TOWARDS A VERIFIED CARDIAC PACEMAKER

Asankhaya Sharma
asankhaya@nus.edu.sg

Technical Report
November 2010

TOWARDS A VERIFIED CARDIAC PACEMAKER

Abstract – *In this report we describe our attempt to solve the pacemaker formal methods challenge issued by the software quality research laboratory (SQRL). Based on the informal specification released by Boston Scientific we formally model the pacemaker in PROMELA using the SPIN model checking tool. Each component of the pacemaker is modeled as a separate process in PROMELA. We model all the 18 modes of the pacemaker along with advanced features like rate controlled pacing and hysteresis. Using LTL properties we verify desired behavior of the system, all timing requirements from the specification are successfully verified in our model. Taking guidance from the PROMELA model we also show how it is possible to generate C code for the implementation. The implementation is then validated against the PROMELA model using the same tool SPIN. This is the first attempt which we know of where the pacemaker software is verified and validated entirely end to end (from requirements to implementation) using formal methods. In addition to our original sequential model (PROMELA and C) we show that it is possible to extend the model to Concurrent and Distributed setting. This allows us to capture many more behaviors of the system and thus generate a more flexible model.*

1. Introduction

The pacemaker challenge was issued by SQRL, McMaster University in 2007. An artificial cardiac pacemaker is a critical system which is used to treat patients with various heart conditions in which the natural pace generation is affected. The device is actually implanted inside the patient's body and generates stimulated paces to the heart using electric impulses. As in any critical system it is of utmost importance that the device itself should not contain any bugs or defects. A good way to ensure the reliability of the device is to use formal methods in order to capture and validate important requirements for the functioning of the device. Most of the devices are proprietary and the implementation details are only known to the manufacturer. In order to facilitate the use of formal methods and to encourage participation from a wide audience Boston Scientific released into public domain the system specification [1] for a previous generation pacemaker. This informal specification is the basis of the modeling and verification results described in this report. A pacemaker system consists of several components some of them are illustrated in figure 1.

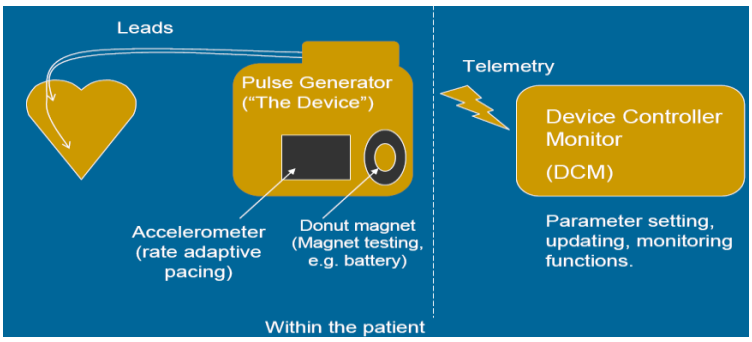


Figure 1 – Components of a Pacemaker

Among the various components (in figure 1) the most critical ones are on the left, these components are physically implanted inside a patient via surgery. The focus of our approach is to use formal methods to model and validate these components since they are most important for the entire pacemaker system to function. Another aspect of this problem is to model the behavior of the heart (or environment) which interacts with the pacemaker in different ways. Some of the heart conditions are characterized by the kind problems in the heart and others capture the normal functioning of a human heart. Timing requirements play a major role in pacemaker software, since the pulses to heart have to be delivered at specified intervals and failure to do so may be fatal. There are several programmable parameters in the pacemaker system which provides flexibility to control the rate, time, amplitude and other properties of a pulse. We have tried to capture all the parameters which may influence the timing requirements of the system. This report is organized as follows, in the next section we look at some related work in this domain, in section 3 we describe our PROMELA model, section 4 we define various desired LTL properties and show verification results, section 5 and 6 show how to extend the model to a concurrent and distributed system respectively, finally we conclude in section 7.

2. Related Work

The cardiac pacemaker challenge has been known since 2007 and there have been 3 published papers which describe the attempts to solve it. The first is due to H. D. Macedo, P. G. Larsen and J. Fitzgerald [3], in this paper they show it is possible to evolve the model of the pacemaker system from a sequential to concurrent and finally to a distributed model. In various models different aspects of the system are verified and validated. Our approach though is inspired by them by is different in two important ways, firstly we specify and verify same properties in all the 3 models and secondly their approach uses VDM while we use SPIN and hence we are able to verify the implementation C code as well (which is supported by SPIN). In [2] Artur Oliveira Gomes and Marcel Vin´ıcius Medeiros Oliveira show how it is possible to formally specify the cardiac pacemaker using Z, they fail to capture advanced features like rate controlled pacing and hysteresis which we are able to model in our system. The work which comes closest to ours is [4] where Luu Anh Tuan, Man Chun Zheng and Quan Thanh Tho use CSP to model the system and PAT to verify the desired properties of the pacemaker. We are also able to model and verify all the properties they describe and in addition we can also validate the C Code against the same properties.

The use of SPIN for this project was partially driven by the flexibility and comprehensiveness of the tool. SPIN is a well-known model checking tool which has been under constant development and improvement. This report describes the first such attempt that we know of using SPIN to validate a cardiac pacemaker system. SPIN has successfully been used to verify important mission critical software like the NASA Mars Rover [7]. All the three models we describe in this report are modeled as PROMELA specifications in SPIN; in addition to PROMELA code, SPIN (version 4 onwards) also allows limited use of embedded C as part of the model. We exploit this feature to validate the C code we have written using the PROMELA model as guidance. This allows us to validate the C code against the same LTL properties which were used to verify the PROMELA model, thus ensuring end to end verification of the whole system.

3. Modeling the Pacemaker with SPIN

In the SPIN modeling tool each component of the pacemaker can be modeled as a different process. These processes are specified in PROMELA, which captures the behavior of the processes. In order to communicate with each other these processes can use global variables or channels. In the first such model (sequential) that we describe here we use 4 such processes. As shown in figure 2, each of the process can read and write any of the global variables illustrated by the both sided arrows. In case of a sequential model at any given instant only a single process is running so there is no problem with letting each process read or write any global variable. The Heart process captures the environment which is in constant interaction with the Pace Generator via a Sensor. The Pace Generator process models the generation of pulses. The Update Timers process is used to simulate a global clock in SPIN, this enables us to specify the desired timing properties in LTL and verify them.

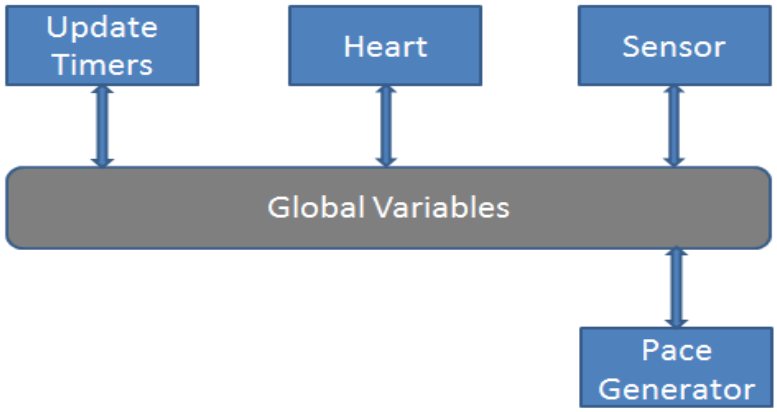


Figure 2 – Modeling Pacemaker using SPIN

We believe these 4 processes are sufficient to capture most of the common modes and the important timing requirements of the pacemaker system. The default execution semantics in SPIN is to allow interleaving of the various PROMELA processes (concurrent execution). In order to restrict it to a sequential model we need to simulate the sequential behavior by allowing only one process to run at any given time. In the next section we describe how to do this in SPIN.

4. Sequential Model

In order to ensure sequential execution semantics for the PROMELA model we construct a static scheduler which executes each of the processes in a round robin fashion (one after the other). At any given instant a schedule variable 'schd' keeps track of which processes is currently allowed to execute. All processes have full access to global variables and that is how they communicate with each other. In addition to the processes shown in figure 2 we need to add two more processes viz. Accelerometer and Rate Controller. This enables us to model advanced features like rate controlled pacing and hysteresis. Now our model looks as shown in figure 3. Next we describe all the processes and the behaviors of the cardiac pacemaker that they capture.

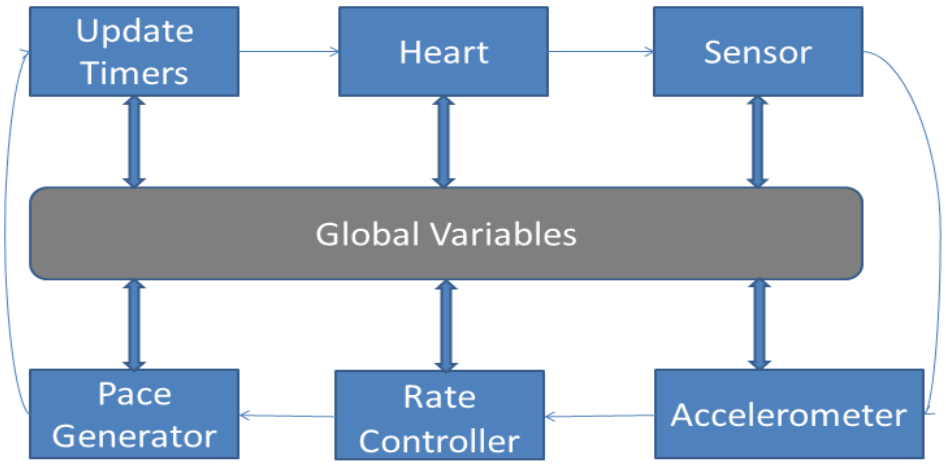


Figure 3 – Sequential Model of Pacemaker

Update Timers – This process maintains a global clock which is incremented in every round, the clock is used by other processes to express timing requirements of the system. It is reset after one full AV cycle. Also some of the global variables which capture the pulses and senses are reset along with the various timestamps which capture the time of various pulses and senses.

Heart – The heart process models the environment in which the pacemaker is run, we model the following four different behaviors of the heart.

- Normal – Wait NR, Pace A, Wait AVD, Pace V, Repeat
- Miss Ventricle Pace – Wait NR, Pace A, Wait AVD, Skip, Repeat
- Dead – Wait NR, Skip, Wait AVD, Skip, Repeat
- Non Deterministic – Wait NR, May Pace A, Wait AVD, May Pace V, Repeat

where, NR is Normal Rate, A is Atria, V is Ventricles and AVD is AV Delay.

Sensor – This process captures the senses from the Heart and the Pace Generator. It also records the time of the various pulses which is used to verify the refractory period property.

Accelerometer – This is another sensor in the pacemaker system which detects the motion of the human body. Based on the raw acceleration data there are 5 different activity thresholds mentioned in the informal specification [1]. We use these thresholds and set the response factor (RF) which is in turned used by the Rate Controller to set the appropriate rate of pacing.

Rate Controller – This process controls the rate at which the artificial pulses are delivered by the Pace Generator. In event of increased body activity such as during exercise, the Accelerometer would detect the motion and set up the appropriate RF, which is used by the Rate Controller to determine the appropriate pacing rate. This rate is used by the Pace Generator to wait for the given time before delivering the next pulse.

Pace Generator – This is the most important component of the pacemaker system. Pace Generator is responsible for generating the pulses according to the mode of operation of the device. In total there are 18 different modes, these modes specify which portions of heart are sensed and/or paced, how the paces are delivered and the whether they are rate controlled. Each mode can be written as WXYZ where,

- W – Specifies which chambers of heart are paced and can take values A (Atria), V (Ventricles) and D (both).
- X – Specifies which chambers of heart are sensed and can take values A, V, D and O (not specified).
- Y – Specifies how the senses are handled and can take values I (Inhibited), T (Triggered), D (Tracked) and O.
- Z – Specifies if the pacing is rate controlled or not (value R for Rate Controlled and unspecified otherwise).

In what follows we group the various modes together based on similar behavior and describe each group.

VOO, AOO and DOO – The first group consists of the asynchronous pacing modes, where the paces are delivered at fixed intervals without regards to any of the senses. Figure 4 illustrates this by use of a state diagram for the VOO mode, as can be seen form the diagram we start in some initial state 1 and wait for minimum time required for pacing to get to state 2 and then deliver the pulse at V in state 3 and go back to the state 1 after another duration of reset time. AOO and DOO mode can be handled in the same fashion by pacing only Atria in first case and both the chambers of Heart in the second case.

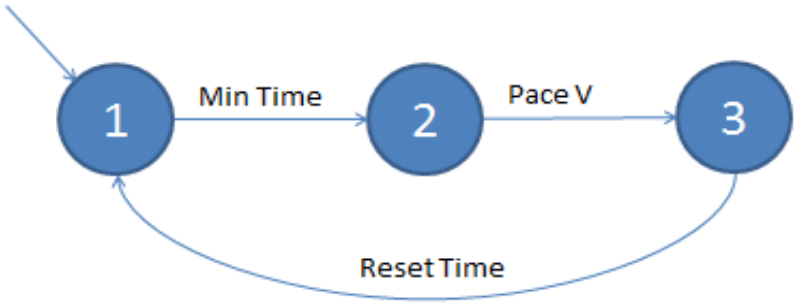


Figure 4 – VOO Mode

VVI, AAI, and DDI – This group consists of the inhibiting pacing modes, where if we detect a sense in that chamber before pacing the next generated pace is inhibited. This is shown for the VVI mode in figure 5 with similar states as in figure 4 but the only difference now is that in state 2 if you sense a pulse in the ventricles we go back to state 1, thus inhibiting the next pending pace (state 3). Similarly for AAI and DDI mode, we can do the same each time inhibiting the pending pace if a sense is detected in that chamber.

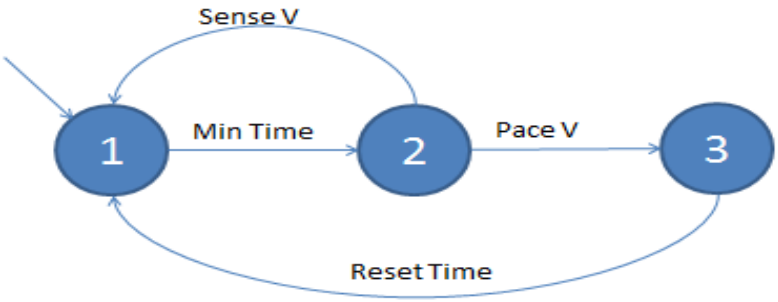


Figure 5 – VVI Mode

VVT and AAT – These are the triggered pacing modes, in which if there is a sense in a chamber it triggers an immediate pulse in that chamber. The VVT mode is illustrated in figure 6, which shows if we sense a pulse in state 1 we immediately trigger a pace by going to state 3. Similarly we can construct the AAT mode. In the triggered pacing modes the other properties of the pulse are important like Width, Amplitude etc. Since these modes are used for a weak heart just the mere fact that the particular chamber is paced is not enough to capture the full behavior of the pacemaker system. These are the only properties of the pulse which are not related to the timing constraints and can be easily added on to the model. We not consider and model them in our system since our main focus is to verify the timing requirements.

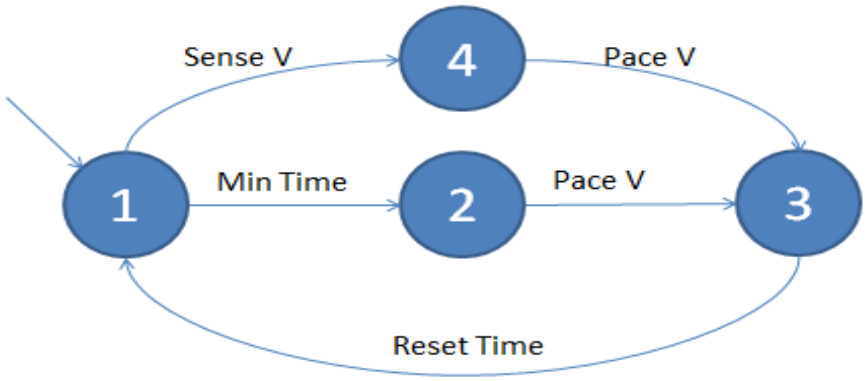


Figure 6 – VVT Mode

VDD and DDD – These are the tracked pacing modes, in which whenever there is a pulse in Atria a tracked pulse after fixed time (AVD) is generated in Ventricles. In addition if there is a pulse in Ventricles the impending pace is inhibited. This is shown in figure 7 for the VDD mode, where a new state 4 is added which handles the tracked pacing and transitions to state 3 after waiting for AVD time. The DDD mode is similar in function to the VDD mode just that the Atria are also asynchronously paced (like in AOO) and that can be sensed to start the tracked pulse in the Ventricles. In all the pacing modes discussed so far the time which is taken to transition from one state of other is fixed thus all paces are delivered at the same rate, next we will see how this can be made dynamic and the rate of pacing can be changed.

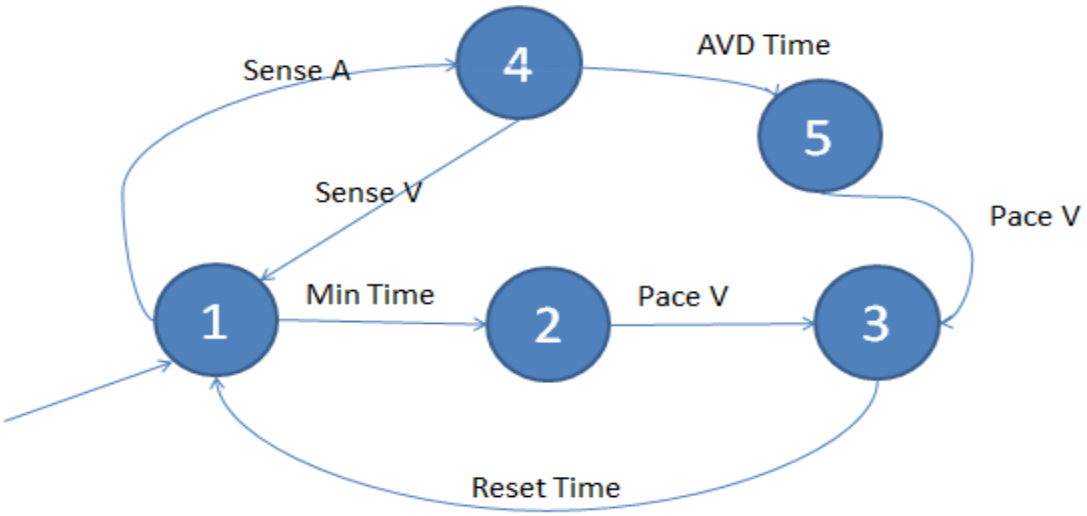


Figure 7 – VDD Mode

Rate Controlled Pacing – In rate controlled pacing the rate at which the paces are delivered is varied according to the activity level of the human body which is detected using the Accelerometer. There are 8 rate controlled modes viz. VOOR, AOOR, DOOR, VVIR, AAIR, DDIR, VDDR and DDDR. As mentioned earlier that the Rate Controller sets the rate for pacing based on the RF. How this RF can be used to vary the rate is illustrated in figure 8 for the VDDR mode. Recall that in VDD we used to wait for duration of Min Time before going from state 1 to 2, now this interval is based on the rate decided by the Rate Controller. So, before going to state 2 we wait for Min Time + RF*Increment, where Increment represents the minimum increment allowed in the change of rate. The RF values are set so that the following relation holds, Min Time + max (RF)*Increment < Max Time. This ensures that the Rate Controller does not change the rate beyond what is allowed by the Max Time. Similarly for other modes we can include rate controlled pacing by changing the time between each paced pulses.

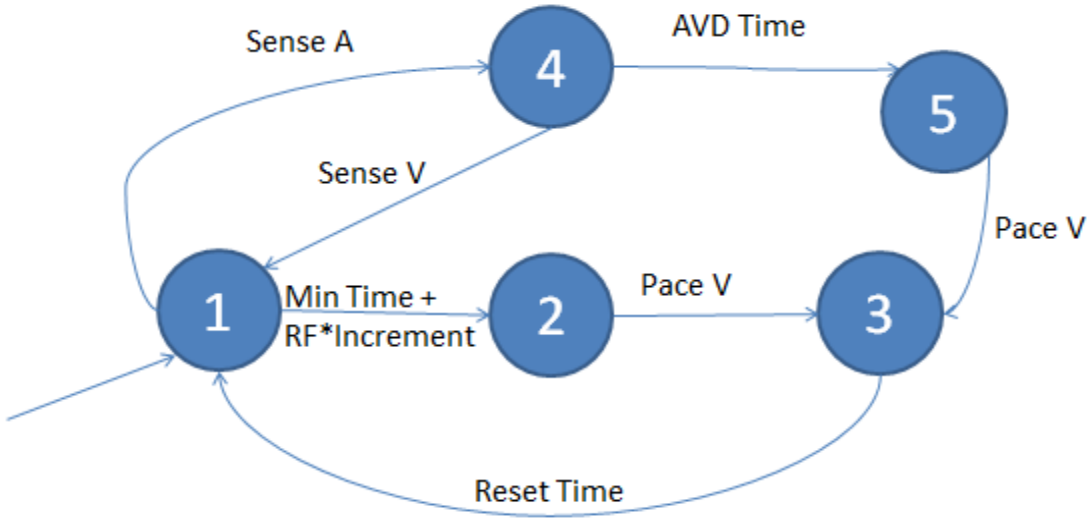


Figure 8 – VDDR Mode

Hysteresis Pacing – This feature is valid only for inhibiting and tracking modes (with or without rate controlled pacing). When Hysteresis mode is turned on after every sense there will be a longer time period before next pace. This will make sure if there is another sense then it will inhibit the pending pace (recall from above that it is indeed the case with inhibiting and tracking modes). This longer time period to wait for the next pace can again be set using the RF to the maximum value, this will ensure that after every sense there is a longer time period of waiting. This is illustrated in figure 9 for the VDDR mode, at state 4 we wait for a longer period which is determined by AVD Time + RF*Increment before we go to the state 3. The rest of the diagram is same as figure 8. Similarly for other modes we can use the RF to set the time period for hysteresis pacing. With hysteresis mode we now have an additional 10 modes viz. VVIOH, AAIOH, DDIOH, VVIRH, AAIRH, DDIRH, VDDOH, DDDOH, VDDRH and DDDRH, where H represents hysteresis pacing.

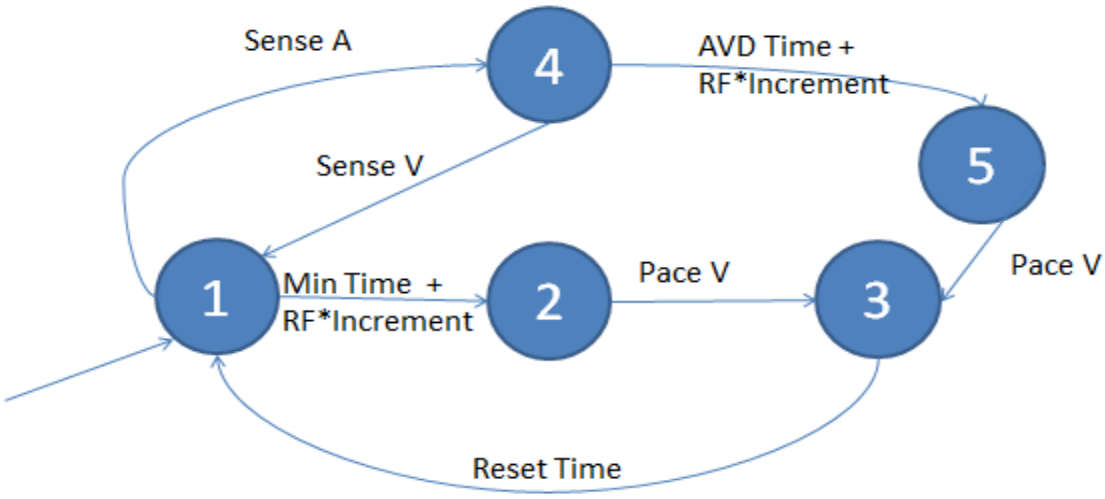


Figure 9 – VDDR Mode with Hysteresis (VDDRH)

4.1 LTL Properties

In the previous section we saw how we can model all the modes of the pacemaker in SPIN using PROMELA. The desirable properties of the system can be written in LTL, which can be used by SPIN to verify the PROMELA model. In the following section we describe the various LTL properties which we verified in our model. Most of these properties represent timing requirements which are captured from the specification.

Deadlock – This property captures the fact that there is no deadlock in the system. The property of deadlock is actually specified by default in SPIN when it checks for invalid end states of the model. This ensures that all the processes are indeed deadlock free.

Pace Limit – There is an Upper Rate Limit and a Lower Rate Limit specified in the requirements which means that the rate of pacing at any time should be between these two values. In order to ensure this we construct two LTL formulae, LRLURLA.ltl and LRLURLV.ltl. Where LRLURLA.ltl can be written as, G (Rate of pacing A < URL && Rate of pacing A > LRL) this property captures the Pace Limit for Atria. Similarly we can construct the LTL formula for Ventricles.

AV Delay – This property represents the fact that between every A and V pulse the delay is always less than a fixed time period. Hence we can construct a LTL formula AVD.ltl as, G (AV_Delay < Fixed_AVD)

Refractory Period – The time taken between a sense and a pace in that chamber is called the Refractory Period; this property ensures that there is always some time delay between subsequent paced events in the Heart. There are three such properties which we capture in our model ARP.ltl, VRP.ltl and PVARP.ltl. Where ARP.ltl can be a LTL formula of the following kind, G ((Last_PacedA – Last_SensedA) > ARP)

Inhibiting Property – This property is valid for inhibiting pacing modes and checks if the pending paces are inhibited in presence of a sense in that chamber. These are represented by two LTL formulae AAI.ltl and VVI.ltl. Where AAI.ltl can be constructed as G (sense A -> not pace A)

Triggering Property – This property is valid for the triggered pacing modes and checks if the paces are triggering in the chamber whenever a sense is detected in that chamber. These are represented by two LTL formulae AAT.ltl and VVT.ltl. Where AAT.ltl can be represented as G (sense A -> pace A)

Tracked Property – This property is valid for the tracked pacing modes and checks if the tracked pace V is delivered after a sense A and inhibited if there is a sense V before that. It is represented by the LTL formula XDD.ltl and can be written as G (sense A - >F (pace V && AV_Delay < Fixed_AVD))

The properties described above are the 7 basic properties of the pacemaker system which capture the basic timing constraints and the functions of various modes. Out of these 7 the first 4 properties, Deadlock, Pace Limit, AV Delay and Refractory Period are generic properties of the system, while the last 3 are only valid for the specific kind of modes. In addition to these 7 properties we now describe the following 3 advanced properties which are valid for rate controlled and hysteresis pacing modes.

Rate Limit – This property captures the fact that even in a rate controlled pacing mode the rate of the pacing does not exceed the Maximum Sensor Rate as specified in the requirement. It is represented by two LTL formulae LRLURLA_R.ltl and LRLURLV_R.ltl where LRLURLA_R.ltl can be written as G (Rate of pacing A < Max Sensor Rate).

Rate Control Limit – This property captures the rate modulation in the paces, this ensures that the rate of pacing changes according to the activity of the human body as measured by the accelerometer. It is represented by two LTL formulae LRLURLA_RC.ltl and LRLURLV_RC.ltl. where LRLURLA_RC.ltl can be written as GF (Rate of pacing A == RF * Activity Threshold)

Hysteresis Limit – This property captures the fact that in a hysteresis pacing mode the pace after a sense is delivered after waiting for a longer period which never exceeds the hysteresis limit. It is represented by the following LTL formulae AAI_H.ltl, VVI_H.ltl and XDD_H.ltl. where AAI_H.ltl can be written as G ((Last_PacedA – Last_SensedA) < Hysteresis Limit).

Thus we have a total of 10 desired properties of the system represented by 18 LTL formulae. We use the SPIN model checker to verify these properties for our PROMELA model; the results are discussed in the next section.

4.2 Verification of Sequential Pacemaker Model

We use SPIN to verify the various properties described in the previous section, table 1 collects the results of verifying these properties in our model. A cross mark indicates we were able to verify the property while a blank indicates that the property does not make sense for that particular mode. As is clear from table 1 that not only are we are able to verify the generic but also the mode specific properties in our model. Since the focus here is to validate the timing constraints of the Pace Generator if a particular heart behavior doesn't make sense for a particular mode (e.g. AOO mode, Dead Heart and Pace Limit for V) we just ignore it.

LTL Property	VOO	AOO	DOO	VVI	AAI	DDI	VVT	AAT	VDD	DDD
Deadlock	X	X	X	X	X	X	X	X	X	X
Pace Limit	X	X	X	X	X	X	X	X	X	X
AV Delay			X			X			X	X
Refractory Period	X	X	X	X	X	X	X	X	X	X
Inhibiting				X	X	X				
Triggering							X	X		
Tracking									X	X

Table 1 – Verification Results of 7 Basic LTL Properties for Sequential Model

In addition to the 7 basic properties we are also able to verify the advanced properties like rate control and hysteresis limit. The results of the verification are shown in table 2, where XXXH represents a mode with hysteresis enabled. Thus we have successfully verified the sequential model of the pacemaker system for all 10 desired properties. Now we will see how we can use this PROMELA model as guidance to write an implementation of the system in C.

LTL Property	VOOR	AOOR	DOOR	VVIR	AAIR	DDIR	VDDR	DDDR	VVIH	AAIH	DDIH	VDDH	DDDH
Rate Limit	X	X	X	X	X	X	X	X					
Rate Control	X	X	X	X	X	X	X	X					
Hysteresis Limit									X	X	X	X	X

Table 2 – Verification Results of 3 Advanced LTL Properties for Sequential Model

4.3 C Code Generation

The syntax of PROMELA is very close to that of C but with many differences and distinct features like non deterministic branching and communication using channels. Ideally it would be good to have an automated compiler which would generate C code from PROMELA. In fact people have tried to build such a system with limited success [5]. The features of non-deterministic branching and communications using channels are absent in C. The system in [5] uses the code from State Transition Matrix built internally by SPIN (pan.m and pan.t) to generate a C implementation which simulates the behavior of the PROMELA model. For our purposes this is not appropriate as the use of an explicit State Transition Matrix would generate a very large amount of C code. Moreover the target pacemaker hardware platform is a PIC based microcontroller which has many restrictions on the kind of C that can be compiled and run on it (Small Device C compiler).

We would want to build our own translator which would take PROMELA source and convert it to C code which can be compiled by the Small Device C compiler to generate an executable for the target hardware. However such a compiler would need to formally prove that the translations from PROMELA to C are going to preserve the LTL properties. This we consider to be beyond the scope of this work as it would require a non-trivial effort in development of such a compiler. Instead we take a more pragmatic approach and generate hand written C code from PROMELA following certain guidelines in order to ensure the properties are preserved. In particular we do the following steps

- Remove all Non Deterministic Branching from the PROMELA code
- Remove the static scheduling code since C code is sequential
- Remove all non-compatible data types like chan, mtype etc.
    - Replace mtype with enum.
    - Use variables instead of channels where possible.

By mechanically doing the translation by hand we were able to generate C code for the pacemaker which is based on the sequential PROMELA model. All the processes in the PROMELA model are represented by different functions with same name in C. The main () function in C takes over the role of init process in PROMELA which initializes and calls all the other functions. The name of all the variables is the same and the execution semantics of the program mimic that of the processes (which are also called one after the other in a round robin fashion).

As is clear from the above description, we took care while converting the PROMELA model into C code so as to preserve the timing constraints. Since this code used a formally specified and validated model as guidance we can be confident that the code written is more likely to be bug free. However we cannot be certain since the translations that we did are mechanical in nature and based on heuristics, not formal proofs. In order to validate this C code we can either use some C model checker like CBMC or do proper testing to ensure we cover a large number of scenarios. In the first case we would need to again specify the requirements as assertions in code and in second we would need to cover a large number of test cases (still we won't be certain that we have covered all possible behaviors). We describe our approach in the next section which avoids both this problems.

4.4 Validation of C Code

From version 4.0 onwards SPIN allows the use of embedded C code as part of the PROMELA model, this embedded code is treated syntactically and executed as single step by the SPIN. But this can still be used to validate parts of implementation directly as shown in [6]. Since the C code that we generate used the underlying PROMELA model as guidance we can easily integrate this back into SPIN using embedded C feature of tool. Recall from the previous section that we do a mechanical translation of PROMELA to a sequential C program, where each process in PROMELA now becomes a function in C. Now among these function is also the Pace Generator () which is the most important and critical component of the entire pacemaker system. Let us first see how we can try to validate this function using SPIN.

The PROMELA model also has a process of the form proctype Pace Generator (), we remove all the code inside the process and add the code from C using c_code { } construct in SPIN. This embedded C code is now part of the PROMELA model and since the rest of the processes are still the same we can validate the LTL properties which would use this C code as part of a process. There is still one important details missing SPIN cannot access the C variables defined inside the embedded C code, however recall that during the mechanical translation we kept the variable names and types intact. Hence we can add extra code at the end of embedded C code from Pace Generator () function which will update the corresponding PROMELA variables from the values changed through the C function. Using the now.variable_name construct we can update PROMELA variables at the end of function. Now when we try to verify the properties in SPIN the values of the corresponding variables are updated through the actual implementation code embedded inside a process in SPIN. Hence we have validated that the implementation of Pace Generator () indeed satisfies all the desired properties. This is illustrated in the following schematic.

PROMELA: Update Timers -> Heart -> Sensor -> Pace Generator- > Update Timers

C Code: Timers () -> Heart () -> Sensor () -> Pace Generator () -> Timers ()

Replace Pace Generator Process with code from C function Pace Generator ()

PROMEA: Update Timers -> Heart -> Sensor -> c_code {Pace Generator () now.PROMELA_var = C_var} -> Update Timers

Verify LTL Properties

Replace Heart with code from C function Heart ()

PROMEA: Update Timers -> c_code {Heart () now.PROMELA_var = C_var} -> Sensor -> Pace Generator-> Update Timers

Verify LTL Properties

…

Continue for all functions in implementation

Once we have validated a single function we can replace it with the PROMELA process again and do the same for other functions in the implementation. Thus we can use this modular approach for verification of the entire implementation using the same SPIN model which was used to specify the system. Once we have completed the validation of all the functions against the desired LTL properties we can be sure that our implementation is reliable. This shows how to apply formal methods at every stage of the development cycle to generate pacemaker software which is verified and validated end to end.

5. Concurrent Model

In the previous sections we described the end to end verification of a sequential model of the cardiac pacemaker. Even though we were able to generate code which is fully validated against the requirements, the model we choose being sequential is not very flexible. In this section we will show how we can evolve the model to a concurrent system which will capture many more behaviors than the previous model and still verify important desired properties of pacemaker. Recall that the SPIN has a interleaved semantics of various processes and we had to use a static scheduler in order to ensure sequential behavior among the processes. If we do not ensure that, any process is allowed to run at any given time in a full concurrent setting.

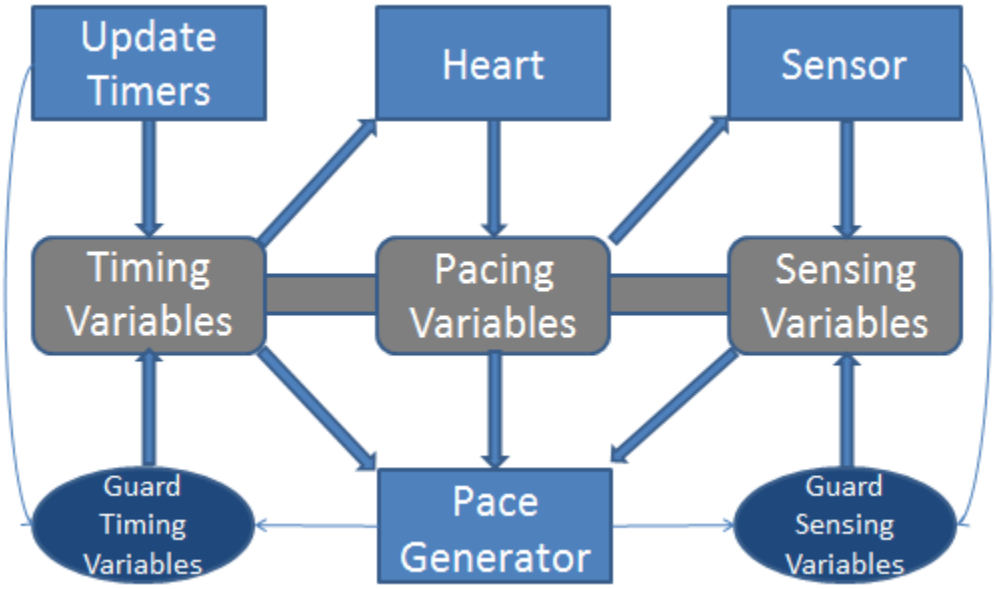


Figure 10 – Concurrent Model of Pacemaker

In order to enforce the desired timing constraints and still allow concurrent interleaving among the various processes we need to handle the conflicts that can arise when more than one process requests access to the same resource or variable. As illustrated in the figure 10 in our concurrent model as before we have global variables which are used for communication among the processes but now the arrows are one sided and the memory is also divided into various sections. Thus we enforce orderings among the various read and write to variables in a particular section. For example, only Update Timers process is allowed to write to the Timing Variables while Heart and Pace Generator are allowed to read, but only

in the order, that always the Heart reads the Timing Variables followed by the Pace Generator. These orderings along with the use of atomic operations to update each section of variables (Timing Variables) ensure that we do not have a Read-Write or Write-Read conflict among the processes. Now in order to prevent a Write-Write conflict between the processes we guard the write to the variables using special flags. These flags are reset only after a full AV cycle and then the variables are all reset to original values.

In other words, the execution semantics is such that between AV cycles the processes all execute concurrently, ordering their read and write to the section of variables and at the end of a AV cycle all processes must finish their execution before the guarded timing and sensing variables are reset and a new cycle begins. The modeling of each of the processes is the same as described in the section 4. The only change is that the processes are now allowed to interleave their execution as explained above. Similarly all the desired LTL properties are also the same as before. In the next section we will discuss the results of the verification of these properties on the concurrent model.

5.1 Verification of Concurrent Pacemaker Model

For the model of figure 10 we specified all the basic modes of operation of the Pace Generator, the advanced modes are not part of the concurrent model as of now. These modes are verified again the same generic LTL properties of Deadlock, Pace Limit and AV Delay. The results are summarized in table 3. As was the case before we were successfully able to verify these properties, even though in the concurrent model the numbers of states explored by SPIN were much more compared to the sequential one.

LTL Property	VOO	AOO	DOO	VVI	AAI	DDI	VVT	AAT	VDD	DDD
Deadlock	X	X	X	X	X	X	X	X	X	X
Pace Limit	X	X	X	X	X	X	X	X	X	X
AV Delay			X			X			X	X

Table 3 – Verification Results of Generic LTL Properties for Concurrent Model

The time and memory taken by SPIN to finish verification for the concurrent model was much higher than the sequential one thus showing that this model is more flexible in terms of the number of behaviors that it captures. In order to verify other properties and add rate controlled pacing what we found was it was very tedious to ensure proper timing constraints, since ordering between the processes grew. While adding more processes to this model in order to ensure the orderings we would need to restrict a large amount of behaviors and slowly degrade the model to a more sequential one. Thus instead of going further with this model we changed to a new one as described in next section.

6. Distributed Model

To fully exploit the capabilities of SPIN and to generate a model which would capture most number of behaviors we considered the following model for the pace maker. As shown in figure 11 in this model we do not have any global variables which are shared between the different processes, each process has access to the local variables and only message passing is used to communicate between the processes. There are 3 channels which are used to communicate among the processes, pulse, sense and AVD channel. Each of these processes has their own timer and clock which keeps on incrementing independent of the other process. The only time they meet is when they synchronize the channels and exchange messages.

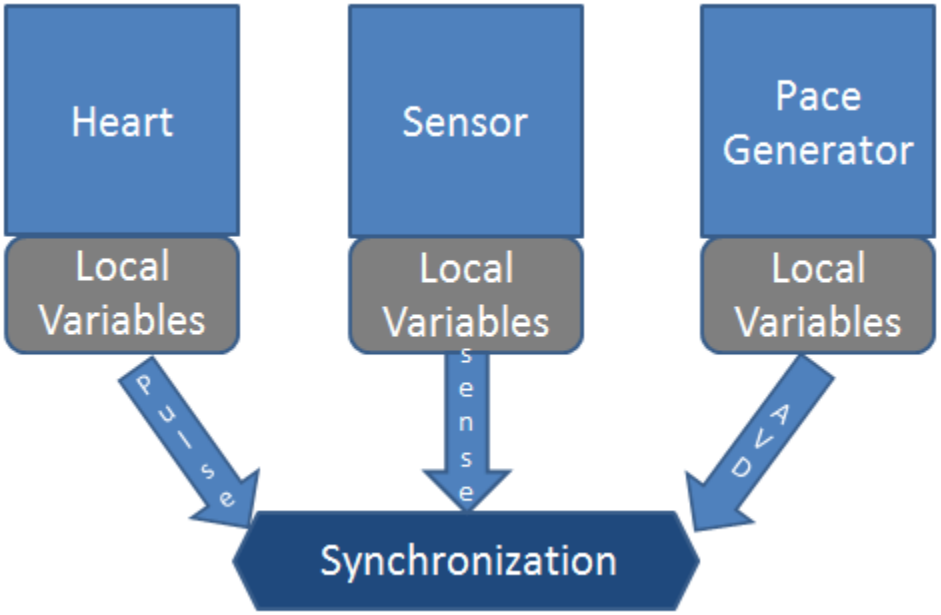


Figure 11 – Distributed Model of Pacemaker

In our current implementation all the channels we have used are synchronous in nature which is required to enforce the behavior of pacing, sensing and to calculate AV Delay. In order to add some more processed and to implement advanced modes like rate controlled pacing and hysteresis pacing we may get an opportunity to use the buffered channels which will allow us to capture more behaviors without restricting too much in the model. This was the problem which we ran into while extending the concurrent model to advanced features of the Pace Generator.

So far we have implemented all the basic modes of operation of the Pace Generator for this model as well. The desired LTL properties stay the same as was the case with the concurrent and the sequential model. In order to use the same LTL properties which are formulae on the global variables we have to introduce certain global variables in this model. But these global variables are not shared between the processes and are only used to specify and validate the LTL properties. In the next section we will discuss the results of verification of this model.

6.1 Verification of Distributed Pacemaker Model

The distributed model is the most flexible and advanced of the all the three models discussed so far. The properties we have verified for the distributed model are as shown in table 4. In addition to the properties we listed in the concurrent model we have one more additional property in the distributed case. Since each of the process has its own independent clock the timing related properties need to be understood in a different way for this model. So the property of AV Delay can be calculated by using the clocks in any of the three processes. But in order to satisfy this property the results should be the same always, since the clocks are synchronized for this. Thus in addition to the AV Delay property we have a new property listed as Distributed AV Delay. This property checks if the AV Delay as calculated by the clocks in the Heart process and the Pace Generator process give the same value. We were able to successfully verify this property for the distributed model.

LTL Property	VOO	AOO	DOO	VVI	AAI	DDI	VVT	AAT	VDD	DDD
Deadlock	X	X	X	X	X	X	X	X	X	X
Pace Limit	X	X	X	X	X	X	X	X	X	X
AV Delay			X			X			X	X
Distributed AV Delay			X			X			X	X

Table 4 – Verification Results of Generic LTL Properties for Distributed Model

In order to compare the different models in terms of their expressiveness and flexibility, in the table 5 we show the number of state transitions that are explored by SPIN in order to verify the same property (Deadlock freeness in this case). As is clear from table 5 the three models show an increasing number of states and thus behaviors specified in them. Thus the distributed model is the most expressive and flexible model among the three. It is possible to make it even more powerful by use of buffered channels for adding advanced features from the requirements like rate controlled pacing and hysteresis pacing.

Model	Sequential	Concurrent	Distributed
Number of States	392,719	35,684,919	125,373,000

Table 5 – Number of states explored by SPIN in each model

7. Conclusion and Future Work

In this report we described the progress made toward the pacemaker challenge, we believe ours is the most complete and comprehensive attempt made so far on this problem. We build a sequential model of the pacemaker in PROMELA covering all the 18 operation modes and verified 10 desired properties of the system using SPIN. Then we generated an implementation from the formal PROMELA model by following mechanical rules. The generated C code was once again validated against the specification by a novel modular verification technique using embedded C code in SPIN. Thus we achieved end to end verification and validation of the all of the operation modes in the pacemaker. Then we explored how to evolve the sequential model into a concurrent and finally in a distributed model. We verified all the generic LTL properties and basic operations modes in the concurrent as well as distributed model. In addition we compared the three models and found the distributed one to be the most powerful and flexible.

Still much works needs to be done; in future work we would want to extend the distributed model to include the advanced modes like rate controlled pacing and hysteresis pacing. We would also need to modify the LTL properties for the distributed setting and validate all the timing constraints in the distributed model. The idea of using embedded C in order to validate an actual implementation in a modular fashion can be further explored. In addition some of the mechanical rules which we used to translate the PROMELA Code to C code can be supported by formal proofs which would ensure that the translation preserves the LTL properties. Other minor improvements can be adding more parameters including the non-timing related like Amplitude Width etc. Support for other features of the Pace Generator like Noise, ATR and Diagnosis mode.

More work needs to be done on the automation for the hardware implementation side as well, ideally one would want to specify the components in a high level modeling tool like SPIN and automatically generate code which can be compiled on Small Device C Compiler directly for the target hardware PIC microcontroller.

8. References

[1] Pacemaker System Specification, Jan3, 2007 Boston Scientific.
[2] Formal Specification of a Cardiac Pacing System, Artur Oliveira Gomes and Marcel Vin´ıcius Medeiros Oliveira, FM 2009.
[3] Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM, H. D. Macedo, P. G. Larsen and J. Fitzgerald, FM 2008.
[4] Modeling and Verification of Safety Critical Systems: A Case Study on Pacemaker, Luu Anh Tuan, Man Chun Zheng and Quan Thanh Tho, ICSSIRI 2010.
[5] From Specification to Implementation: A PROMELA to C Compiler, Siegfried LÖFFLER, Project Report, Ecole Nationale Supérieure des Télécommunications.
[6] Logic Verification of ANSI-C code with SPIN, Gerard J. Holzmann, SPIN 2000.
[7] Model-Driven Software Verification, Gerard J. Holzmann and Rajeev Joshi, SPIN 2004.

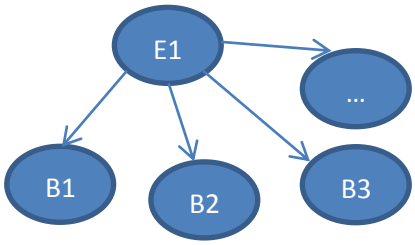
9. Appendix

Preserving LTL Properties during translation from PROMELA to C

In section 4 we just alluded to the rules which we use to convert a given PROMELA model into a corresponding C implementation. These rules were just provided as mechanical translations that were given without any proofs. This required that we validate the resulting C code w.r.t the LTL properties. In this section we provide a necessary condition for the construction of one of the rules which proven to preserve the LTL properties. Consider the first rule which required removing the non-deterministic branching.

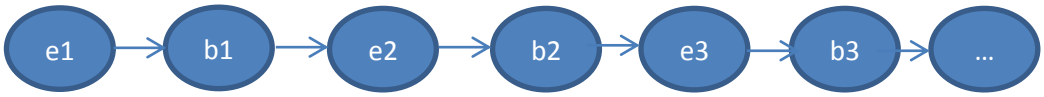
In PROMELA the syntax of a selection statement can be given as follows,

```
if
:: (E1) ->
        :: B1 ;
        :: B2 ;
        :: B3 ;
        :: … ;
fi;
```



Which may be translated into a corresponding list of if statements in C as below,

```
if (e1) b1;
if (e2) b2;
if (e3) b3;
…
```



The figures show the corresponding control flows of the program. LTL formulae in PROMELA are built by using Atomic Propositions (APs) of the global variables in the model. Since while translation from PROMELA to C we preserve the variables we can now express the same LTL formulae using global variables in C (ap).

Theorem

If the forward slice at e1 with respect to the variables in APs does not contain any of the variables of AP themselves then the LTL formula is preserved in translation from PROMELA to C and the same property will hold for the C code written using the corresponding variables from ap.

Proof (Sketch): For the variables in APs of the LTL formula the forward slice will contain all the data and control dependencies. If this slice does not contain any of the other variables from AP that means none of the statements dependent on this variable modify any other APs. Thus the execution trace of the C program would be shuttering equivalent to the transition of states in the PROMELA model. Hence all the LTL properties of the PROMELA model will be preserved while translating into C. □