) $R_C(r)$ ($R_V(r)$) maps clock (data) resets in r to an equivalent Stateflow assignment,
(3) $G_C(r, I(l_{k_j}))$ maps the condition that the clock valuation after the reset r satisfies the invariant at the 'new' location l_{k_j},
(4) $R_S(r)$ controls execution of the chart.

Data resets ($R_V(r)$) and guard conditions ($G_V(g)$) are directly mapped into the identical Stateflow expressions. Mapping local clocks' resets and guards is described below.

5.2.1. Mapping Clock Conditions and Resets. In UPPAAL, each clock condition $h \in B(C)$ is specified as $h = h_1 \wedge h_2 \wedge ... \wedge h_M$, where h_i's are basic clock conditions. Therefore, $G_C(h) = G_C(h_1) \wedge G_C(h_2) \wedge ... \wedge G_C(h_M)$, and it is only necessary to provide a set of rules for the translation of basic clock conditions of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C$ and $n \in \mathbb{Z}$ (or an expression over integer variables and constants).

To specify conditions over clocks, UPP2SF employs event based Stateflow temporal logic operators that (only) count the number of clk event occurrences. When the temporal logic operators are used in a chart with the two-level hierarchy shown in Figure 3, Stateflow associates a unique counter with each parallel (i.e., parent) state. It is important to highlight here that Stateflow semantics specifies that when the appropriate event activates the chart (i.e., when clk triggers the chart) all these counters

Table I. Mapping UPPAAL conditions over clocks into Stateflow

UPPAAL condition ($x \in C, n \in \mathbf{N}_0$)	Stateflow condition – x is replaced with $u^{\mathcal{S}}(x)$
$x \leq n$	$(temporalCount(clk) \leq n - n_x)$
$x < n$	$before(n - n_x, clk)$
$x = n$	$(temporalCount(clk) == n - n_x)$
$x \geq n$	$after(n - n_x, clk)$
$x - y \bowtie n$	$n_x - n_y \bowtie n$

are incremented at the **beginning** of the chart's execution - i.e., even before the first parallel state begins its execution. Consequently, when each of the parallel states is executed, the counter value is equal to the number of the event's appearances from the activation of its currently active substate. For each parent state P_k this values is $temporalCount(clk)$, and thus we denote this value by tC^{P_k}. In addition, we define $tC^x = tC^{P_k}$ if x is a local clock defined in the automaton P_k.

Unlike in UPPAAL where clocks might not be reset to zero during a transition, for a parallel state in the Stateflow chart the aforementioned counter is always reset when a transition occurs (when the associated substate is activated). Thus, while mapping edges from the automaton P_k, we explicitly model **each local** (from P_k) **clock** x by introducing the accounting variable n_x that maintains the clock value from the moment of the last state activation. This is done using $R_C(r)$ from (3), which is specified as

$$[R_C(r)](x) = \begin{cases} n_x = n_x + temporalCount(clk), & \text{if } x \notin r \\ n_x = r(x), & \text{if } x \in r \end{cases} \tag{4}$$

Our goal is that at integer time points UPPAAL valuation u of the clock x (i.e., $u(x)$) is equal to the value $u^{\mathcal{S}}(x)$ defined as $u^{\mathcal{S}}(x) = n_x + tC^x$ (we will show this in the next section). Note that a single counter value is used for all local clocks defined within the same automaton (i.e., $tC^x = tC^y$ if x, y are local clocks defined in automaton P_k).

The transformation of the basic clock conditions presented in Table I employs event-based temporal logic operators while taking into account the values of the accounting variables for all used clocks. In the mapping each clock x is replaced with the value $u^{\mathcal{S}}(x)$. In addition, we used a relationship between Stateflow temporal logic operators from (2) to simplify the notation. For example, the condition $x < n$ for a local clock x is mapped into $n_x + temporalCount(clk) < n$, which is equivalent to $before(n - n_x, clk)$ (since $before(n - n_x, clk) = 1 \Leftrightarrow temporalCount(clk) < n - n_x$).

Finally, as specified in Def. 2.2, the requirement that the new clock valuation satisfies the invariant at the (new) location l_{k_j} is equivalent to the condition that both the non-reset and the reset clock values satisfy the clock invariants at location l_{k_j}. Hence, if $I(l_{k_j}) = h_1 \wedge ... \wedge h_k$, then $G_C(r, I(l_{k_j})) = G_C(r, h_1)) \wedge ... \wedge G_C(r, h_k))$ where

$$G_C(r, x \bowtie n) = \begin{cases} r(x) \bowtie n, & \text{if } x \in r \\ u^{\mathcal{S}}(x) \bowtie n & \text{if } x \notin r \end{cases}, \quad G_C(r, x - y \bowtie n) = \begin{cases} r(x) - r(y) \bowtie n, & \text{if } x, y \in r \\ r(x) - u^{\mathcal{S}}(y) \bowtie n, & \text{if } x \in r, y \notin r \\ u^{\mathcal{S}}(x) - r(y) \bowtie n, & \text{if } x \notin r, y \in r \\ u^{\mathcal{S}}(x) - u^{\mathcal{S}}(y) \bowtie n & \text{if } x, y \notin r \end{cases} \tag{5}$$

The expression for $G_C(r, I(l_{k_j}))$ can be significantly simplified. If $r(x)$ resets the clock x to a constant (which is the prevailing case in UPPAAL), conditions from (5) can be evaluated during the translation and can be replaced with fixed terms ($false$ or $true$).

5.3. Obtaining an MPA Execution of the Chart

The execution semantics of Stateflow ensures that in each of the parent states transitions from the active state will be evaluated at least once during a *clk* execution. However, to obtain an MPA run of the model, after a transition occurs it is necessary that in each parent state transitions from the active state are reevaluated. We guarantee this by reactivating the chart if at least one transition has occurred. Thus, in

Table II. Mapping UPPAAL edges from automaton P_k into Stateflow transitions

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

the chart, UPP2SF introduces the parallel state Eng (Figure 3), which is executed last among the parent (and clock) states. Furthermore, additional chart event tt and flag act are defined, and as a part of each transition, by adding $act = 1;$ to $R_S(r)$ from (3), act is set to 1. Finally, Eng contains a single substate and it broadcasts the event tt to the chart if act has been set to 1, using the lowest priority self-transition of the form

$$[act == 1]\{act = 0; send(tt)\} \tag{6}$$

5.4. 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 $ID(c)$. To translate edge $l_i \xrightarrow{g,c!,r} l_j$ from automaton P_k, UPP2SF uses a centralized approach where the Eng state broadcasts events and controls execution of the chart by using additional variable $sent$ that can have the following values (here, $ExO(P_k) > 0$ is the execution order of the parent state P_k)

$$sent = \begin{cases} ID(c), & \text{event } c \text{ is scheduled for broadcast} \\ 0, & \text{no event scheduled for broadcast} \\ -ExO(P_k), & \text{an event is broadcast, only receiving edges are enabled} \end{cases} \tag{7}$$

Note that $ID(c) > 0$, and $ExO(P_i) \neq ExO(P_j)$ if $P_i \neq P_j$.

Table II shows the mapping of UPPAAL edges into Stateflow. Action $c!$ is mapped into $sent = ID(C)$ assignment, thus disabling all 'non-receiving' transitions due to their condition $(sent == 0)$. Similarly, condition $(sent \sim= -ExO(P_k))$ disables all 'receiving' transitions in the parent state P_k, ensuring that the parent state does not synchronize with itself. Finally, the Eng state is used to broadcast events by adding for each event c the following self-transition in the state:

$$[(sent == ID(c))]\{sent = -ExO(P_k); send(c);\} \tag{8}$$

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

$$[(sent < 0)]\{sent = 0; send(tt);\} \tag{9}$$

Transitions (8), (9) have precedence (i.e., lower execution order) over the transition (6).

Remark 5.2. In general, more than one UPPAAL automaton could transmit over a shared broadcast channel. In this case, Eng state would not always be able to determine the parent state P_k that has initiated the event broadcast. Thus, variable $ExOP$ would have to be defined along with additional reset action $ExOP = ExO(P_k)$ in transitions with $sent = ID(c);$ (from Table II). Also, transition (8) would take the form $[(sent == ID(c))]\{sent = -ExOP; send(c);\}$. However, since this case does not occur in most UPPAAL models, due to the space limitation we present the simpler formulation.

5.5. Translating Urgent and Committed States

UPP2SF also preserves semantics of urgent and committed states, and urgent channels. By extracting MPA runs of the UPPAAL model we ensure that no time passes in the states from which there exists an enabled transitions. Thus, as a byproduct, semantics of urgent channels and locations are preserved. On the other hand, 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 $comm = comm + 1;$ is added to the reset operations (i.e., $R_S(r)$ from (3)). Similarly, for all outgoing transitions from a committed state $comm = comm - 1;$ 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 non-committed state. Note that setting act to 1 (as specified in (6), for all transitions in parent states) reactivates the chart to ensure that all transitions are reevaluated, including the ones that have been disabled due to $(comm == 0)$ condition.

5.6. Stateflow Chart Optimization

Stateflow charts obtained using the described set of rules can usually be significantly simplified. For example, clock guards and invariants specify fixed left-closed intervals if in conditions from Table I n denotes a constant. These intervals can be expressed in Stateflow with maximum two terms from Table I (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 updates to an accounting variable n_x from transitions incoming to a state, if on all paths from the state there exist resets of the clock x before the clock is used in a transition guard or invariant. Similarly, due to (9) 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, and no shared variable has been updated on the incoming transition. For example, in the buffer from Figure 5(f), 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 $l0$. Finally, transitions between two states with the same conditions and transition actions can be combined into a single transition.

6. CORRECTNESS OF THE TRANSLATION PROCEDURE

In this section, we show that the set of rules specified in the previous section preserves the UPPAAL semantics. Specifically, we show that the execution of the obtained Stateflow chart presents one of MPA runs of the initial UPPAAL model. However, since the Stateflow semantics is informally defined, formally proving correctness of the translation procedure is not possible. For a subset of Simulink features, there exist some attempts to derive formal semantics (e.g., [Hamon and Rushby 2007; Hamon 2005]), which have been validated by testing on many examples. We follow a similar approach in this work. We start by formulating basic assumptions on the semantics of the Stateflow charts obtained by UPP2SF – i.e., with the structure shown in Figure 3 and which utilize only a small subset of Stateflow functionalities.

Consider a deadlock-free UPPAAL model with automata $P_1, ..., P_n$, where each automaton has at least one location, and the extracted two-level Stateflow chart as in Figure 3, with parent states $P_1, ..., P_n$ and the parallel state Eng. Since none of the chart's parallel states has transitions[5] the following proposition holds.

PROPOSITION 6.1. *All parallel states in the chart will always be active.*

[5]Parallel states in Stateflow do not typically use transitions [sta 2012; Hamon and Rushby 2007].

The translation rules specify that all transitions in parent states do not have $condition\_action$ (from (1)), and no event broadcasting is specified in $transition\_actions$. Thus, when a transition $P_k.l \rightarrow P_k.l'$ occurs in a parent state P_k, substate l' will be directly activated (i.e., activity will be directly passed from l to l').[6] In addition, transitions in the Eng state do not have $transition\_actions$ and they broadcast events as part of $condition\_actions$.[7] That means that the state $Eng.l0$ will never be deactivated.

PROPOSITION 6.2. *Each parallel state in the chart always has an active substate.*

In the general case, event broadcasting to the whole chart introduces recursive behavior (we will also see this later, in Section 9.1, during the analysis of the pacemaker code – Listing 5, Figure 6). These recursions are in general very difficult to control and analyze, and [Hamon and Rushby 2007] restricted the use of events to the definition of sequencing behaviors. In UPP2SF-derived charts only Eng can broadcast events, on self-transitions from the (only) substate $Eng.l0$. Since Eng state is executed last within each chart activation, and within its execution only a single transition may occur, local event broadcasts can **only occur at the end of chart activations**. Therefore, although event broadcasts from the self-transitions in Eng state introduce recursive behavior to the chart, we can consider these recursive runs as series of sequential chart activations with different active events – i.e., we can disregard the recursive behavior.

Consequently, we can describe the state of the chart as $\theta^{\mathcal{S}} = (\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}})$, where $\bar{l}^{\mathcal{S}}$ is the vector of size n containing active substates for each of the n parent states.[8] In addition, $w^{\mathcal{S}} = (u^{\mathcal{S}}, v^{\mathcal{S}})$ where $u^{\mathcal{S}}$ is defined in (5.2.1) and $v^{\mathcal{S}}$ denotes the Stateflow variables mapped from UPPAAL variables – i.e., $v^{\mathcal{S}}(i)$ is the value of the Stateflow variable i. Finally, $f^{\mathcal{S}} = (act, sent, comm, AE, k)$ denotes the values of all control flags introduced by UPP2SF. AE ($AE \in \{clk, tt, \phi, SCE\} \cup \{c | c \in K\}$) is the currently active event being processed by the chart,[9] while k, $k \in \{0, 1, ..., n, n+1\}$, is the state index of the currently executed parallel state.[10] We denote by $\bar{l}^{\mathcal{S}}[l_{k_j}/l_{k_i}]$ a vector where the active substate of the parent state P_k in vector $\bar{l}^{\mathcal{S}}$ has changed from l_{k_i} to l_{k_j}, and use a similar notation for $w^{\mathcal{S}}$ and $f^{\mathcal{S}}$ updates – e.g., $f^{\mathcal{S}}[k = k+1]$ denotes $f^{\mathcal{S}}$ where only k is increased by 1. For UPPAAL models we use the notation from Sec. 2, and for $w = (u, v)$ we write $w = w^{\mathcal{S}}$ (and $u = u^{\mathcal{S}}, v = v^{\mathcal{S}}$) if $\forall x \in C, i \in V$, $u(x) = u^{\mathcal{S}}(x)$ and $v(i) = v^{\mathcal{S}}(i)$.

We can now formalize the behavior (i.e., semantics) of extracted Stateflow charts. From translation rules, the initial vector of active substates $\bar{l}_0^{\mathcal{S}}$ is equal to the initial location vector in UPPAAL (i.e., $\bar{l}_0^{\mathcal{S}} = \bar{l}_0$), and $w_0^{\mathcal{S}} = w_0$ (w_0 is the initial UPPAAL valuation), $f_0^{\mathcal{S}} = (0, 0, 0, SCE, 1)$ ($AE = SCE$, as charts are executed during initialization).

Definition 6.3. A *transition relation* for a UPP2SF-derived chart is defined as:

(1) $(\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}) \rightarrow (\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}[u^{\mathcal{S}} = u^{\mathcal{S}} + 1], f^{\mathcal{S}}[k = k+1])$ if $k = 0$ and $AE = clk$,
(2) $(\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}) \rightarrow (\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}[AE = clk])$ if $k = 0$ and $AE = \phi$,
(3) $(\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}) \rightarrow (\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}[k = k+1])$ if $k = 0$ and $AE \notin \{clk, \phi\}$,

[6]Broadcasting events in transition actions would result in deactivation of the substate l and event broadcasting before the substate l' is activated. Thus, during the event's processing (including processing events sent during the event's processing) the parent state P_k would not have active substates and would be effectively removed from the execution. On the other hand, broadcasting events in $condition\_actions$ would usually result in an infinite behavior (since in most cases the $condition$ would still be satisfied) [sta 2012].
[7]Note that in this case infinite cycle behavior does not occur, since the data values enabling a transition guard are changed before broadcasts – this disables the transition in the next activation.
[8]Note that since Eng state has a single substate that is always active, we do not specify it in the vector $\bar{l}^{\mathcal{S}}$.
[9]Here, ϕ denotes the case when no event is active - when the chart is sleeping, between consecutive *clk* executions, while SCE is the Simulink Call Event, an intrinsic way for Simulink to activate a Stateflow chart.
[10]In general, f should also contain a $transition\_index$ denoting the transition (from the active substate of the parallel state k) which is being evaluated. However, to simplify our notation we have omitted this term.

(4) $(\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}) \rightarrow (\bar{l}_1^{\mathcal{S}}, w_1^{\mathcal{S}}, f_1^{\mathcal{S}})$ if $k \in \{1, ..., n\}$, where

$$(\bar{l}_1^{\mathcal{S}}, w_1^{\mathcal{S}}, f_1^{\mathcal{S}}) = \begin{cases} (\bar{l}^{\mathcal{S}}[l_{k_{i_0}}/l_k], R_{C,V}^{i_0}[w^{\mathcal{S}}], R_S^{i_0}[f^{\mathcal{S}}][k = k+1]), & \exists i, (G_s^i \wedge G_{C,V}^i) = True, \text{and} \\ & (e^i \text{ not specified or } AE = e^i) \\ & i_0 = \min i \\ (\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}[k = k+1]), & \text{otherwise} \end{cases}$$

and all transitions outgoing from the active substate l_k (in P_k) are represented as $e^i[G_s^i \wedge G_{C,V}^i]/\{R_{C,V}^i; R_S^i;\}$ (i is the transition index, and for example, $R_{C,V}^{i_0}[w^{\mathcal{S}}]$ denotes the value of $w^{\mathcal{S}}$ after reset operations specified in $R_{C,V}^{i_0}$ are performed on $w^{\mathcal{S}}$);

(5) $(\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f^{\mathcal{S}}) \rightarrow (\bar{l}^{\mathcal{S}}, w^{\mathcal{S}}, f_1^{\mathcal{S}})$ if $k = n+1$, where

$$f_1^{\mathcal{S}} = \begin{cases} R_S^{i_0}[f^{\mathcal{S}}][k = 0, AE = e^i], \ \exists i, G_s^i = True, & \text{where } i_0 = \min i \\ f^{\mathcal{S}}[k = 0, AE = \phi]), & \text{otherwise} \end{cases}$$

and all self-transitions in Eng state are described as $[G_s^i]\{R_S^i; send(e^i);\}$.

We define the chart's execution trace $\mathcal{R}^{\mathcal{S}}$ as a sequence of consecutive chart states $\theta_0^{\mathcal{S}} \rightarrow \theta_1^{\mathcal{S}} \ldots \rightarrow \theta_t^{\mathcal{S}} \rightarrow \ldots$. We also denote by $\theta_{t_0}^{\mathcal{S}} \xrightarrow{*} \theta_{t_i}^{\mathcal{S}}$ (referred to as an *SF-transition*) a sequence $\theta_{t_0}^{\mathcal{S}} \rightarrow \theta_{t_1}^{\mathcal{S}} \ldots \rightarrow \theta_{t_i}^{\mathcal{S}}$ of the execution such that one of the following holds:

— the sequence does not contain any transitions mapped from UPPAAL edges, and $k = 1, AE = clk$ in $\theta_{t_i}^{\mathcal{S}}$; in this case we denote the sequence by $\theta_{t_0}^{\mathcal{S}} \xrightarrow{*(+d)} \theta_{t_i}^{\mathcal{S}}$, since (as none of the transitions in parent states has occurred during the sequence) for some $d \in \mathbb{N}$ (when $i > 0$), for all $x, y \in C, u_{t_i}^{\mathcal{S}}(x) - u_{t_0}^{\mathcal{S}}(x) = u_{t_i}^{\mathcal{S}}(y) - u_{t_0}^{\mathcal{S}}(y) = d$.

— $k \neq 0$ or $AE \neq clk$ in all $\theta_{t_j}^{\mathcal{S}}, j \leq i-1$; also, $\theta_{t_i-1}^{\mathcal{S}} \rightarrow \theta_{t_i}^{\mathcal{S}}$ is a transition mapped from some UPPAAL edge $l_f \xrightarrow{g,\tau,r} l_j$, and it is the only transition in the sequence mapped from any UPPAAL edge; in this case we also use the notation $\theta_{t_0}^{\mathcal{S}} \xrightarrow{*\tau} \theta_{t_i}^{\mathcal{S}}$,

— $k \neq 0$ or $AE \neq clk$ in all $\theta_{t_j}^{\mathcal{S}}, j \leq i-1$, and the sequence contains exactly one transition mapped from some UPPAAL edge $l_f \xrightarrow{g,b!,r} l_j$, which is the first transition in the sequence that is mapped from any UPPAAL edge; followed by a number of transitions (i.e., 0, 1 or more) mapped from UPPAAL edges of the form $l_p \xrightarrow{g,b?,r} l_q$ (i.e., receiving over channel b), where $\theta_{t_i-1}^{\mathcal{S}} \rightarrow \theta_{t_i}^{\mathcal{S}}$ is the last of them; in addition, the next transition in the execution trace which is mapped from an UPPAAL edge is not mapped from an edge $l_n \xrightarrow{g,b?,r} l_m$; here, we also use the notation $\theta_{t_0}^{\mathcal{S}} \xrightarrow{*b!?} \theta_{t_i}^{\mathcal{S}}$.

From the above definition, the following lemma follows directly.

LEMMA 6.4. *If* $\theta_{t_0}^{\mathcal{S}} \xrightarrow{*\tau} \theta_{t_i}^{\mathcal{S}}$ *then for all* $n \in \{t_0, ..., t_i - 1\}$, $\bar{l}_n^{\mathcal{S}} = \bar{l}_{t_0}^{\mathcal{S}}$ *and* $w_n^{\mathcal{S}} = w_{t_0}^{\mathcal{S}}$.

PROPOSITION 6.5. *Assume that* $\theta_{t_0}^{\mathcal{S}} \xrightarrow{*\tau} \theta_{t_i}^{\mathcal{S}}$, *and* $\bar{l}_{t_0}^{\mathcal{S}} = \bar{l}$ *and* $w_{t_0}^{\mathcal{S}} = w$. *If* $\theta_{t_i-1}^{\mathcal{S}} \rightarrow \theta_{t_i}^{\mathcal{S}}$ *is a transition mapped from UPPAAL edge* $l_{k_f} \xrightarrow{g,\tau,r} l_{k_j}$ *in automaton* P_k *(k is the state index from f), then* $(\bar{l}, w) \rightarrow (\bar{l}[l_{k_j}/l_{k_f}], r(w))$ *in UPPAAL, and* $r(w) = w_{t_i}^{\mathcal{S}}, \bar{l}[l_{k_j}/l_{k_f}] = \bar{l}_{t_i}^{\mathcal{S}}$.

PROOF. From Lemma 6.4, $\bar{l}_{t_i-1}^{\mathcal{S}} = \bar{l}_{t_0}^{\mathcal{S}} = \bar{l}$ and $w_{t_i-1}^{\mathcal{S}} = w_{t_0}^{\mathcal{S}} = w$. Since $\theta_{t_i-1}^{\mathcal{S}} \rightarrow \theta_{t_i}^{\mathcal{S}}$ is a transition mapped from $l_{k_f} \xrightarrow{g,\tau,r} l_{k_j}$, it follows that $\bar{l}[l_{k_j}/l_{k_f}] = \bar{l}_{t_i}^{\mathcal{S}}$, and from (3) $G_C(I(l_{k_f}) \wedge g) \wedge G_V(g) \wedge G_C(r, I(l_{k_j}))$ is satisfied. $G_V(g)$ presents the identical conditions over data variables as in the UPPAAL guard g, and since $v_{t_i-1}^{\mathcal{S}}$ satisfies $G_V(g)$, then v satisfies data conditions in g. Similarly, since $u_{t_i-1}^{\mathcal{S}}$ satisfies $G_C(I(l_{k_f}) \wedge g)$ from Table I, then from $u_{t_i-1}^{\mathcal{S}} = u$ we have that u satisfies the guard and invariant at l_{k_f} – i.e., $u \in g$

and $u \in I(l_{k_f})$. Finally, $u^{\mathcal{S}}_{t_i-1}$ satisfies $G_C(r, I(l_{k_j}))$ defined in (5), and using the same reasoning we have that $r(u) \in I(I(l_{k_j}))$, implying $w \in g$ and $r(w) \in I(I(l_{k_j}))$.

In addition, $w^{\mathcal{S}}_{t_i} = (u^{\mathcal{S}}_{t_i}, v^{\mathcal{S}}_{t_i})$, where $v^{\mathcal{S}}_{t_i} = R_V(r)[v^{\mathcal{S}}_{t_i-1}]$ and $u^{\mathcal{S}}_{t_i} = R_C(r)[u^{\mathcal{S}}_{t_i-1}]$. Since $R_V(r)$ specifies the identical data expressions as r, $v^{\mathcal{S}}_{t_i} = r(v^{\mathcal{S}}_{t_i-1}) = r(v)$. On the other hand, when a transition occurs $temporalCount(clk)$ is reset. Thus, for all $x \in C$, $u^{\mathcal{S}}_{t_i}(x) = n_x$, and if $x \in r$ from (4) $u^{\mathcal{S}}_{t_i}(x) = r(x)$; otherwise $n_x = u^{\mathcal{S}}_{t_i-1}(x)$, meaning that $u^{\mathcal{S}}_{t_i}(x) = u^{\mathcal{S}}_{t_i-1}(x)$. Consequently, $u^{\mathcal{S}}_{t_i} = r(u)$, and $w^{\mathcal{S}}_{t_i} = r(w)$, which concludes the proof. □

The following results can be proven using similar approaches as in the above proof.

PROPOSITION 6.6. *Assume that* $\theta^{\mathcal{S}}_{t_0} \xrightarrow{*b!?} \theta^{\mathcal{S}}_{t_i}$*, for a broadcast channel* b*, and* $\bar{l}^{\mathcal{S}}_{t_0} = \bar{l}$ *and* $w^{\mathcal{S}}_{t_0} = w$*. If the sequence contains transitions mapped from UPPAAL edges* $l_j \xrightarrow{g_j, b!, r_j} l'_j$ *in automaton* P_j*, and* $l_{j_i} \xrightarrow{g_{j_i}, b?, r_{j_i}} l'_{j_i}$ *in automata* P_{j_i} $(i = 0, ..., m)$*, then* $(\bar{l}, w) \rightarrow (\bar{l}[l'_j/l_j, l'_{j_0}/l_{j_0}, ..., l'_{j_m}/l_{j_m}], (r_j \cup_{i=0}^{m} r_{j_i})(w))$ *in UPPAAL. Furthermore,* $(r_j \cup_{i=0}^{m} r_{j_i})(w) = w^{\mathcal{S}}_{t_i}$ *and* $\bar{l}[l'_j/l_j, l'_{j_0}/l_{j_0}, ..., l'_{j_m}/l_{j_m}] = \bar{l}^{\mathcal{S}}_{t_i}$*.*

PROOF SKETCH. We first show that after the transition mapped from the edge $l_j \xrightarrow{g_j, b!, r_j} l'_j$ occurs it is not possible in the obtained chart to have a 'non-receiving' transition, or a 'receiving' transition conditioned with an event c, where $c \neq b$. We prove then that $j \neq j_i$ for $i = 0, ..., m$, and $j_p \neq j_q$ for $p, q = 0, ..., m$ and $p \neq q$ (i.e., an automaton cannot synchronize with itself, or synchronize twice with another automaton for a single broadcast). After showing these properties, using a similar approach as for Prop. 6.5 we show that the UPPAAL semantics for broadcast channels is preserved. □

PROPOSITION 6.7. *Consider a sequence* $\theta^{\mathcal{S}}_{t_0} \xrightarrow{*(+d)} \theta^{\mathcal{S}}_{t_i}$*, for some* $d \in \mathbb{N}$*, and lets assume* $\bar{l}^{\mathcal{S}}_{t_0} = \bar{l}$ *and* $w^{\mathcal{S}}_{t_0} = w$*. Then* $w^{\mathcal{S}}_{t_i} = w + d$*, and* $(\bar{l}, w) \rightarrow (\bar{l}, w + d)$ *is the only transition relation in the semantics of the UPPAAL model from* $(\bar{l}, w)$ *– i.e., there does not exist a transition* $(\bar{l}, w) \rightarrow (\bar{l}_1, w_1)$ *specified by either case (2) or (3) of Def. 2.2.*

Note that the chart's execution trace $\mathcal{R}^{\mathcal{S}}$ can be decomposed into a sequence of *SF-transitions*. Therefore, since the initial UPPAAL location vector $\bar{l}_0$ and valuation w_0 are equal to the initial Stateflow vector of active states and valuation $w^{\mathcal{S}}$, from the above three propositions and Theorem 3.2 we have that the sequence of *SF-transitions* for the UPP2SF-derived chart corresponds to an MPA run of the initial UPPAAL model. Furthermore, it is worth noting that these proofs can be easily extended to show that the semantics of committed locations is also preserved by the translation rules (semantics of urgent channels and locations are guaranteed by the *MPA-runs* requirement).

In the rest of the paper, we will demonstrate the use of the UPP2SF-based MDD framework on the pacemaker case study.

7. IMPLANTABLE CARDIAC PACEMAKERS

The primary function of an implantable pacemaker is to maintain an adequate heart rate, and ensure safe and efficient cardiac output. Pacemakers usually have two leads placed in the atrium and ventricle, capable of both sensing electrical activity in the heart and pacing the heart. To illustrate the UPP2SF-enabled MDD framework, we developed the most commonly used pacemaker mode, the DDD mode that paces both the atrium and ventricle, senses both, and employs the dual tracked response that synchronizes the chambers. We start by presenting DDD mode requirements from [Boston Scientific 2007], followed by the pacemaker modeling and verification in UPPAAL (UPPAAL modeling of more complex pacemaker modes can be found in [Jiang et al. 2012]).

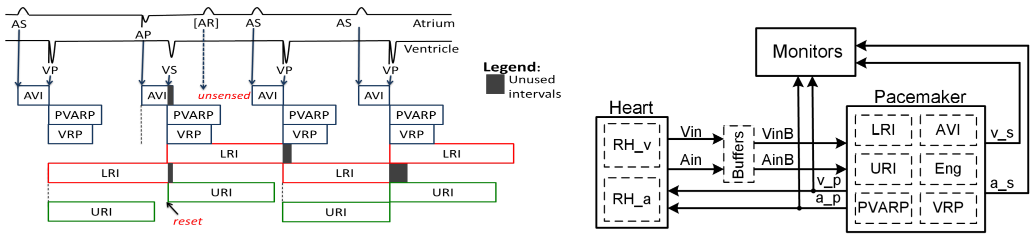


Fig. 4. (a) Pacemaker timing cycles [Jiang et al. 2012]; (b) Structure of the pacemaker model in UPPAAL, including the interaction between the pacemaker and heart, and the monitors used for verification.

7.1. Pacemaker Specification

We use the term *ventricular event* to specify either a pace in the ventricle (denoted VP) or a sense of electrical pulse (i.e., ventricular sense - VS). Similarly, we use *atrial event* to denote either atrial sense (AS) or pace (AP). However, not every intrinsic electrical activity will be registered as a sense in a chamber because some of the intrinsic pulses might fall within time intervals where sensing is disabled (refractory periods).

For heart therapy in DDD mode, the following requirements are defined:

— *Lowest Rate Interval (LRI)* defines the longest allowable VP interval if no VS is detected; the interval should start when a ventricular event occurs.
— *Upper Rate Interval (URI)* specifies the minimum time between a ventricular event and the next VP.
— *Atrio-Ventricular Interval (AVI)* is the time period from an atrial event to a VP.
— *Ventricular Refractory Period (VRP)* specifies the time interval following a ventricular event when intrinsic ventricular activity does not inhibit paces.
— *Post Ventricular Atrial Refractory Period (PVARP)* is the time interval following a ventricular event during which atrial cardiac event should not trigger a delayed VP.

Figure 4(a) illustrates the pacemaker's basic timing cycles. Beside imposing limits on the heart-rate and the synchronization requirement between the atrium and ventricle, the above specifications introduce refractory periods after ventricular events. These intervals are used to filter noise, ventricular signal reflections, and early events which could otherwise cause undesired pacemaker behavior. To achieve this, during a refractory interval sensing in the appropriate chamber (atrium for PVARP and ventricle for VRP) should be disabled. For example, in Figure 4(a) AR denotes intrinsic atrial activity that is not taken into account since it occurs within the PVARP interval.

To be able to perform system verification we used the above (informal) system requirements to derive a set of formal specifications. Due to space constraint, the description of the formal pacemaker specifications can be found in online appendix A.

7.2. Pacemaker Modeling and Verification in UPPAAL

To obtain a pacemaker model in UPPAAL, we modeled each of the (informal) requirements from Section 7.1 as a separate timed-automaton. Furthermore, as described in the introduction, to enable closed-loop system verification in UPPAAL it is necessary to provide models of the environment (i.e., the heart), along with the models of the interfaces (i.e., interaction) between the environment and controller (as shown in Figure 4(b)). Thus, the UPPAAL pacemaker model contains the following software components, shown in Figure 5, where each automaton only uses its own local clocks.

1. LRI automaton (Figure 5(a)) models the LRI requirement that keeps the heart rate above a minimum value by delivering atrial pace events (AP). The LRI timer is

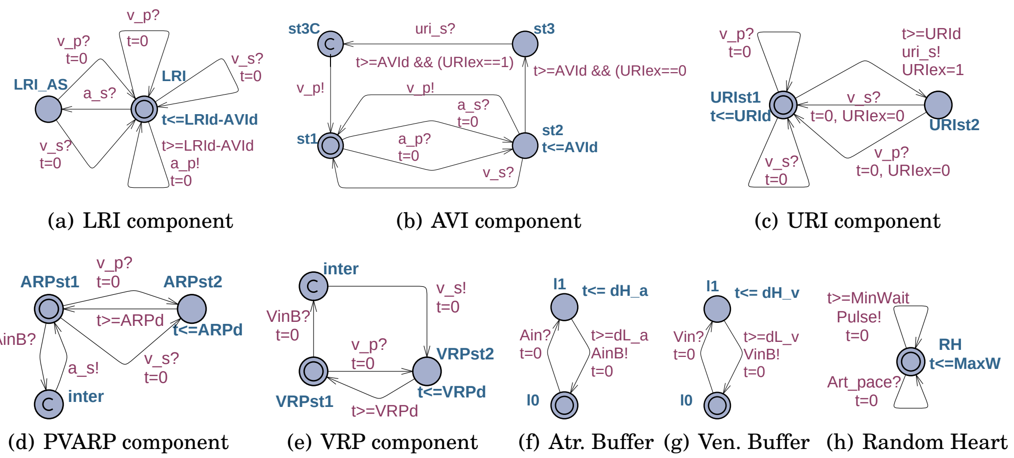


Fig. 5. DDD pacemaker model in UPPAAL - each automaton uses its local clock; two Random Heart templates were instantiated for Atrium (Pulse:=Ain, Art_pace:=a_p) and Ventricle (Pulse:=Vin, Art_pace:=v_p).

reset after a ventricular event (VP or VS). Also, if no atrial event has been sensed (AS) before the timer runs out, the pacing event will be delivered from the atrial lead (AP).

2. AVI automaton from Figure 5(b) models the AVI requirement, by mimicking 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. URI automaton in Figure 5(c) limits the ventricular pacing rate by enforcing a lower bound on the time between consecutive ventricle events.

4. PVARP and VRP automata are used to model the refractory periods. The PVARP and VRP automata generate atrial (AS) and ventricular (VS) sensing events from the buffered atrial and ventricular inputs (*AinB* and *VinB*, respectively). The PVARP automaton (Figure 5(d)) models the blocking interval after each ventricular event (VP or VS) where the atrial sensing (AS) cannot occur. The VRP automaton (Figure 5(e)) models the blocking interval for ventricular events. The interval follows 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.

6. Random Heart (RH) automata (Figure 5(h)) model the heart as two (for both chambers) uncorrelated random pulse generators with a single constraint that in each chamber the times between two consecutive events are within the predefined interval.

The descried pacemaker model was used to verify the formal pacemaker specifications in UPPAAL. We present the verification approach in online appendix B.

8. PACEMAKER STATEFLOW DESIGN

From the model shown in Figure 5, using the UPP2SF tool we obtained the pacemaker Stateflow chart presented in Figure 6. For closed-loop verification in UPPAAL we modeled both the heart and pacemaker, and therefore the obtained chart contains both models of the controller (i.e., pacemaker) and environment (i.e., the heart). To be able to use the obtained Stateflow chart for both simulation and code generation it was necessary to decouple the pacemaker from the heart model.

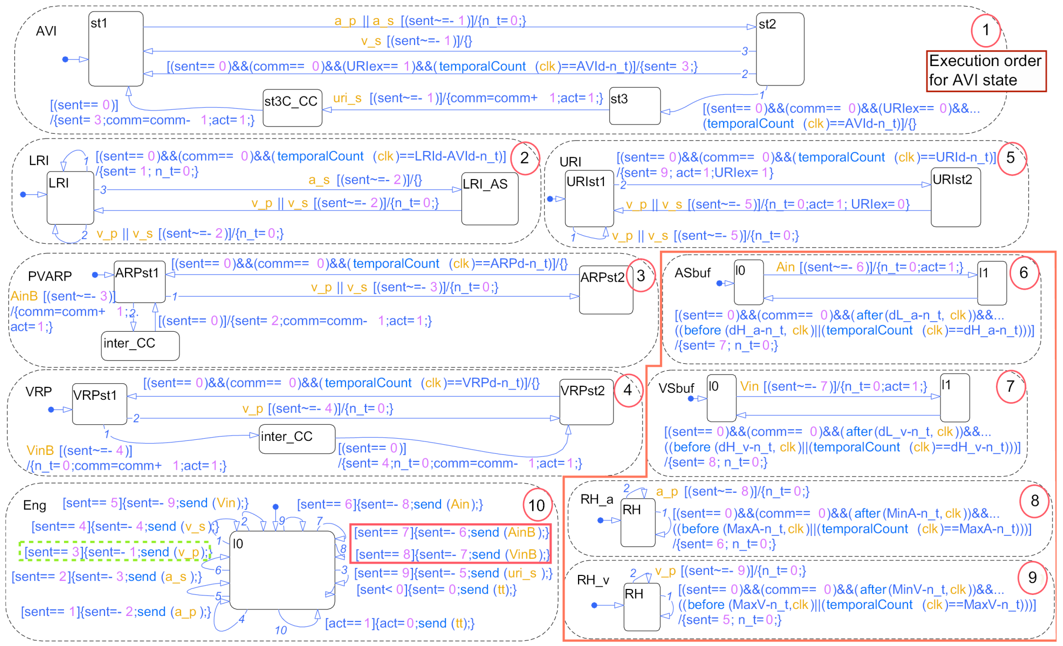


Fig. 6. Pacemaker Stateflow chart extracted using UPP2SF from the UPPAAL model in Figure 5; the heart and buffer models are highlighted.

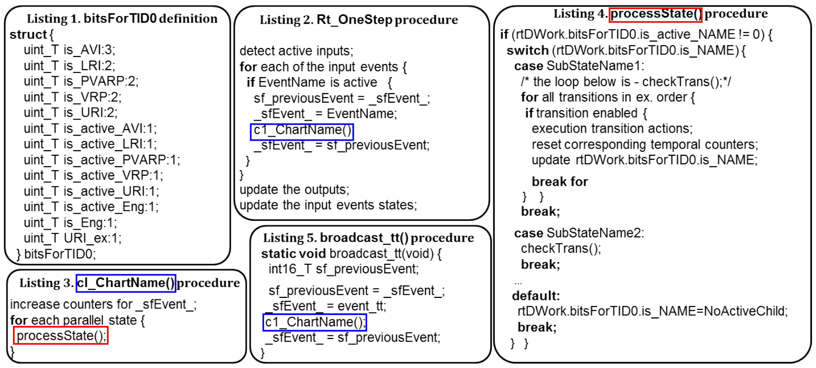
Note that the verified UPPAAL model also contains several monitors used to specify verification queries. Since none of these monitors uses shared variables, and they only interact with the rest of the model by receiving synchronization over broadcast channels, they do not affect behavior of the basic automata from Figure 5. Thus, to simplify Figure 6, we did not show the parallel states that were obtained from them.

8.1. Decoupling the Controller and Environment

Here we present the approach used to decouple the pacemaker and heart. The same approach can be used to decouple models of the environment and controllers in most commonly used scenarios where they only interact by broadcasting events.

Figure 4(b) shows the interaction between the pacemaker and the heart. Since the interaction is modeled using synchronization over broadcast channels, the pacemaker model can be easily extracted from the chart shown in Figure 6. 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 Figure 6).

Stateflow does not allow the use of output events to condition internal transitions. Hence, it is necessary to define additional output events from the chart, and in our case for local events a_p and v_p two output events (AP and VP) were defined. These events are broadcast on 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 9), 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 Simulink chart simulations, but allows for the correct output generation from the synthesized code. For



Listing 1. bitsForTID0 definition

```
struct {
   uint_T is_AVI:3;
   uint_T is_LRI:2;
   uint_T is_PVARP:2;
   uint_T is_VRP:2;
   uint_T is_URI:2;
   uint_T is_active_AVI:1;
   uint_T is_active_LRI:1;
   uint_T is_active_PVARP:1;
   uint_T is_active_VRP:1;
   uint_T is_active_URI:1;
   uint_T is_active_Eng:1;
   uint_T is_Eng:1;
   uint_T URI_ex:1;
 } bitsForTID0;
```

Listing 2. Rt_OneStep procedure

```
detect active inputs;
for each of the input events {
 if EventName is active  {
  sf_previousEvent = _sfEvent_;
  _sfEvent_ = EventName;
  c1_ChartName()
  _sfEvent_ = sf_previousEvent;
 }
}
update the outputs;
update the input events states;
```

Listing 3. cl_ChartName() procedure

```
increase counters for _sfEvent_;
for each parallel state {
 processState();
}
```

Listing 4. processState() procedure

```
if (rtDWork.bitsForTID0.is_active_NAME != 0) {
 switch (rtDWork.bitsForTID0.is_NAME) {
  case SubStateName1:
   /* the loop below is - checkTrans();*/
   for all transitions in ex. order {
    if transition enabled {
     execution transition actions;
     reset corresponding temporal counters;
     update rtDWork.bitsForTID0.is_NAME;

     break for
   }  }
   break;

  case SubStateName2:
   checkTrans();
   break;
  ...
 default:
  rtDWork.bitsForTID0.is_NAME=NoActiveChild;
  break;
 }  }
```

Listing 5. broadcast_tt() procedure

```
static void broadcast_tt(void) {
 int16_T sf_previousEvent;

 sf_previousEvent = _sfEvent_;
 _sfEvent_ = event_tt;
c1_ChartName();
 _sfEvent_ = sf_previousEvent;
}
```

Fig. 7. Structure of the pacemaker code obtained from the Stateflow chart shown in Figure 6.

example, the Eng transition highlighted with dotted green rectangle was modified to

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

Note that if the user specifies all components that are part of the controller, UPP2SF can automatically perform the above actions to decouple it from the environment.

It is interesting to compare the chart from Figure 6 with the manually designed Stateflow model of the DDD pacemaker [Jiang et al. 2010], which is slightly simpler as it does not use event broadcasting. Thus, each *clk execution* has a single chart activation, causing a violation of several of the pacemaker requirements from Sec. 7.1 (and appendix A). For example, the URI requirement is not satisfied because URI state is always scheduled after AVI state; since the chart is not reactivated within a *clk execution*, after URI period expires, AP will be generated late – in the next *clk activation*.

Since the UPP2SF mapping has been validated, we ensure that the chart's execution will be equivalent to one of the MPA runs of the initial UPPAAL model. However, the chart also contains the model the environment and has no inputs and outputs, and thus we performed validation of the pacemaker Stateflow chart after the decoupling, by extending the approach for testing real-time constraints by [Clarke and Lee 1995]. Due to space limitations, more details can be found in online appendix C.

9. PACEMAKER IMPLEMENTATION

We generated C code from the pacemaker Stateflow chart using the Simulink Real-Time Workshop Embedded Coder (RTWEC).[11,12] 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 input events should be created every 1ms, followed by the procedure's activation.[13] This makes it suitable for implementation on top of a real-time operating system (RTOS).

9.0.1. 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`, along with counter values used for temporal logic operators (i.e., a counter per parallel state). In addition, `rtDWork` contains a structure (List. 1, Figure 7) that for

[11]Since Matlab 2011b, RTWEC toolbox is referred to as Simulink Embedded Coder.
[12]Although we focus on a specific implementation, code with the same structure would be generated from all Stateflow charts obtained from UPPAAL models using UPP2SF (due to the derived charts' structure).
[13]In this case, the procedure's execution corresponds to the clk execution.

each parent state specifies if it is active, along with which of its substates is active. For example, for the state AVI variable `is_active_AVI` describes whether the state is active, while `is_AVI` specifies which of its exclusive substates is active.[14]

The structure of `rt_OneStep` is shown in List. 2, Figure 7. After detecting active input events, an execution of the chart procedure `c1_ChartName` is invoked for each active input event. 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 `rt_OneStep` procedure. This can cause a couple of problems. First, ordering of the generated output events can differ 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.[15] 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 execution procedure (List. 3, Figure 7) 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 Figure 7 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 prespecified 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, Figure 7, broadcasting an event associates the (current event) variable $\_sfEvent\_$ with the event, before it reactivates the chart (by calling `c1_ChartName()`).

9.1. Platform Implementation

The pacemaker code generated by the Simulink RTWEC was executed on nanoRK [nan 2013], a fixed-priority preemptive RTOS that runs on a variety of resource constrained platforms. We tested the implementation on the TI MSP-EXP430F5438 Experimenter Board interfaced with a signal generator that provides inputs for the pacemaker code (Figure 8). The compiled (without optimization) pacemaker procedure uses 2536 B for

[14]Note that since all parent states are decomposed into exclusive states, activity status for all of the substates within a parent state can be specified with a single variable. However, in the general case, if a state consists of a group of parallel states, RTWEC would define a new variable for each of the parallel states.

[15]These issues do not present a problem for the pacemaker design from Figure 6, 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 translation tool for all UPPAAL models.

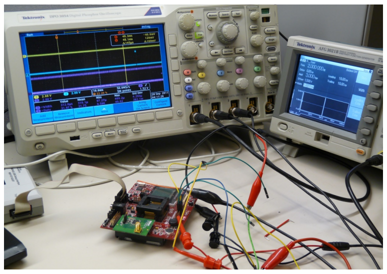


Fig. 8. Hardware setup with MSP430F5438 experimenters board.

code and additional 180 B for data. To interface the code with the environment, each of the inputs ($AinB$, $VinB$) 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 III shows measured execution times for the pacemaker tasks, for two different CPU frequencies. As expected, an increase in CPU frequency scales into a reduction in the task's execution time. Note that the measurements from Table III 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. More details about platform testing, including testing of the pacemaker formal requirements can be found in online appendix D.

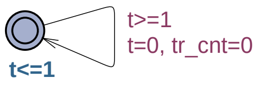
9.2. Decoupling the Controller and the Environment

In Sec. 8 and 9.1 we have described the method we used to decouple models of the pacemaker and the heart. The solution guarantees that the implemented code generates output events as soon as the corresponding local events are generated. 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 for 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 up to the task's period (i.e., clk interval, in our case 1ms) before the input signals are processed. To solve this issue, the extracted procedure could be activated as soon as an input appears. Beside problems with tasks scheduling, if the input signal could affect clock valuations in the initial UPPAAL model this would introduce a time measurement error in the code. For example, if Ain occurs 0.5ms before the next clk activation and the procedure is instantaneously activated as a result of the input, the clock in $ASbuf$ would be reset to zero. Thus, the next clk activation of the procedure would set t to 1, although only 0.5 ms have passed since the input.

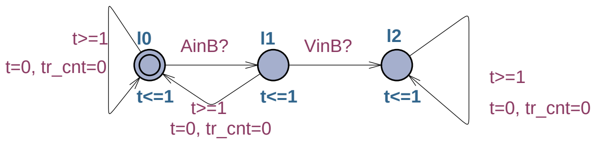
This problem occurs even if the code has been generated using Times, or any other tool, since the number of clocks used in models is usually greater than the number of

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

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



(a) Basic monitor



(b) Transition automaton with input ordering

Fig. 9. Transition monitors (`TrMonitor`) used for the worst-case execution time estimation.

timers that CPU provides. To avoid this type of errors, we opted to use the aforementioned approach where input events only set a flag to indicate the need to process input events in the following procedure activation. To take this into account in the initial UPPAAL model, we reverified the safety properties for the model where input buffers increase the upper bound on the introduced delay (i.e., dH_a, dH_v). The bounds are increased to incorporate the maximal input latency introduced by synchronous processing of the input events (in our case 1ms).

9.3. 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` procedure (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). This enables a WCET estimation at an early stage, during system modeling in UPPAAL.

Since the chart is reactivated with event broadcasts and some transitions, to determine the bound for N_i we extend the model with the following accounting features:

— Global variable tr_cnt and the automaton `TrMonitor` (Figure 9) that resets the variable at integer time points,
— In the controller part of the UPPAAL model, reset operation $tr\_cnt = tr\_cnt{+}1$ should be added to all edges with transmissions over a broadcast channel, or edges that would be translated into Stateflow transitions with $act = 1$ reset (i.e., the transition for which Eng state would reactivate the chart),
— Reset $tr\_cnt = tr\_cnt + 1$ to the edges with transmissions over broadcast channels that present inputs to the controller,
— Introduce UPPAAL temporal formula $A\Box\ tr\_cnt \le \tilde{N}_i$.

With the above changes the variable tr_cnt bounds the number of internal activations of the chart. Therefore, 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 Figure 5 we added the reset operation to 8 transitions. We proved that the formula holds for $\tilde{N}_i = 5$. Note that since the UPPAAL model contains a model of the environment (i.e., the heart), $\tilde{N}_i$ takes into account chart activations caused by inputs. On the other hand, when we considered open-loop execution of the pacemaker (without input events from the heart), using the pacemaker model Figure 5 without the model of the environment (i.e., RH automata) we proved that the formula holds for $\tilde{N}_i^{ol} = 1$. Thus, in this case at most two chart executions can occur within a single clk execution (i.e., task activation).

If these results are compared with the execution time measurements from Table III, we can notice that for the open-loop experiments, the ratio between the maximal and minimal execution time is less than 3. Similarly, for the experiments with the test generator, the ratio is less than 5. Since in our case the minimal execution time corresponds to a single chart execution during the task's activation (which in general might not be the case), we can infer that $N_i^{ol} = 1$ and $N_i = 3$. Therefore, our WCET analysis provided the exact bound for the open-loop scenario and a conservative bound for the closed-loop case.

The reason for this is that the transition monitor from Figure 9(a) does not take into consideration the order of the input events processing, and if both $AinB!$ and $VinB!$ occur at the same time instance, the UPPAAL model might synchronize over the channel $AinB$ first. On the other hand, the pacemaker model in Stateflow, and thus the obtained code, have a fixed input ordering, meaning that the inputs are always processed in the predefined order; in the pacemaker code $VinB$ is always processed before $AinB$ (and the clk event is processed last). To take this into account we used the monitor from Figure 9(b) and specified the proposition as $A\Box\ ((tr\_cnt \leq \tilde{N}_i)\ ||\ (TrMonitor.l2))$. This effectively disregards scenarios in which $AinB$ is processed before $VinB$ within a task execution. We proved that this formula holds for $\tilde{N}_i = 4$, thus improving the bound. Note that if the obtained code has more than two inputs (beside clk) it is necessary to specify all possible invalid combinations of the inputs' ordering, which might significantly increase the verification time.

10. CONCLUSION

We have described the design of the UPP2SF tool for automatic translation of UPPAAL models in Stateflow. We have shown that for a large class of UPPAAL models, UPP2SF preserves behavior of the initial UPPAAL model. Furthermore, we have presented an UPP2SF-enabled Model-Driven Development framework for safety-critical system design. By applying the UPP2SF model translation tool on the dual-chamber, implantable cardiac pacemaker case study, we have demonstrated the process starting from the formalization of the device specifications, followed by system modeling and verification in UPPAAL, to closed-loop system simulation in Simulink/Stateflow and testing of the physical implementation. We have also shown how the translation tool provides a way to estimate WCET during modeling and verification stage in UPPAAL, and facilitates development of modular code from UPPAAL timed-automata based models. The presented case-study fits into the scenarios where the system is controlled using a single, centralized, controller. We plan to investigate the use of the UPP2SF-based MDD framework for code synthesis for distributed applications.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to thank Anaheed Zaki for fruitful discussions during the preparation of this paper. This research was supported in part by NSF CNS-0720518, NSF CNS-1035715, NSF MRI-0923518, and NSF CAREER-1253842 grants. Preliminary version of this paper have appeared in [Pajic et al. 2012a].

REFERENCES

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

2012. Matlab R2012a Documentation → Stateflow. http://www.mathworks.com/help/toolbox/stateflow. (2012).

2013. nano-RK Sensor RTOS. http://nanork.org. (2013).

K. Altisen and S. Tripakis. 2005. Implementation of Timed Automata: An Issue of Semantics or Modeling? In *Formal Modeling and Analysis of Timed Systems*. Vol. 3829. 273–288.

R. Alur. 1999. Timed Automata. In *Computer Aided Verification*. Vol. 1633. 688–688.

R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. 1995. The algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138, 1 (1995), 3–34.

T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. 2004. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Formal Modeling and Analysis of Timed Systems*. Vol. 2791. 60–72.

A. Ayoub, A. Wahba, A. Salem, and M. Sheirah. 2010. Code Synthesis for Timed Automata: A Comparison Using Case Study. In *Abstract State Machines, Alloy, B and Z*. Lecture Notes in Computer Science, Vol. 5977. 403–403.

G. Behrmann, A. David, and K. Larsen. 2004. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*. Vol. 3185. 33–35.

J. Bengtsson and W. Yi. 2004. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*. Vol. 3098. 87–124.

Boston Scientific. 2007. PACEMAKER System Specification. (2007).

D. Clarke and I. Lee. 1995. Testing Real-Time Constraints in a Process Algebraic Setting. In *Proceedings of the International Conference on Software Engineering*. 51–60.

E. M. Clarke and E. A. Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Workshop on Logic of Programs (LNCS)*, Vol. 131. 52–71.

G. Hamon. 2005. A Denotational Semantics for Stateflow. In *EMSOFT'05: Proc. of the 5th ACM international conference on Embedded software*. 164–172.

G. Hamon and J. Rushby. 2007. An Operational Semantics for Stateflow. *International Journal on Software Tools for Technology Transfer* 9, 5 (2007), 447–456.

M. Hendriks. 2001. *Translating UPPAAL to Not Quite C*. Technical Report CSI-R0108. Comp. Sc. Institute.

Z. Jiang, M. Pajic, A. Connolly, S. Dixit, and R. Mangharam. 2010. Real-Time Heart Model for Implantable Cardiac Device Validation and Verification. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*. 239–248.

Z. Jiang, M. Pajic, and R. Mangharam. 2012. Cyber-Physical Modeling of Implantable Cardiac Medical Devices. *Proc. IEEE* 100, 1 (2012), 122–137.

Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam. 2012. 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*. 188–203.

B. G. Kim, A. Ayoub, P. Jones, O. Sokolsky Y. Zhang, R. Jetley, and I. Lee. 2011. Safety-Assured Development of the GPCA Infusion Pump Software. In *EMSOFT'11: ACM conf. on Embedded software*. 89–98.

K. G. Larsen, P. Pettersson, and W. Yi. 1997. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer* 1, 1 (1997), 134–152.

I. Lee, G. J. Pappas, R. Cleaveland, J. Hatcliff, B.H. Krogh, P. Lee, H. Rubin, and L. Sha. 2006. High-Confidence Medical Device Software and Systems. *IEEE Computer* 39, 4 (2006), 33–38.

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

M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. 2012a. From Verification to Implementation: A Model Translation Tool and a Pacemaker Case Study. In *18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 173–184.

M. Pajic, I. Lee, R. Mangharam, and O. Sokolsky. 2012b. *UPP2SF: Translating UPPAAL models to Simulink*. Technical Report. University of Pennsylvania.

M. Pajic, R. Mangharam, O. Sokolsky, D. Arney, J. Goldman, and I. Lee. 2012c. Model-Driven Safety Analysis of Closed-Loop Medical Systems. *IEEE Transactions on Industrial Informatics* 99 (2012), 13. DOI:http://dx.doi.org/10.1109/TII.2012.2226594

K. Sandler, L. Ohrstrom, L. Moy, and R. McVay. 2010. Killed by Code: Software Transparency in Implantable Medical Devices. *Software Freedom Law Center* (2010).

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

Max Schürenberg. 2012. Scalability Analysis of the Simulink Design Verifier on an Avionic System. (2012).

Online Appendix to: Safety-critical Medical Device Development using the UPP2SF Model Translation Tool

MIROSLAV PAJIC, University of Pennsylvania
ZHIHAO JIANG, University of Pennsylvania
INSUP LEE, University of Pennsylvania
OLEG SOKOLSKY, University of Pennsylvania
RAHUL MANGHARAM, University of Pennsylvania

A. FORMAL PACEMAKER SPECIFICATION

We classified the real-time constraints for the pacemaker into two categories: *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 $[t_1, t_2)$. From the system requirements it can be noted that pacemakers exert a highly repetitive behavior, where every requirement needs to be satisfied in each time interval between consecutive ventricular and/or atrial events. Therefore, to formalize the specifications for DDD pacemaker it is necessary to consider two time axes, t_v and t_a that measure the time since the last ventricular event (VP or VS) and the last atrial event (AP or AS), respectively. Now, we can define a set of formal pacemaker requirements (performance constraints are denoted by **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 at $t_v = LRI_d - AVI_d$.

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 $t_v \in (0, URI_{def})$;[16]
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 $t_a \in (0, AVI_d)$, a VP should not be generated at $t_a = AVI_d$.

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

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

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.

A.0.1. Pacemaker Specifications with Tolerances. The above specifications are referred to as the *ideal* pacemaker specification. However, each of the intervals is assigned with a certain level of tolerance [Boston Scientific 2007]. Thus, we define *realistic* pacemaker specifications, where each of the real-time constraints is modified to incorporate a tolerance. For example, we modify constraints **P1.1** and **B1.1** for pacing in the atrium to:
P1.1: AP cannot occur during at $t_v \in [0, LRI_d - AVI_d - \Delta_{ap})$;
B1.1: If AS does not occur within interval $t_v \in [0, LRI_d - AVI_d - \Delta_{ap})$, an 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 specification using the tolerances: Δ_{vp} - tolerance for ventricular pacing; Δ_{as} - tolerance for atrial sensing; Δ_{vs} - tolerance for ventricular sensing. A set of parameters values, along with their tolerances, is specified in [Boston Scientific 2007] for each of the aforementioned intervals.

B. PACEMAKER VERIFICATION IN UPPAAL

The first sanity check that was done for the model is to verify the absence of deadlocks in the model.[17] Furthermore, we verified the formal properties from Section A by designing a set of monitors to specify the corresponding UPPAAL queries. We express the queries in the subset of the Computational Tree Logic (CTL) [Clarke and Emerson 1981] used by UPPAAL. The main temporal operators of this logic that we use are $A\Box\phi$, which means that ϕ is satisfied in every state along every execution path from the current state, and $A\Diamond\phi$, meaning that ϕ is satisfied eventually along every path.

In this section we present the verification procedure for the representative properties, performance **P1.1** and behavioral **B2.1** specifications. A similar approach was used to verify the rest of the formal specifications. To verify these two properties we developed monitors $Event\_occur(Hevent))$, $Timer(Sense, Pulse)$ and URI_mon from Figure 10.[18] Automata $Event\_occur$ and URI_mon can be used to specify propositions that require information related to occurrences of certain events, while the clock t defined for the monitor $Timer$ measures the time from the last pulse or sense signal. For example, in automaton $Timer(a\_s, a\_p)$ the clock t has the value of the previously defined time t_a - the time since the last atrial activity.

To specify the property **P1.1** in UPPAAL we used monitors $ME\_ap = Event\_occur(a\_p))$ from Figure 10(a) and $MT\_v = Timer(v\_s, v\_p)$ from Figure 10(b). Furthermore, we specified the property as:

$$A\Box(ME\_ap.occur \Rightarrow (MT\_v.t \geq LRId - AVId)), \tag{10}$$

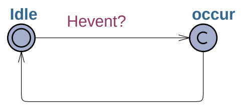
and verified it in UPPAAL. Similarly, for the property **B2.1** we used monitors $ME\_vp = Event\_occur(v\_p))$, $MT\_a = Timer(a\_s, a\_p)$, and URI_mon (Figure 10(c)). The verified query was formulated as:

$$A\Box(ME\_vp.occur \wedge not(URI\_mon.vped) \Rightarrow (MT\_a.t \leq AVId)). \tag{11}$$

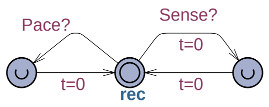
C. STATEFLOW MODEL VALIDATION

We performed validation of the pacemaker Stateflow chart after the decoupling by extending the approach for testing real-time constraints described in [Clarke and Lee

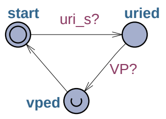
[17]It is worth noting that the initial UPPAAL model must be deadlock-free to guarantee that the UPP2SF will derive a Stateflow chart with execution that corresponds to an MPA run of the UPPAAL model.
[18]In the UPPAAL notation, input parameters for a component are specified between brackets.For example, $Event\_occur(a\_p)$ denotes the automaton $Event\_occur$ where $Hevent := a\_p$.



(a) $Event\_occur(Hevnt)$ monitor



(b) $Timer(Sense, Pace)$ monitor



(c) URI monitor

Fig. 10. Monitors used for verification of the pacemaker model in UPPAAL - each automaton uses its own local clock.

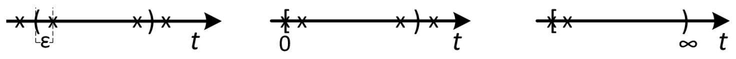


Fig. 11. Test points for behavioral real-time constraints.

1995]. The same approach was also used for the testing of the final physical implementation (Section D).

In [Clarke and Lee 1995], the authors proposed a testing procedure for behavioral and performance constraints (as the ones specified in Section A) if the following assumptions are satisfied:

(1) All output sequences of a system under test must be eventually distinguishable.
(2) All time bounds must be constant.
(3) There is no sharing of resources between concurrent threads.
(4) Specification intervals are implemented in software as continuous, linear domains.

In our case, all of the assumptions are inherently satisfied, and thus we adapted the method for the pacemaker testing. From the formal specifications we derived an appropriate set of tests to validate correctness of the obtained pacemaker design. For each performance constraint we used a test that validates whether the appropriate output has been generated within the required interval. On the other hand, testing behavioral constraints was more complex. For each interval boundary we generated two tests. For closed boundaries we applied inputs that were exactly at the boundary point and tests that were outside the interval, at the distance ϵ from the boundary point (see Figure 11). For open boundaries we generated inputs that were inside and outside the interval, at the distance ϵ from the boundary point.

To perform validation of the Stateflow chart and the physical implementation, we used the model parameters from Table IV. 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 $\epsilon = 0.5ms$ and $\epsilon = 1ms$ for simulations. All the '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, and outgoing transitions from the states would occur immediately after the states were activated. To illustrate a more complex behavior we also showed that, as in UPPAAL, if $st3$ state in AVI parent state was active when the transition $URIst1 \rightarrow URIst2$ occurred (causing broadcast

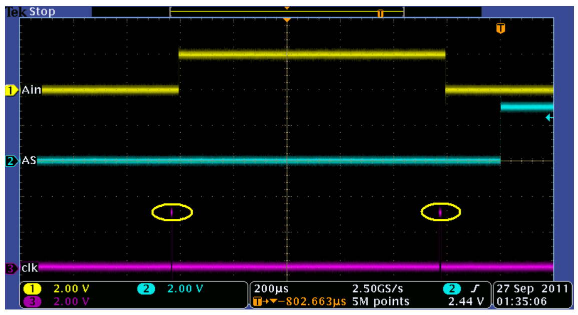


Fig. 12. A test screen shot for property **B4.2**; clk pulses are highlighted.

of uri_s event), then no time had elapsed in the $URIst2$ state, before the transition $URIst2 \rightarrow URIst1$ conditioned with v_p took place.

D. TESTING OF THE PHYSICAL IMPLEMENTATION

We validated the physical implementation using the procedure from Section C. 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 III). Similarly, since the values for all the predefined tolerances are $\pm 4ms$, for the second set of tests we used $4ms < \epsilon \leq 4.08ms$.

Table V presents testing results for the pacemaker implementation executed on the MSP430 Experimenter 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 V.

For example, consider the property **B4.2**. Figure 12 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

Table IV. Pacemakers parameters.

Parameter	Range	Value	Tolerance
LRI_d	343-1200 ms	1000ms	$\pm 4ms$
AVI_d	70-300 ms	150ms	$\pm 4ms$
URI_d	1000 ms	400ms	$\pm 4ms$
VRP_d	150-500 ms	150ms	$\pm 4ms$
ARP_d	150-500 ms	200ms	$\pm 4ms$

Table V. Results of the tests performed on the setup from Figure 8.

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	

is generated. This, along with the time (up to $1ms$) between Ain and the 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.