Modeling and Verification of Safety Critical Systems: A Case Study on Pacemaker

Luu Anh Tuan and Man ChunZheng
School of Computing
National University of Singapore
{*tuanluu, zmanchun*}*@comp.nus.edu.sg*

Quan Thanh Tho
Hochiminh City University of Technology
Vietnam
qttho@cse.hcmut.edu.vn

***Abstract*—The pacemaker challenge proposed by Software Quality Research Laboratory is looking for formal methods to produce precise and reliable systems. Safety critical systems like pacemaker need to guarantee important properties (like deadlock-free, safety, etc.), which concern human lives. Formal methods have been applied in designing safety critical systems with verified desirable properties. In this paper, we propose a formal model of pacemaker, modeling its behaviors and its communication with the external environment, using a real-time formalism. Critical properties, such as deadlock freeness and heart rate limits are then verified using the model checker PAT(Process Analysis Toolkit). This work yields a verified formal model of pacemaker systems, which can serve as specification for real pacemaker implementations.**

***Keywords*-Pacemaker; PAT; model checking; verification;**

I. Introduction

The failure of a safety critical system may result in loss of life, severe damage to equipment or environment [18]. Systems such as medical devices, aircraft flight control, weapons, and nuclear systems are always safety critical. Therefore, safety critical systems require high correctness to guarantee reliability and robustness. Flaws or bugs are always vital to breaking correctness of a system. Traditional software testing techniques are valid for finding bugs in implementations of systems. Testing always requires carefully designed test cases to cover all possible paths of execution, but it is hard to guarantee that the design of test cases is a complete cover of all possible executing paths. Usually, testing technicians are unable to tell whether all bugs existing in the system have been detected after testing.

Model checking, is a formal verification technique introduced to exhaustively explore the complete state space of a system for finding flaws or bugs. To apply model checking techniques, a system is first specified as a formal model, which is then verified against properties. In [28] the properties of safety critical systems such as dependability, safety and real-time constraint are considered, and formal specification is introduced as a valuable way to understand them. Formal method is considered as an important technique to improve correctness of safety critical systems, as presented in [14]. [7] discusses the standards of applying formal methods technique, while [6] examines the industrial uses of formal methods for the development of safety critical system, and discusses the application details of formal methods.

A pacemaker is an electronic device used to treat patients who have symptoms caused by abnormally slow heartbeats. The purpose of using a pacemaker is to maintain heartbeats so that adequate oxygen and nutrients can be delivered through the blood to the organs of the human body. A pacemaker is safety-critical, since its failure may cause severe damage to human body or even loss of human life. Software Quality Research Laboratory proposes the pacemaker challenge, which is looking for formal methods to produce correct and reliable systems. The verification grand challenge sets the stage for the program verification community to embark upon a collaborative effort to build verifiable programs. Boston Scientific has released into the public domain the system specification [1] for a pacemaker of a previous generation.

According to the informal requirements presented in [1], we establish a formal model (i.e. *RTS* model) of the pacemaker. *RTS*, is a process algebra based formalism, with timed extensions to *CSP*. Our home-grown model checker *PAT* has been enhanced for verifying timed refinements. The *RTS* model of pacemaker is then verified against safety properties and timed constraints. In this paper, we first explain the construction of the *RTS* model of pacemaker, as a parallel of timed processes, modeling pacemaker's components and its interaction with the environment (i.e. the implanted heart). It is vital that a pacemaker maintains a normal heart rate and prevents abnormal slow heart beatings. Such requirements are described as safety properties and timed constraints, and are defined as *LTL* (Linear Temporal Logic) formulae and timed refinements. Experiments are then carried out to verify the *RTS* model against those critical properties of pacemaker, in our home-built model checker *PAT*. We believe that our work will produce a reliable and correct pacemaker formal model which satisfies a set of critical properties.

Outline The rest of the paper is organized as follows. Section II discusses related works on formal models and verifications of pacemaker. Section III presents some background knowledge, including a summary of formal methods, an overview of *RTS* model and the informal description of pacemaker. Section IV details the *RTS* models of pacemaker and the approach to verify the models for critical properties. Results of experiments of verifying the *RTS* model are presented in Section V, and Section VI finally concludes the whole paper.

II. Related Work

Wiggelinkhuizen uses *mCRL2* [13] and *UPPAAL* [5] for formally modeling the pacemaker from *Vitatron*. Several formal models have been constructed to investigate the feasibility of model checking on the pacemaker firmware designs of *Vitatron*, modeling components of the firmware and the environment as well. He applies mainly two approaches for verifications. One is to verify the firmware model in the context of a formal heart model, which only succeeds for heart rates in a range from low to 'average' and fails for high heart rates because of dramatic state space explosion. The other approach is to verify a design part of the firmware, which is feasible because the specific part has little dependencies on external variables. The second approach is able to find the known deadlock rather soon. However, this work still is not able to verify the whole formal model.

Macedo et. al propose a concurrent and distributed real-time model of pacemaker using VDM [8]. The sequential model is then validated primarily by scenario-based tests like absence of sensed pulses and outputted pulses at the correct time, helping to detect potential bottlenecks within suggested architectures for the subsequent development of implementation [20]. It uses the notion of event sensing and reaction to specify the pacemaker operations. They model 8 bradycardia operation modes including corresponding parameters.

Gomes et al. present a formal specification of the pacemaker system using the *Z* notation in [12], according to the informal requirements in [1]. The overall state of the pacemaker is based on a *Z* state called *PulseGen* (i.e. Pulse Generator), over which many operations are modeled. Then they validate that the formal specification satisfies the informal requirements, by using a theorem prover, *ProofPower* − *Z*, to formulate proofs of specification-to-model correspondence for high-assurance secure systems like the pacemaker.

Our approach to formally model the pacemaker is also based on the Pulse Generator, with reactions to environmental information sensed by sensors placed in chambers. Nevertheless, our approach differs from them in that we model the pacemaker on the fundamental of timed aspects, since most critical properties of the pacemaker concerns time. Our approach is able to model such timing patterns via timed processes with timed operators (which will be as discussed in Section III-B). And the time-related properties are then verified by timed refinements in *PAT*.

III. Background

A. Formal Methods

Formal methods for specifying, designing and verifying real-time safety critical systems in order to improve their safety and reliability are surveyed in [24]. [35] illustrates six well-known formal specification languages with examples, including *Z*, *VDM* [8], temporal logic, *CSP* [15], transition axioms, and so on.

CSP [15] (Communicating Sequential Processes) is a specification language for describing patterns of interaction in concurrent system. *CSP* has been used by many researchers in many areas. [19] presents the use of *CSP* in designing safety critical systems with high degree of correctness. [26] discusses the issues involved in modeling and verifying key-exchange protocols with *CSP* and its model-checking tool FDR [27]. [36] presents a methodology based on *CSP* to validate critical properties of ad-hoc networks. In addition, *CSP* is used in [31] to model check Live Sequence Charts.

Timed CSP is built as an extension to *CSP*, designed to handle concurrency combined with timing considerations, and incorporate timing information for reasoning about real-time systems [30], [10]. Operators dealing with timing information are defined in timed-CSP, such as *Wait*, *Timeout*, *Timed Interrupt* and so on, as presented in [9]. The operational semantics for *Timed CSP* is defined in [29], discussing the timing operators in detail. *Timed-CSP* has been applied widely in many areas. In [2], A. F. Ates et al. present an example of using *Timed CSP* in specifying fine-grain synchronization of multimedia systems, focusing on safety and liveness timing requirements in terms of temporal logic formulas. The specification of synchronization among media items using timed-CSP is described in [11]. In [23] protocols for updating copies and recovering from failures are specified in *Timed CSP*.

Formal verification, *model checking* [3] in specific, has been recognized as an important method to prove the correctness of distributed algorithms formally and automatically. Model checking first builds a finite state machine of a formal model of a system, and then verifies if a property, written in some temporal logic, about the system holds or not through a state space search. A counterexample can be generated when the checked property fails to hold, which explains why the formal model does not satisfy the property. With careful design, formal verification can even assure faultless system. In [17], R. Kaivola et al. present the work in replacing testing with formal verification in Intel Core i7 processor execution engine validation. A number of frameworks for model checking *CSP* or *Timed CSP* models have been proposed, such as SPIN [16], FDR [25], [27], etc.

B. *Overview of* RTS *model*

RTS (Real-time System) is presented in [34], as an dialect of *Timed CSP* with extensions like variables, event operations, and so on. Interested readers are strongly recommended to refer to [34] for its operational semantics. An *RTS* process is a timed process defined as the following BNF, where P and Q range over processes, $e \in \Sigma$ is an observable event, b is a Boolean expression on global variables or process parameters and d is an integer constant.

$P = Stop \mid Skip$	– primitives
$\mid\ action \rightarrow P$	– data-operation prefixing
$\mid\ if\ b\ then\ P\ else\ Q$	– if-then-else
$\mid\ P \Box Q$	– general choice
$\mid\ P \parallel Q$	– parallel composition
$\mid\ P;\ Q$	– sequential composition
$\mid\ P \setminus X$	– hiding
$\mid\ P \mathrel{\widehat=} Q$	– process referencing
$\mid\ Wait[d_0, d_1]$	– delay
$\mid\ P\ timeout[d]\ Q$	– timeout
$\mid\ P\ interrupt[d]\ Q$	– timed interrupt
$\mid\ P\ within[d_0, d_1]$	– react within some time
$\mid\ P\ waituntil[d]$	– wait until
$\mid\ P\ deadline[d]$	– deadline

Process *Stop* does nothing but idling. Process *Skip* terminates (possibly after some idling). Process $e \rightarrow P$ engages in event e first and then behaves as P. Notice that e may be an abstract event or a data operation, e.g. written in the form of $e\{x = 5;\ y = 3;\ \}$ or an external C# program. The data operation may update data variables (and is assumed to be executed atomically). A guarded process is written as *if b then P else Q*. If b is true, then it behaves as P, else it behaves as Q. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization or shared variables. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately. Given a channel *ch* with predefined buffer size, process $ch!exp \rightarrow P$ evaluates the expression *exp* (with the current valuation of the variables) and puts the value into the respective buffer and then behaves as P. Process $ch?x \rightarrow P$ gets the first element in the respective buffer, assigns it to variable x and then behaves as P.

Timed process constructs are used to capture common real-time system behavior patterns. Process $Wait[d]$ idles for exactly d time units. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse (since the process starts). Otherwise, Q takes over control after exactly d time units elapse. Process $P\ interrupt[d]\ Q$ behaves exactly as P (which may engage in multiple observable events) until d time units elapse, and then Q takes over control. Process $P\ within[d]$behaves as P but will not terminate before d time unit elapse. Process $P\ deadline[d]$ constrains P to terminate before d time units.

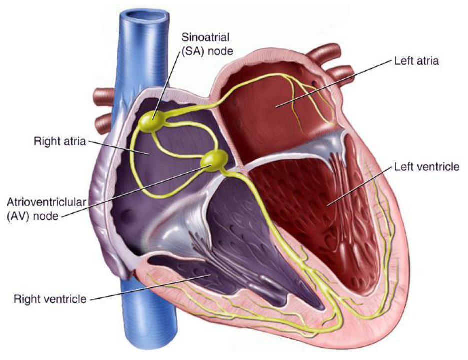

Figure 1. Heart Architecture

Our home-built model checker *PAT*[1] (Process Analysis Toolkit) is designed to apply state-of-the-art model checking techniques for system analysis. PAT [33][32] supports a wide range of modeling languages including *CSP♯* (short for communicating sequential programs), which shares similar design principle with integrated specification languages like TCOZ [21][22]. The verification features of *PAT* are abundant in that on-the-fly refinement checking algorithm is used to implement Linear Temporal Logic (LTL) based verification, partial order reduction is used to improve verification efficiency, and *LTL* based verification supports both event and state checking. Furthermore, *PAT* has been enhanced to accept *RTS* models, for verifying properties such as deadlock freeness, divergence freeness, timed refinement, temporal behaviors, etc [34].

C. *Informal Specification of Pacemaker*

As shown in Fig. 1, a heart is divided into right and left sections by the interventricular septum. Each of these (right and left) sections is also divided into upper and lower compartments known as atria and ventricles, respectively. The four main chambers of the heart are therefore the: right atrium, right ventricle, left atrium and left ventricle. A human heart can show a wide range of rates depending on the activity level of the rest of the body and the heart can suffer from several arrhythmias; such arrhythmias are often the reason that a patient needs a pacemaker.

A pacemaker is a medical device which uses electrical impulses, delivered by electrodes contacting the heart muscles, to regulate the beating of the heart, and Fig. 2 is a pacemaker device. The primary purpose of a pacemaker is to maintain an adequate heart rate by delivering electrical stimuli (paces) to the chambers of the heart, and prevent human from being harmed by low heart rate. In this paper, we adopt the system requirements of pacemaker specified by Boston Scientific [1].

The pacemaker determines when paces must be delivered based on calculating the timing of incoming contraction events. The pacemaker must deal with all possible rates

[1]http://www.patroot.com

Figure 2. A pacemaker

and arrhythmias, which makes it a complex composition of collaborating and interacting processes. Therefore, the pacemaker has several operating modes that address different malfunctions of the human heart. As described in [1], there are 18 operating modes, each of which is to deal with a certain kind of heart malfunction. The operating modes of the device are classified using a code consisting of three or four characters, as illustrated in Table I.

	I	II	III	IV
Type	Chambers paced	Chambers sensed	Response to sensing	Rate modulation
Letters	O:None A:Atrium V:Ventricle D:Dual	O:None A:Atrium V:Ventricle D:Dual	O: None T:Triggered I:Inhibited D:Tracked	Rate Modulation

Table I
THE CODE OF OPERATING MODES

In the third column, *O* stands for *No Response To Sensing*, meaning pacing without sensing, i.e. asynchronous pacing, and delivering paces without regard to senses. *T* means *Triggered Response To Sensing*, meaning a sense in a chamber will trigger an immediate pace in that chamber. *I* represents *Inhibited Response To Sensing*, when a sense in a chamber shall inhibit a pending pace in that chamber. *D* indicates *Tracked Response To Sensing*, during which an atrial sense shall cause a tracked ventricular pace after a certain delay, unless a ventricular sense has been detected beforehand.

DOO mode is the operating mode in which both chambers are paced but none are sensed, i.e. the pacemaker paces the heart without any interaction with the heart activities. *VVI* mode is the operating mode in which the pacemaker paces the ventricles with sensed data from them, and a sense in them shall inhibit a pending pace. As for *DDD* mode, the pacemaker will sense and pace both the atria and the ventricles, and an atrial sense will cause a tracked ventricular pace after a certain delay, unless a ventricular sense has been detected beforehand. A number of parameters are defined to capture the timing patterns of the pacemaker, as summarized in Table II.

Abbrev.	**Description**	**Detail**
LRL	Lower Rate Limit	The smallest number of pulses delivered per minute.
URL	Upper Rate Limit	The biggest number of pulses delivered per minute.
AVD	Atrial-Ventricular Delay	The shortest period from an atrial event to a ventricular pace.
ARP	Atrial Refractory Period	The interval following a atrial event when atrial senses shall not inhibit nor trigger paces.
VRP	Ventricle Refractory Period	The interval following a ventricular event when ventricular senses shall not inhibit nor trigger paces.
PVARP	Post Ventricular Atrial Refractory Period	The interval following a ventricular event when an atrial cardiac event shall neither inhibit atrial paces nor trigger ventricular paces.
RS	Rate Smoothing	The constraint that limits the change of pacing rate.
AT	Activity Threshold	The value that the accelerometer sensor output shall exceed before the pacemaker's rate is affected.
ReT	Reaction Time	The time required to increase the rate from *LRL* to *URL* with maximum activity.
RcT	Recovery Time	The time required to decrease the rate from *USR* to *LRL* upon cessation of activity.

Table II
THE PARAMETERS

In *VVI* mode, parameters *LRL*, *URL* and *VRP* should be considered. In *DDD* mode, parameters *LRL*, *URL*, *AVD*, *VRP*, and *PVARP* should be included. As for *DDDR* mode, all parameters presented in Table II should be considered.

Up to now, several modes have been discussed, and details about all the 18 operating modes can be found in [1]. The pacemaker shall support single and dual chamber rate adaptive pacing. The atrial and ventricular pacing pulse amplitudes shall be independently programmable. The device shall have the ability to adjust the cardiac cycle in response to metabolic need as measured from body motion using an accelerometer. The accelerometer shall determine the rate of increasing or decreasing the pacing rate.

IV. MODELING AND VERIFYING PACEMAKER IN PAT

In this section, we illustrate how the pacemaker is modeled as *RTS* models with critical properties defined for verification in *PAT*. The *RTS* model is modeled according to the informal specification in Section III-C and more detailed sources like [4].

A. *The* RTS *Model for Pacemaker*

Our model for the pacemaker system consists of 4 parts: Environment, Sensor, Rate Controller and Pulse Generator, as shown in Fig.3. *Environment* models the behavior of the human heart, i.e. the contraction events of the atria and the ventricles. *Sensor* models the sensors paced in the chambers, which senses the contraction events of the human heart and the human activity level, and thus is a communicating medium between the environment (i.e. the human heart) and the *Pulse Generator*, *Rate Controller*. *Rate Controller* models the accelerometer which determines the changing rate of the heart rate. *Pulse Generator* will generate stimuli (paces) if necessary based on the sensing data from the sensor. Our work models the behaviors and operations of all 18 modes, which are described in [1]. All modes share the same models for *Environment* and *Sensor* (for heart contractions), which are presented in Section IV-A1 and Section IV-A2 respectively. Only models for rate-modulation modes (eg. *VVIR*, *DDDR*, etc) modes include the activity sensor and *Rate Controller*, as discussed in Section IV-A3. Since each mode has different operations upon same sensing events, *PulseGenerator* is modeled separately for each mode, as illustrated in Section IV-A4.

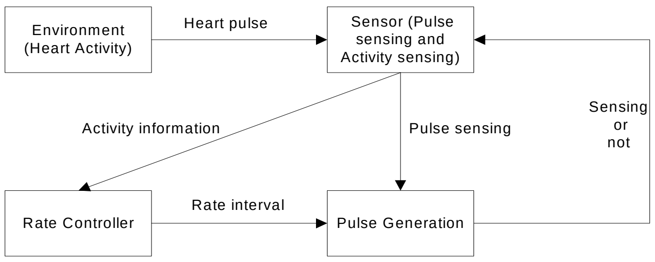

Figure 3. An Overview of the Model

Since some components of the pacemaker work in a similar way, not all details of each of the 18 modes are presented in this section. We discuss the key approach and idea of modeling the pacemaker in this paper, and interested readers can download the whole model at [37].

1) *Modeling the Environment:*

Environment is modeled as a process *Heart*, which generates atrial and ventricle contraction events, i.e. *pulseA* and *pulseV*, periodically. The process *Heart* is composed of internal choices of 6 sub-processes, which models possible malfunctions of the natural heart, as shown below.

$$
\begin{array}{ll}
\mathit{Heart} = & \\
& /*\ \mathit{normal\ case}\ */ \\
& (\mathit{pulseA} \rightarrow \mathit{Wait}[\mathit{AVD}]);\ (\mathit{pulseV} \rightarrow \mathit{Wait}[\mathit{HI}-\mathit{AVD}]);\ \mathit{Heart} \\
\sqcap & /*\ \mathit{pulseA\ missed}\ */ \\
& (\mathit{noPulse} \rightarrow \mathit{Wait}[\mathit{AVD}]);\ (\mathit{pulseV} \rightarrow \mathit{Wait}[\mathit{HI}-\mathit{AVD}]);\ \mathit{Heart} \\
\sqcap & /*\ \mathit{pulseV\ missed}\ */ \\
& (\mathit{pulseA} \rightarrow \mathit{Wait}[\mathit{AVD}]);\ (\mathit{noPulse} \rightarrow \mathit{Wait}[\mathit{HI}-\mathit{AVD}]);\ \mathit{Heart} \\
\sqcap & /*\ \mathit{dead\ heart}\ */ \\
& (\mathit{noPulse} \rightarrow \mathit{Wait}[\mathit{AVD}]);\ (\mathit{noPulse} \rightarrow \mathit{Wait}[\mathit{HI}-\mathit{AVD}]);\ \mathit{Heart} \\
\sqcap & /*\ \mathit{pulseA\ delayed}\ */ \\
& (\mathit{pulseA} \rightarrow \mathit{Wait}[\mathit{AVD}+X]); \\
& (\mathit{pulseV} \rightarrow \mathit{Wait}[\mathit{HI}-\mathit{AVD}]);\ \mathit{Heart} \\
\sqcap & /*\ \mathit{pulseV\ delayed}\ */ \\
& (\mathit{pulseA} \rightarrow \mathit{Wait}[\mathit{AVD}]); \\
& (\mathit{pulseV} \rightarrow \mathit{Wait}[\mathit{HI}-\mathit{AVD}+X]);\ \mathit{Heart};
\end{array}
$$

AVD is a variable storing the value of *Atrial Ventricular Delay*, *HI* is a variable keeping the average interval between two successive atrial/ventricular events of a normally working heart, and *X* is a random length of time which delays a pulsing event. Note that the events *pulseA* (an atrial pulse) and *pulseV* (a ventricular) event will be reused in *Sensor* processes.

2) *Modeling the Sensor:*

There are 3 sensors in of the pacemaker system, two of which are to sense the contraction events from the atria and the ventricles respectively, and the other is to sense the activity level of the human body. As the activity data is only necessary in rate-modulation modes and is not required for non-rate-modulation modes, we put the model of the sensor for activity data to Section IV-A3, where *Rate Controller* shared by rate-modulation modes are discussed.

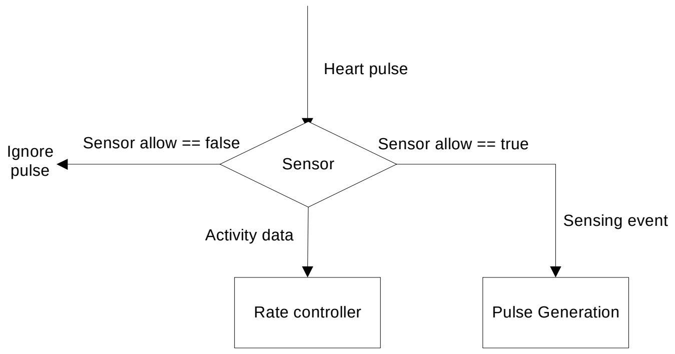

Figure 4. Atrial/Ventricular Sensor

As illustrated in Figure 4, an sensor placed in the atrium/ventricle will sense pulse from the corresponding chamber, and deliver the sensing event to *Pulse Generator* if senses are not inhibited, or ignore the pulse otherwise.

Below is the *RTS* process *AtriaSensor* modeling the sensor placed in the atria.

$$
\begin{aligned}
&/*\ \mathit{Sensor\ in\ Atria}\ */\\
&\mathit{AtriaSensor}\ =\\
&\quad [\mathit{SA} == T]\mathit{pulseA} \rightarrow \mathit{senseA} \rightarrow \mathit{AtriaSensor}\\
&\ \Box\ [\mathit{SA} == F]\mathit{pulseA} \rightarrow \mathit{AtriaSensor};
\end{aligned}
$$

SA is a Boolean variable denoting whether an atrial pulse (*pulseA*) from the environment (*Heart*) will be accepted by the pacemaker to generate an atrial pace event. If *SA* is *true*, then an event *senseA* will be executed, otherwise nothing is done and the process waits for another *pulseA* event from *Heart*. Note that *pulseA* is the common event between processes *Heart* and *AtriaSensor*, and *senseA* is the common event between process *AtriaSensor* and processes that model atrial-sensed modes (eg. *AAT* mode).

The sensor monitoring pulses in the ventricles is modeled as a process named *VentricleSensor* in a similar way, the details of which are not presented for the purpose of simplicity.

*3) **Modeling the Rate Controller**:*

Rate Controller determines the changes of heart rate according to the level of the activity of the human. We assume that a activity sensor is placed in the human body to capture such information, which is modeled as the *RTS* process *ActivitySensor* as bellow.

$$
\begin{aligned}
&\mathit{ActivitySensor} =\\
&\quad (\mathit{senseNone}\{\mathit{sensedAct}\ =\ \mathit{actData}\}\ \rightarrow\ \mathit{RateController}\\
&\ \sqcap\ \mathit{senseVLow}\{\mathit{sensedAct}\ =\ \mathit{VLOW}\}\ \rightarrow\ \mathit{RateController}\\
&\ \sqcap\ \mathit{senseLOW}\{\mathit{sensedAct}\ =\ \mathit{LOW}\}\ \rightarrow\ \mathit{RateController}\\
&\ \sqcap\ \mathit{senseMEDLOW}\{\ldots\}\ \rightarrow\ \mathit{RateController}\\
&\ \sqcap\ \mathit{senseMED}\{\ldots\}\ \rightarrow\ \mathit{RateController}\\
&\ \sqcap\ \mathit{senseMEDHIGH}\{\ldots\}\ \rightarrow\ \mathit{RateController}\\
&\ \sqcap\ \mathit{senseHIGH}\{\ldots\}\ \rightarrow\ \mathit{RateController}\\
&\ \sqcap\ \mathit{senseVHIGH}\{\ldots\}\ \rightarrow\ \mathit{RateController})\mathit{within}[0];
\end{aligned}
$$

While *Heart* and *AtriaSensor* (or *VentricleSensor*) communicate via common events, *ActivitySensor* and *Rate Controller* communicate via the global variable *sensedAct*. *sensedAct* holds the value of the activity, which will be used in *Rate Controller* to determine the new rate of the heart, as shown in the following *RTS* process. *within*[0] requires that the first event of the process will be engaged immediately once it is enabled.

$$
\begin{aligned}
&\mathit{RateController}\ =\\
&\quad (\mathit{rateControlling}\{\\
&\qquad \mathit{if}(\mathit{sensedAct}! = \mathit{actData})\\
&\qquad\quad \{\mathit{actData}\ =\ \mathit{sensedAct};\ \mathit{newInterval} = \ldots;\ \}\\
&\qquad \mathit{if}(\mathit{interval} < \mathit{newInterval})\{/*\ \mathit{increaseinterval}\ */\}\\
&\qquad \mathit{if}(\mathit{interval} > \mathit{newInterval})\{/*\ \mathit{decreaseinterval}\ */\}\\
&\quad\ \} \rightarrow \mathit{Skip})\ \mathit{within}[0]\ ;
\end{aligned}
$$

actData is the variable keeping the value of the current activity level, *interval* is the time interval between two successive pacing events, and *newInterval* is the target pulse interval based on the new sensed activity data. Note that the heart rate (number of pulse per minute) is represented by *interval* (number of milliseconds between two consecutive pulses). According to [1], the activity data is classified into seven levels, from very low (*VLOW*) to very high (*VHIGH*). *Rate Controller* decides the value of the *newInterval* based on the activity data and , and changes the interval at a certain rate based on reaction time (*ReT*) or recovery time (*RcT*).

*4) **Modeling the Pulse Generator**:*

Pulse Generator is responsible for generating paces if necessary, the operations of which are different for different modes. As discussed in Section III-C, the pacemaker may pace either atria or ventricles or both, with or without sensing the chambers. Therefore, we discuss the models of *Pulse Generator* for *AOO* mode, *VVI* mode and *DDD* mode in the following. The models of *Pulse Generator* for all modes can be found in [37].

AOO mode

Working in *AOO* mode, *Pulse Generator* generates an atrial stimulus (*paceA*) at fixed rate, without interacting with *Heart* via *Sensor*. The *RTS* process modeling the *AOO* mode *Pulse Generator* is presented as bellow.

$$
\begin{aligned}
&\mathit{PulseGen}^{\mathit{AOO}} =\\
&\qquad (\mathit{paceA} \rightarrow \mathit{Wait}[\mathit{interval}])\mathit{within}[0];\ \mathit{PulseGen}^{\mathit{AOO}};
\end{aligned}
$$

VVI mode

In *VVI* mode, *Pulse Generator* paces the ventricles according to the activity of the heart captured by *Sensor*. As shown in Fig. 5, initially if there is a ventricular sense, the *Pulse Generator* goes to state 2; Otherwise, it goes to state 4 and generates a ventricular pace after the timeout of *LRI* (i.e. the interval corresponding to *LRL*). State 2 stands for a state that a ventricular event (sensed or paced) has just happened, where $Pulse - Generator$ will be idle for a time period of *URL interval* (*URI*) and go to state 3. State 3 is similar to the initial state, except that the amount of time allowed without sense from ventricle is *(LRI-URI)*.

Following the state machine described above, the *RTS* process $\mathit{PulseGen}^{\mathit{VVI}}$ is defined as the following.

$$
\begin{aligned}
&\mathit{PulseGen}^{\mathit{VVI}} =\\
&\quad ((\mathit{atomic}\{\mathit{senseV} \rightarrow \mathit{paceV}\{\mathit{SV} = F\} \rightarrow \mathit{Skip}\})\\
&\quad \mathit{timeout}[\mathit{LRI}]((\mathit{paceV}\{\mathit{SV} = F\} \rightarrow \mathit{Skip})\mathit{within}[0]);\\
&\quad \mathit{Wait}[\mathit{URI}];\ (\mathit{enableSV}\{\mathit{SV} = T\} \rightarrow \mathit{Pace}^{\mathit{VVI}})\mathit{within}[0]);\\
&\mathit{Pace}^{\mathit{VVI}} = ((\mathit{atomic}\{\mathit{senseV} \rightarrow \mathit{paceV}\{\mathit{SV} = F\} \rightarrow \mathit{Skip}\})\\
&\quad \mathit{timeout}[\mathit{LRI} - \mathit{URI}]((\mathit{paceV}\{\mathit{SV} = F\} \rightarrow \mathit{Skip})\mathit{within}[0]);\\
&\quad \mathit{Wait}[\mathit{URI}];\ (\mathit{enableSV}\{\mathit{SV} = T\} \rightarrow \mathit{Pace}^{\mathit{VVI}})\mathit{within}[0]);
\end{aligned}
$$

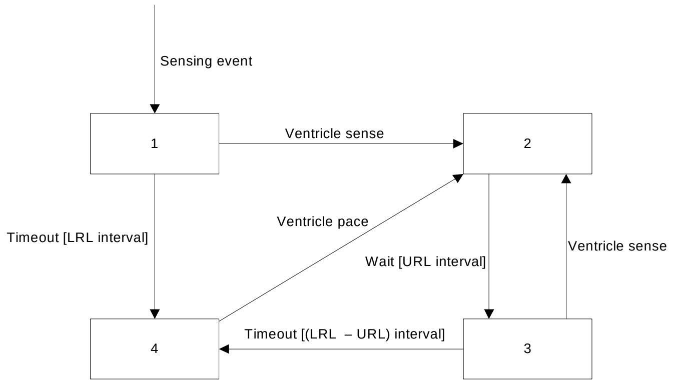

Figure 5. The State Machine of the VVI-mode Pulse Generator

When a *senseV* event is engaged, a *paceV* event will be engaged immediately. Every time when there is a pacing event, the *SV* is set as *F* (*FALSE*), disallowing *Pulse Generator* to generate a pacing event, based on a sensed event. After a *URI*'s time, the *SV* is set as *T* (*TRUE*) by event *enableSV*. In this way, the *Pulse Generator* assures that the interval between each two successive pacing event should be larger than *URI* and smaller than *LRI*.

DDD mode

As for *DDD* mode, *Pulse Generator* will pace both the atria and the ventricles based on senses from both chambers, and an atrial sense will cause a tracked ventricular pace after a delay (*AVD*), unless a ventricular sense has been detected beforehand. The *RTS* process $PulseGen^{DDD}$ is defined as follows to model such behaviors.

$$\begin{aligned}&PulseGen^{DDD} =\\&\quad (atomic\{senseV \rightarrow paceV\{SA = F;\ SV = F\} \rightarrow Skip\}\\&\quad\ timeout[LRI](paceV\{SA = F;\ SV = F\} \rightarrow Skip)within[0]);\\&\quad\ Wait[URI];\ (enableSA\{SA = T\} \rightarrow Skip)within[0];\\&\quad\ (atomic\{senseA \rightarrow paceA\{SA = F;\ SV = T\} \rightarrow Skip\}\\&\quad\ timeout[LRI - AVD - URI](paceA\{SA = F;\ SV = T\}\\&\quad\ \rightarrow Skip)within[0]);\ Pace^{DDD};\end{aligned}$$

$$\begin{aligned}Pace^{DDD} = {} & (atomic\{senseV \rightarrow paceV\{SA = 0;\ SV = 0\} \rightarrow Skip\}\\& timeout[AVD](paceV\{SA = 0;\ SV = 0\} \rightarrow Skip)within[0]);\\& Wait[URI];\ (enableSA\{SA = 1\} \rightarrow Skip)within[0];\\& (atomic\{senseA \rightarrow paceA\{SA = 0;\ SV = 1\} \rightarrow Skip\}\\& timeout[LRI - AVD - URI](paceA\{SA = 0;\ SV = 1\}\\& \rightarrow Skip)within[0]);\ Pace^{DDD};\end{aligned}$$

5) Composing the Top-level Process:

After each part of the pacemaker has been modeled as a *RTS* process, a top-level process Sys^{MODE} is defined as hierarchial parallel or sequential composition of *Environment*, *Sensor*, *Rate Controller* and *Pulse Generator*.

Non-rate-modulation Mode

For non-rate-modulation modes, the rate controller as well as the activity sensor are not considered since rate will never be changed according to the activity level of human. Generally, the top-level process for a mode is defined as the following *RTS* process.

$$\begin{aligned}&Sys^{MODE} =\\&\qquad Heart \parallel AtriaSensor \parallel VentricleSensor \parallel PulseGen^{MODE};\end{aligned}$$

In *RTS* semantics, processes composed by $\parallel$ operator will synchronize with others at each of their common events. For instance, *AtriaSensor* and $PulseGen^{AAT}$ shares the event *senseA*, and thus in process Sys^{AAT}, every time when *AtriaSensor* executes *senseA*, $PulseGen^{AAT}$ will executes *senseA* as well. Note that one of the sensor processes may be missed in single chamber sensed mode, such as *VVI* mode shown in the following.

$$\begin{aligned}Sys^{VVI} = {} & (Heart \parallel VentricleSensor \parallel VVIpace)\\& \setminus \{pulseA,\ pulseV,\ noPulse, enableSV,\\& \quad disableSV,\ senseV\};\end{aligned}$$

The operator $\setminus$ in this case is to make invisible the events in subsequent $\{\}$ block, which in fact only keeps event *paceV* in the traces of the process Sys^{VVI}. Such a design is to make the process Sys^{VVI} valid for checking timed refinement targeting a specification process only composing of event *paceV*, as the property *P*2 presented in Section IV-B.

Rate-modulation Mode

As for rate-modulation modes like *DDDR* mode, all four components described above should be included.

$$\begin{aligned}&Sys^{MODE} =\\&\qquad Heart \parallel AtriaSensor \parallel VentricleSensor \parallel RatePace^{MODE};\end{aligned}$$

$$\begin{aligned}RatePace^{MODE} = {} & ActivitySensor;\ RateController;\\& PulseGen^{MODE};\ RatePace^{MODE};\end{aligned}$$

The sub-process $RatePace^{MODE}$ of Sys^{MODE} is the one models the interactions between the *ActivitySensor*, *RateController*, $PulseGen^{MODE}$, which communicate via shared global variables such as *sensedAct* and *interval*.

B. Critical Properties

In order to formally validate that our specification for the pacemaker satisfies the informal requirements presented in [1], we define several categories of properties, against which the *RTS* model illustrated in last section is verified. These properties are defined as assertions in *PAT*'s assertion annotation language for *RTS* models, including *timed refinements*, *LTL formulae*, *reachability* and so on [34].

Deadlock Freeness (P1)

Deadlock is a common undesired state for safety-critical systems, especially in our case, a pacemaker. The existence of a deadlock state means that there is a state when the pacemaker may terminate while in fact it is expected to perform calculations. Being aware that such a terminal state may cause the loss of the life of the implanted patient, we firstly verify our specification for deadlock freeness, i.e. the specification exists no deadlock in every execution path. An assertion for verifying deadlock freeness can be defined in PAT as the following syntax, where *MODE* is the reference of the process modeling the behaviors of the pacemaker under a certain mode.

$$\#assert\ Sys^{MODE}\ deadlockfree;$$

Lower and Upper Rate Limits (P2)
Besides deadlock freeness, it is critical that the pacemaker satisfies the lower and upper rate limits (*LRL* and *URL*). If is dangerous if the pacemaker allows the heart to beat at a rate slower than the lower rate limit or faster than the upper rate limit. To verify such constraints, a specification *RTS* process is first defined to described the correct behaviors of the pacemaker with respect to *LRL* and *URL*. The process modeling the pacemaker is then verified that the set of all its timed traces will be a subset of the specification process's, i.e. timed refinement [34]. For example, to verify that the pacemaker working in *VVI* mode (i.e. Sys^{VVI}) has no conflicts with *LRL* or *URL*, the a specification process $SPEC^{VVI}_{Rate}$ representing the correct behaviors is defined as bellow.

$$SPEC^{VVI}_{Rate} = paceV \rightarrow Loop;$$
$$Loop = (paceV \rightarrow Loop)\ within[URI,\ LRI];$$

Process $SPEC^{VVI}_{Rate}$ requires that the interval between each two consequent pulses in ventricles should be larger than *URI* and smaller than *LRI*. Note that *URI* is the interval corresponding to *URL* and *LRI* corresponding to *LRL*. And the corresponding timed refinement is written as bellow.

$$\#assert\ Sys^{VVI} \sqsupseteq_T SPEC^{VVI}_{Rate};$$

Refractory Period (P3)
Refractory period is another timing requirement, during which natural stimuli (i.e. sensing events) shall neither trigger nor inhibit paces, and there are *Atrial Refractory Period* and *Ventricular Refractory Period*, as shown in Table II. For instance, while working in *AAT*, the pacemaker might not response to a sensing event from the atria in each *Atrial Refractory Period* (*ARP*). Similarly, timed refinement is used to express the property and the specification process representing the correct behaviors satisfying *ARP* is given as follows.

$$SPEC^{AAT}_{ARP} = (paceA \rightarrow Skip)waituntil[ARP];\ SPEC^{AAT}_{ARP};$$

Atrial-Ventricular Delay (P4)
For the *Tracked* modes, an atrial sense shall cause a tracked ventricular pace after a certain delay, unless a ventricular sense has been detected beforehand. Therefore, after an atrial event, there should be ventricular event within a time of *AVD*. This property is verified via timed refinement as well, and the specification process $SPEC^{DDD}_{AVD}$ is defined as the following.

$$SPEC^{DDD}_{AVD} = paceA \rightarrow Loop;$$
$$Loop = (paceV \rightarrow SPEC^{DDD}_{AVD})within[AVD];$$

Mode	P1	P2	P3	P4	P5
AAT	√	√	√	-	-
VVT	√	√	√	-	-
AOO	√	√	√	-	-
AAI	√	√	√	-	-
VOO	√	√	√	-	-
VVI	√	√	√	-	-
VDD	√	√	√	√	-
DOO	√	√	√	√	-
DDI	√	√	√	√	-
DDD	√	√	√	√	-
AOOR	√	√	√	-	√
AAIR	√	√	√	-	√
VOOR	√	√	√	-	√
VVIR	√	√	√	-	√
VDDR	√	√	√	√	√
DOOR	√	√	√	√	√
DDIR	√	√	√	√	√
DDDR	√	√	√	√	√

Table III
SUMMARY OF PROPERTIES FOR 18 MODES

Rate Controlling (P5)
In rate controlling mode, the pacemaker allows the heart rate to be modified according to the activity of the human. Therefore, rate controlling properties are to validate that whether the pacemaker working in rate modulation modes is able to change the heart rate according to changes of human activities. Such properties are expressed in *LTL* formulae. For example, the following code defines the property that a pacemaker shall finally change the heart rate equivalent to an pacing interval of 750 ms when the activity data becomes *MEDIUM*.

$$\#define\ SenseMed\ sensedAct == MED;$$
$$\#define\ RateMed\ interval == 750;$$
$$\#assert\ Sys^{XXXR}\ \models\ \Box\ (SenseMed \rightarrow (\Diamond NewRateMed));$$

Summary
We have modeled the pacemaker for all 18 operating modes as described in [1], and all of them should satisfy properties $P1$, $P2$ and $P3$. The properties that each mode should satisfy are presented in Table III.

V. EXPERIMENTS

In last section, we discuss how each mode is modeled as *RTS* processes and how critical properties are defined in assertions. Based on those models and assertions, experiments are carried to verify each property for each mode

Property	Mode	Result	#States	Time (sec.)
P1	*AAT*	True	3774	0.4766
	VVI	True	3653	0.4175
	DDD	True	15179	1.3116
	DDDR	True	221933	20.1130
P2	*AAT*	True	3990	0.3489
	VVI	True	3780	0.3281
	DDD	True	15711	1.7212
	DDDR	True	223529	24.7651
P3	*AAT*	True	3990	0.4375
	VVI	True	3780	0.3451
	DDD	True	18347	2.2235
	DDDR	True	285941	33.1382
P4	*VDD*	True	16955	2.0637
	DDI	True	16129	1.7552
	DDD	True	15711	1.7179
	DDDR	True	223529	24.9738
P5	*AOOR*	True	201743	27.4588
	VVIR	True	203496	30.2149
	DDIR	True	218113	31.4460
	DDDR	True	244801	36.6312

Table IV
RESULTS OF EXPERIMENTS

as presented in Table III. Some modes share the similar operations. As an example, *VVI* mode and *AAI* mode is only different in that *VVI* mode senses and paces only the ventricles while *AAI* mode senses and paces only the atria. For full verification, we have run all verifications for those properties of all 18 modes in Table III. However, we only present the results of 4 modes for each category of property for simplicity. The experiments were carried out on a PC with Intel Core2 CPU at 2.33GH and 3.25GB RAM, and the results are shown in Table IV.

The results show that our *RTS* models satisfy all the properties defined in last section. The time used for verifying the most complex mode (i.e. *DDDR* mode) is less than one minute, visiting more than 200,000 states. This indicates that our approach is efficient and valid for verifying the correctness of the pacemaker model.

VI. CONCLUSION

In this paper, we present a Real-Time model for the pacemaker, modeling all its 18 operating modes described in [1]. Five categories of properties are then defined and verified against the *RTS* models for all modes.

Our contributions include that we present a full model for the pacemaker regarding all 18 operating modes, which provide a complete formal specification of the pacemaker. Meanwhile, necessary properties from the five categories described in Section IV-B are defined for each mode, assuring that the pacemaker working in each mode will satisfy a set of critical properties. Moreover, our work proves that formal modeling and verifying techniques are applicable and useful for assuring critical properties for safety critical systems. Our experiences in modeling the pacemaker and constructing verification goals will be helpful for other researchers to model and verify other safety critical systems, especially real-time systems.

Since the description in [1] is quite abstract, details about complex and advanced operations of the pacemaker are unclear. Therefore, our model are restricted to focus on the sensing and pacing behaviors, and are unable to specify those complex behaviors, such as *AtrialTachycardiaResponse* [1]. Thus, one of our future work is to study the deep details of the pacemaker, and model more complex operations, allowing more detailed properties of the pacemaker to be defined and verified. During our procedure of crafting the *RTS* model for pacemaker, we suffered a lot from fixing bugs of the model for a correct and concise model. This experience has motivated us to study methods for automating create precise *RTS* models from requirement descriptions.

REFERENCES

[1] Pacemaker system specification. Software Quality Research Laboratory, 2007. http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf.

[2] A. F. Ates, M. Bilgic, S. Saito, and B. Sarikaya. Using Timed CSP for Specification Verification and Simulation of Multimedia Synchronization. *IEEE Journal on Selected Areas in Communications*, 14(1):126–137, 1996.

[3] C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, New York, 2008.

[4] S. S. Barold, R. X. Stroobandt, and A. F. Sinnaeve. *Cardiac Pacemakers Step by Step : AN ILLUSTRATED GUIDE*. Futura, Blackwell Publishing, 1st edition, 2004.

[5] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In *SFM*, pages 200–236, 2004.

[6] J. P. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8:189–209, 1993.

[7] J. P. Bowen and A. Tanenbaum. Formal Methods in Safety-Critical Standards. *IEEE Computer*, 27:168–177, 1993.

[8] c. B. Jones. Systematic software development using vdm - teaching notes. 1995.

[9] J. W. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, New York, NY, USA, 1993.

[10] J. W. Davies and S. Schneider. A Brief History of Timed CSP. *Theor. Comput. Sci.*, 138(2):243–271, 1995.

[11] S. A. Ehikioya. A Formal Approach to Multimedia Information Systems Development. In *IEEE Int'l Conf. Systems, Man and Cybernetics*, volume 2, pages 1245 – 1249, 1998.

[12] A. O. Gomes and M. V. M. Oliveira. Formal Specification of a Cardiac Pacing System. In *FM*, pages 692–707, 2009.

[13] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. Weerdenburg. The Formal Specification Language mCRL2. In *MMOSS*, 2006.

[14] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[15] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[16] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[17] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E., and A. Naik. Replacing Testing with Formal Verification in Intel$^{\circledR}$ Core^{TM} i7 Processor Execution Engine Validation. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 414–429, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] J. C. Knight. Safety critical systems: challenges and directions. In *ICSE*, pages 547–550, 2002.

[19] Y. K. H. Lau. The Design of Distributed Safety Critical Software using CSP. In *IEE Colloquium Safety Critical Software in Vehicle and Traffic Control*, pages 8/1–8/5, 1990.

[20] H. D. Macedo, P. G. Larsen, and J. S. Fitzgerald. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In *FM*, pages 181–197, 2008.

[21] B. Mahony and J. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *In Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, 1998.

[22] B. Mahony and J. Dong. Timed Communicating Object Z. In *IEEE Transactions on Software Engineering*, volume 26, pages 150–177, 2000.

[23] N. Ogura, K. Saisho, and A. Fukuda. Design of Protocols in Timed CSP for Highly Reliable and Available Client-Server System. In *APSEC*, pages 495–502, 1997.

[24] J. S. Ostroff. Formal methods for the Specification and Design of Real-Time Safety Critical Systems. *J. Syst. Softw.*, 18(1):33–60, 1992.

[25] A. W. Roscoe. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[26] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *In 8th IEEE Computer Security Foundations Workshop*, pages 98–107, 1995.

[27] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[28] J. Rushby. Critical System Properties: Survey and Taxonomy. *Reliability Engineering and Systems Safety*, 43(2):189–219, 1994. Research report CSL-93-01.

[29] S. Schneider. An Operational Semantics for Timed CSP. *Inf. Comput.*, 116(2):193–213, 1995.

[30] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley and Sons, 2000.

[31] J. Sun and J. S. Dong. Model Checking Live Sequence Charts. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 529–538, Washington, DC, USA, 2005. IEEE Computer Society.

[32] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *21th International Conference on Computer Aided Verification (CAV 2009)*, 2009.

[33] J. Sun, Y. Liu, J. S. Dong, and H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *10th International Conference on Formal Engineering Methods (ICFEM 2008)*, 2008.

[34] J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600, 2009.

[35] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

[36] I. Zakiuddin, M. Goldsmith, P. Whittaker, and P. H. B. Gardiner. A Methodology for Model-Checking Ad-hoc Networks. In *SPIN*, pages 181–196, 2003.

[37] M. C. Zheng and L. A. Tuan. The *RTS* model of pacemaker. http://www.comp.nus.edu.sg/˜pat/pacemaker-rts.zip, 2009.