Feasibility of formal model checking in the Vitatron environment

Wiggelinkhuizen, J.E.

Award date:
2008

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Master's Thesis

Feasibility of formal model checking in the Vitatron environment

J.E. Wiggelinkhuizen

Public version

December 2007

Carried out at:	Medtronic SQDM/Vitatron in Arnhem
Period:	April 2007 - December 2007
Supervisors:	prof.dr.ir. J.F. Groote (TU/e)
	dr.ir. G.J. Tretmans (ESI)
	ing. E. Hendriksen (Medtronic SQDM/Vitatron)

Summary

Implantable medical devices that are developed at Medtronic SQDM/Vitatron (Vitatron in the sequel), such as pacemakers, are highly safety critical. Because errors in these devices could potentially harm humans, their behavior must be absolutely reliable. This behavior is to a large extent determined by the device firmware. This report describes the investigation of the feasibility of applying formal model checking to the design of device firmware in order to verify this design more extensively. For this investigation, the firmware design of Vitatron's DA+ pacemaker is used as a case study.

The model checking tools mCRL2 and Uppaal are considered as a verification environment for formal models that represent a firmware design of Vitatron's devices. mCRL2 is a formal model specification language with accompanying toolset, which contains several analysis and verification tools. mCRL2 is available at [2]. Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems and is available at [3].

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 maintains an adequate rate in the patient's heart by delivering electrical stimuli (paces) to the chambers of the heart. The pacemaker firmware determines when these paces must be delivered through calculations that are based on the timing of incoming contraction events. The firmware must deal with all possible rates and arrhythmias, which makes it a complex composition of collaborating and interacting processes.

From the firmware design, we have translated the most characteristic elements, elements that have the largest influence on the external behavior, and elements with a large flaw risk to a formal model in the model checker languages. Firmware elements with a large flaw risk are elements with a high complexity (e.g. firmware parts consisting of many collaborating and interdependent processes) through which it is hard to evaluate their behavior 'manually' by exploring the firmware design specification. Which specific firmware elements we have modeled is mainly determined through consultations with Vitatron, because of our limited domain knowledge.

To investigate the verification of a pacemaker firmware design by formal model checking, we have exploited several model checking approaches on the created formal models.

The main approach consists of verifying the firmware model in the context of a formal heart model, showing the natural behavior of the human heart for several heart rates and some important arrhythmias, and a formal model of a hardware module that forms the interface between the pacemaker firmware and the patient's heart. In this approach we have verified

three important requirements, which form a representative subset of all requirements to the external behavior of the firmware design.

Unfortunately we were not able to verify the formal model, corresponding to this approach, for a full range of rates that can be shown by the formal heart model. To provide an extensive verification of the firmware model, this was required, but when we allowed the heart model to show high heart rates, the model state space exploded dramatically, thus making verification infeasible. For heart rates in a range from low to 'average', we have successfully verified the corresponding formal model. This resulted in a validation of all three specified requirements to the firmware model.

Another model checking approach has led to the most promising results of our investigation. In this approach we have verified a firmware design part that is responsible for handling one specific arrhythmia, because this part is a complex composition of collaborating processes and because Vitatron had found a deadlock in this part some time ago. The part has been verified by placing it in the context of some driver processes that provide all combinations of input event sequences and external variable values to the part.

We have succeeded to verify this design part and found the known deadlock rather soon. Verification of a design part was very feasible in this case, mainly because the specific part had little dependencies on external variables, through which we were able to verify the design part extensively for all possible variable value combinations. These results show that model checking certainly is a valuable verification technique, although this is mainly the case for limited design parts.

From our investigations, we have drawn some important recommendations to Vitatron. For the specification of future firmware designs, it is recommended to increase the determinism and the clarity of these designs in order to ease the translation to formal models that are suitable for model checking. Furthermore from the successful approach in which a design part was verified, it appeared to be very valuable from model checking perspective, to design according to the 'low coupling/high cohesion' design pattern. Applying model checking onto parts that are designed according to this design pattern can lead to important results.

For the application of model checking in general, we recommend to start the application of the technique already early in the design stage. This will provide early insight and understanding of the consequences of made design decisions and might prevent expensive corrections.

From our comparison between mCRL2 and Uppaal in this project, we recommend the use of mCRL2 as a model checking tool for firmware designs, because of the limited results that we obtained from using Uppaal. However, it is also recommended to further investigate the usability of other available model checking tools.

For the nearby future, it is recommended that Vitatron tries to apply model checking to design components of limited size. It is recommended that the formal models of such design components are created by the firmware designers during the design specification stage, because formally modeling can provide much insight about the consequences of made design decisions. Next to this, it is recommended to state the requirements to the modeled design components also already during the design stage.

Contents

Glossary

atrium	upper chamber of the heart
bpm	beats per minute
cardiac arrhythmias	conditions in which the electrical activity of the heart is not appropriate
counterexample/witnessing trace	sequence of actions leading to a state in which the requirement is violated/satisfied
crosstalk ventricular sense	atrial pace that is detected in the ventricular and wrongly interpreted as a ventricular depolarization
CSPEC	Control SPECification in an H&P system requirements model
deadlock state	state of a formal model from which no transitions are possible
DFD	Data Flow Diagram in an H&P system requirements model
distributed system	system whose functionality is distributed over two or more components
IPG	Implantable Pulse Generator (other term for an artificial pacemaker)
model checking problem	the problem of proving that a requirement holds for a given model
PAT	Process Activation Table which can occur as part of a CSPEC in an H&P system requirements model
PG	Pulse Generator: hardware module in Vitatron's DA+ pacemaker that forms the interface between the pacemaker leads and the pacemaker firmware

PSPEC	Process SPECification in an H&P system requirements model
PVC	Premature Ventricular Contraction (a ventricular contraction which is not the result of an earlier atrial contraction or a stimulus from an artificial pacemaker)
refractory period	period after depolarization in which cell tissue is less or not at all excitable, meaning that it does not react to stimulations
retrograde conduction (RC)	conduction of a stimulus in reverse direction, from the ventricles via the AV node to the atria
system behavior	all possible sequences of actions that a system can perform during its lifetime
ventricle	lower chamber of the heart

Chapter 1

Introduction

Medtronic is the global leader in medical technology. With deep roots in the treatment of heart disease, Medtronic now provides a wide range of products and therapies.

Medtronic Subcutaneous Diagnostics and Monitoring (SQDM) is a recently established organization residing in the Medtronic Arnhem facility, where also the Medtronic Vitatron organization is located (in the sequel we will depict these two organizations by Vitatron). Vitatron focuses on subcutaneous (under the skin) implantable medical monitoring devices, used for diagnostic data collection during a number of years. Before the switch to these implantable monitoring devices, pacemakers were developed at Vitatron.

This project is carried out at the firmware development department. This department is responsible for designing, implementing and testing all firmware located in the pacemakers and implantable monitors based on requirements delivered by the system department. A considerable amount of the development effort consists of testing.

Besides the firmware development department, the Research&Development organization consists of a software group, basically developing Object-Oriented, PC-based software and a hardware development group developing the chip-set used in the pacemakers and implantable monitors. Basically Vitatron works from idea until production preparation for their products.

The firmware forms the intelligent core of the devices that are developed at Vitatron, i.e. it controls the behavior of the device hardware. In a pacemaker the firmware controls the hardware such that an adequate heart rate is maintained in the patient's heart, which is necessary either because the heart's native pacemaker is insufficiently fast, or there is a block in the heart's electrical conduction system. An adequate heart rate is required to ensure that the heart pumps enough blood into the body, according to the patient's activity level. To accomplish this adequate heart rate, the pacemaker is connected to the chambers of the heart by electrodes (pacemaker leads). Via these leads, electrical stimuli (paces) can be delivered to force a contraction of the heart's muscle cells.

The pacemaker also retrieves information from the heart via these leads; contractions of the heart are measured and result in input signals on the pacemaker leads. Together with hardware notifications that indicate a delivered pace, these are the main inputs for the pacemaker firmware, which calculates whether the current heart rate is appropriate on every input signal. These calculations always result in commanding the hardware to deliver a pace to the heart on a certain moment in time. The planned pace will ensure a heart contraction and

prevents that the heart stops pumping blood into the body if it does not contract spontaneously anymore.

Because errors in Vitatron's medical devices could potentially harm humans, testing of the firmware that controls these devices is extremely important. Vitatron spends a lot of effort in testing and verification of their firmware designs and implementations. Currently this is mainly done by reviewing and functional testing, but the coverage of these techniques is not optimal. Therefore it could be the case that certain errors are not discovered. Combining the current test methods with formal methods is perhaps a solution to test the firmware designs more extensively. But at this moment, Vitatron is unaware of the possibilities of formal methods in their environment.

Formal methods are based on solid mathematical principles and increase understanding of systems, increase clarity of descriptions and help solve problems and remove errors. The use of formal methods for software and hardware designs is motivated by the expectation that performing appropriate mathematical analyses can contribute to the reliability and robustness of these designs. However, the high cost of using formal methods means that they are usually only used in the development of safety-critical systems.

Model checking is one of these formal methods that can be used to verify whether a model of a system design satisfies its requirements. Models of hardware designs, software designs and also communication protocols can be verified by means of model checking. The technique is called model checking because it is used to *check* whether a system requirement holds in a formal *model* that represents the verified system design. The great advantage of design verification by using model checking is the completeness of the verification; after validation of a requirement on a formal model of the system design, it is absolutely sure that this model contains no behavior that violates the requirement. Of course such a validation is only useful if the verified model is a valid representation of the system design.

To make model checking feasible, simplifications must be applied when a system design is translated to a formal model. Without these simplifications the model verification generally needs too much time and space resources. Because of this need for simplifications, it generally appears to be very hard to create a formal model that is a valid representation of the system design. Furthermore inventing the right requirements to the system design and correctly formulating them is often not straightforward.

This makes clear that the application of formal model checking to Vitatron's firmware designs is certainly not straightforward. Therefore this project investigates the feasibility of model checking in the Vitatron environment. For this investigation, the firmware design of Vitatron's DA+ pacemaker is used as a case study. We investigate how formal models can be developed that represent (parts of) the firmware design, which requirements to the firmware design (parts) are suitable candidates to be formally verified, and whether the verification of these requirements on the formal models is feasible. In these investigations, we consider two different model checking tools, i.e. mCRL2 and Uppaal.

Developing a formal model that represents a system design, means that the behavior and the responsibilities of all components in the system design and the communications between them are made explicit. As described above, the challenge in formally modeling a system design consists of inventing the right simplifications without letting the formal model deviate too much from the meant behavior in the system design. In our investigation of the

development of formal models representing the pacemaker firmware design, we also address this point of inventing the right simplifications.

The final goal of these investigations is providing recommendations to Vitatron, concerning the applicability of model checking as a minimum. To achieve this final goal, we try to verify requirements on (parts of) the firmware design by means of model checking in both mCRL2 and UPPAAL.

The report is divided into two parts. The first part consists of chapters 2, 3, and 4 and describes the background of the project. Chapter 2 introduces the model checking technique and the used model checking tools, chapter 3 describes some important aspects of the physiology of the human heart, and chapter 4 describes the architecture of the DA+ pacemaker and the design of the pacemaker firmware.

The second part consists of chapter 5, 6, and 7 and describes the investigation that has been carried out during this project. Chapter 5 describes the translation of the pacemaker firmware design to formal models, and chapter 6 describes how model checking is applied on these formal models and which results we have obtained. Chapter 7 describes the recommendations to Vitatron about how they could apply model checking on their firmware designs in the future.

Finally, chapter 8 presents our main conclusions and gives a summary of our recommendations from chapter 7.

Note that the formal models that are produced in this project are not included as appendices to this report for confidentiality reasons. This report is accompanied by a CD on which all relevant formal models can be found.

Furthermore, in this version of the report some parts of the text are removed or changed for confidentiality reasons. These changes all concern pacemaker details which are not crucial for understanding the results of the investigations.

Chapter 2

Model checking

More general information about model checking can be found in [6] and [14].

Many software and hardware systems are composed of separate components that are being executed in parallel and together define the behavior of the system by interacting with each other. The *system behavior* is represented by all possible sequences of actions that a system can perform during its lifetime. Such a system whose functionality is distributed over two or more components is also called a *distributed system*. A component of a distributed system does not contain any concurrency and is described as a sequence of operations on a set of inputs, possibly resulting in one or more outputs.

When designing a large distributed system consisting of several components, it is generally very hard to maintain a clear picture of the overall behavior of the system. The number and the size of all possible executions of a distributed system are exponential in the number of possible input combinations and in the number and the size of the system components. Execution must be understood as the action sequence that is performed by the system in response to one particular set of inputs. It is often impossible to analyze all these executions separately when one wants to check whether the system behavior satisfies its requirements. One possible tactic to verify such systems could be verifying whether the executions that result from the mostly occurring input combinations, are valid. But this generally leaves a rather big number of executions unverified, which could possibly contain an execution that does not satisfy the requirements.

Model checking is a formal method that has been developed for extensively analyzing and verifying complex distributed systems. The model checking method aims at guaranteeing that all executions of the analyzed system satisfy the system requirements, or if these requirements are not met, the method tries to provide information about the circumstances (the particular inputs and system settings) in which the system is not correct.

To provide such an automatic and complete verification method of distributed systems, formal analysis techniques and tools are developed. These techniques require a formal model which is an abstraction of the behavior of the analyzed system. By modeling the system behavior, the responsibilities of the system components and the communication between them are made explicit. Often the system is only partially formalized, because some parts are not relevant for the system requirement that is verified. Each modeled system component translates to one or more concurrent processes in the formal model. Depending on the nature

of the system, one or more concurrent processes can be contained to provide inputs to the processes that represent the system. Figure 2.1 shows how system, formal model and model checking relate to each other.

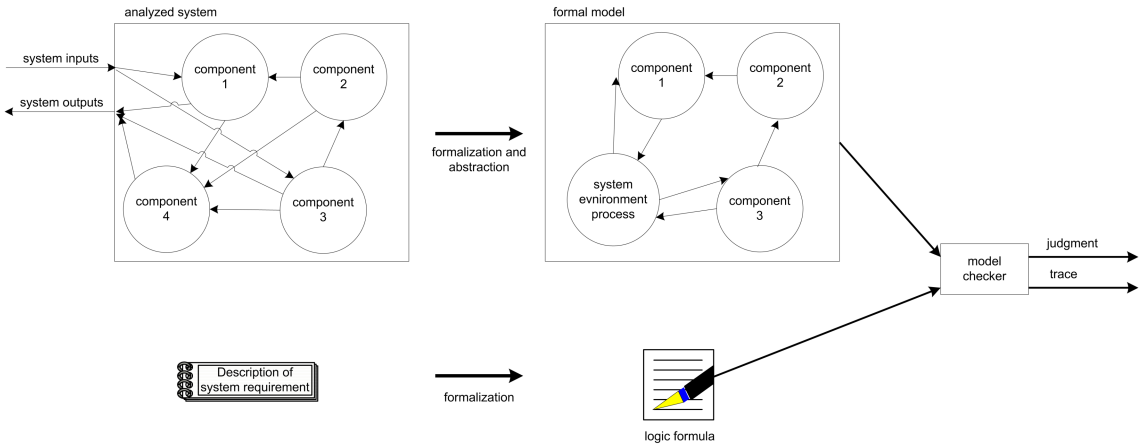


Figure 2.1: The model checking process

Because the modeled system can be very complex, it is generally also difficult to get a valid formal model of the system. Therefore it is wise to use a kind of iterative modeling approach in which the formal analysis techniques are used to find flaws in the model. These flaws can be corrected after which the model can again be analyzed, etc. Eventually this hopefully will result in a valid formal model of the system.

> As a running example for this chapter consider a simple coffee machine system. The system is designed to deliver coffee, tea or water to the user; the user should be able to choose for coffee, tea or water by pushing the button *choose_coffee*, *choose_tea*, or *choose_water* respectively. After choosing a drink, the user can insert money and the machine starts preparing when at least 30 cents are inserted by the user. When the drink has been prepared, the machine serves the drink to the user and returns the money that is paid too much.
>
> The coffee machine system is represented as a state transition diagram in figure 2.2. Unfortunately the required 'choose water' option has not been included in the design, which is reflected by the absence of a *choose_water* transition in the figure. Although the coffee machine system is not a real distributed system, because it consists of only one component, it will be useful to explain the basic notions of model checking.

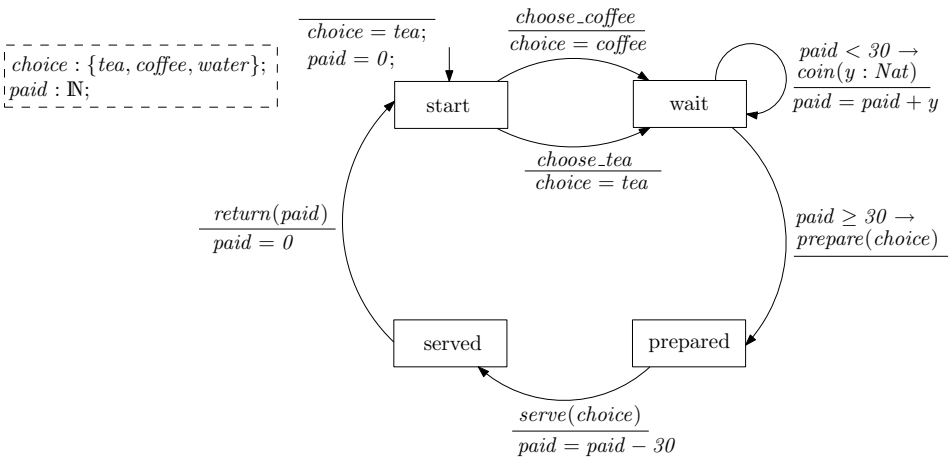


Figure 2.2: Coffee machine system

Because we do not use a standard notation in figure 2.2, a few notation issues must be addressed. In the dashed rectangle the variables *choice* and *paid* are declared that are set when the user chooses a drink or inserts a coin, respectively. A label on a transition may consist of the following fields: a guard followed by an arrow which indicates under which condition this transition may be taken, the action that will cause the transition to be taken, and assignments to variables of the process (below a horizontal line). Each transition must have exactly one action in its label, but there may be zero or multiple guards and variable assignments in a label. Note that a small incoming arc to state 'start' indicates that 'start' is the initial state of the process. This arc also has a label in which the variables *choice* and *paid* are initialized. The action *coin* has an argument y to indicate what value the inserted coin has.

The problem of proving that a requirement holds for a given model is called the *model checking problem*. In our coffee machine example, you want to be sure that the user always gets the drink of his or her choice, provided that enough coins are inserted. If you are verifying a data communication protocol, you want to make sure that a sent message always will eventually reach its recipient and nobody else than the recipient. Such requirements can be expressed in a logic formula, which together with the formal model forms the input to a model checker. The model checker either reports that the requirement holds or does not hold. Depending on the kind of formula and the result of the verification, the model checker can provide a *counterexample trace* or *witnessing trace*, which is a sequence of actions leading to a state in which the requirement is violated or satisfied, respectively. Such a counterexample potentially can provide useful feedback to improve the model and the modeled system itself.

Notice that the described behavior of a model checker is somewhat idealized. Often the available resources are not sufficient to analyze the complete model and then simplifications of the model are needed to make model checking possible. The challenge is to choose the right simplifications to be sure that the analyzed model still represents the behavior of (the modeled part of) the original system. Otherwise a confirmation of the verified requirement is of limited value, because there could be a flaw in a part of the system behavior that is left out due to simplifications.

2.0.1 State space generation

To describe the basic mechanism of a model checker, we first introduce the notion of the model state space. Depending on all previously performed actions the model of the analyzed system resides in a certain state which can be seen as a vector of the values of all component variables and the states of the individual components. For example, the coffee machine system from our example can be straightforwardly translated to a formal model in which the states are depicted by (*state*, *choice*, *paid*) vectors. In such a formal model, the initial state is given by (start,*tea*,0) and performing action *choose*(*coffee*) in the initial state will move the model to state (wait,*coffee*,0).

Each model has an associated state space, consisting of all states that are reachable from the model's initial state. Under the assumption that all states of the formal model of the coffee machine system are reachable from its initial state, the state space of this model would be infinitely large, because the variable *paid* is a natural number. In figure 2.3 a schematic overview of all combinatorial possibilities is given.

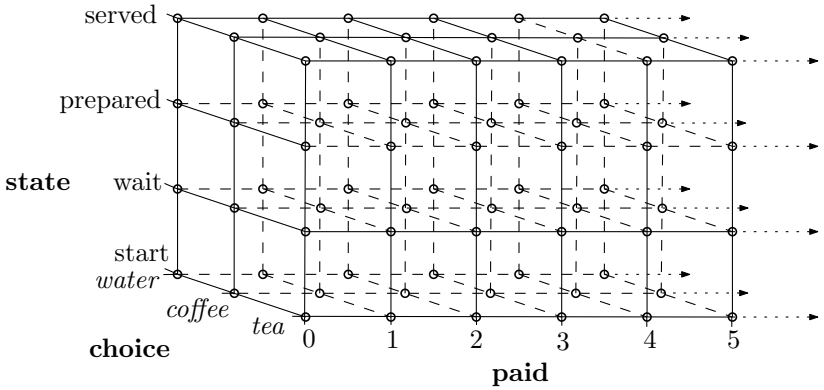


Figure 2.3: Possible states of the coffee machine system

Though in reality there are limitations: the values of the coins that can be inserted are discrete; assume for now that only coins of 5, 10 and 20 cents are accepted by the coffee machine. Furthermore, the system's state transition diagram of figure 2.2 makes clear that it is not possible to insert any more coins when already at least 30 cents have been inserted and that the user can not choose for water. We shall see that these limitations make the actual state space of the coffee machine model finite.

A model checker generates the state space in order to make requirement verification possible. For this generation an algorithm is used that explores the state space from the initial state until it no longer finds new states. The precise order in which the states are explored depends on the used algorithm; this can for instance be a breadth-first search, a depth-first search or a random search algorithm. Here we only address the breadth-first search algorithm; this algorithm starts by exploring which states are reachable from the initial state by taking all possible transitions out of the initial state. The newly discovered states are added to the state space together with the transitions leading to them. The added states are said to be on level 1 of the state space, because they are reachable in one step from the initial state. When all discovered states are added to level 1, the initial state is said to be 'explored'. In the second iteration the algorithm explores which states are reachable from the states on level 1; any newly found state is added to the state space again (on level 2) together with the

transition leading to it, except when a found state is already present in the state space only the transition leading to it is added. If all states of level 1 are addressed by this procedure, the second iteration is finished and the algorithm iterates by exploring the states on level 2. The iteration stops if exploration of the states on a certain level does not discover new states, which means that all states in the state space have been explored after the exploration of this level.

The state space generation results in a labeled transition system in which all transitions represent one single atomic action. For the coffee machine example, the state space is shown in figure 2.4. Note that the initial state is again marked with a small incoming arc. However there is no action on this arc, because variable initialization is made implicit by entering state (start,*tea*,0).

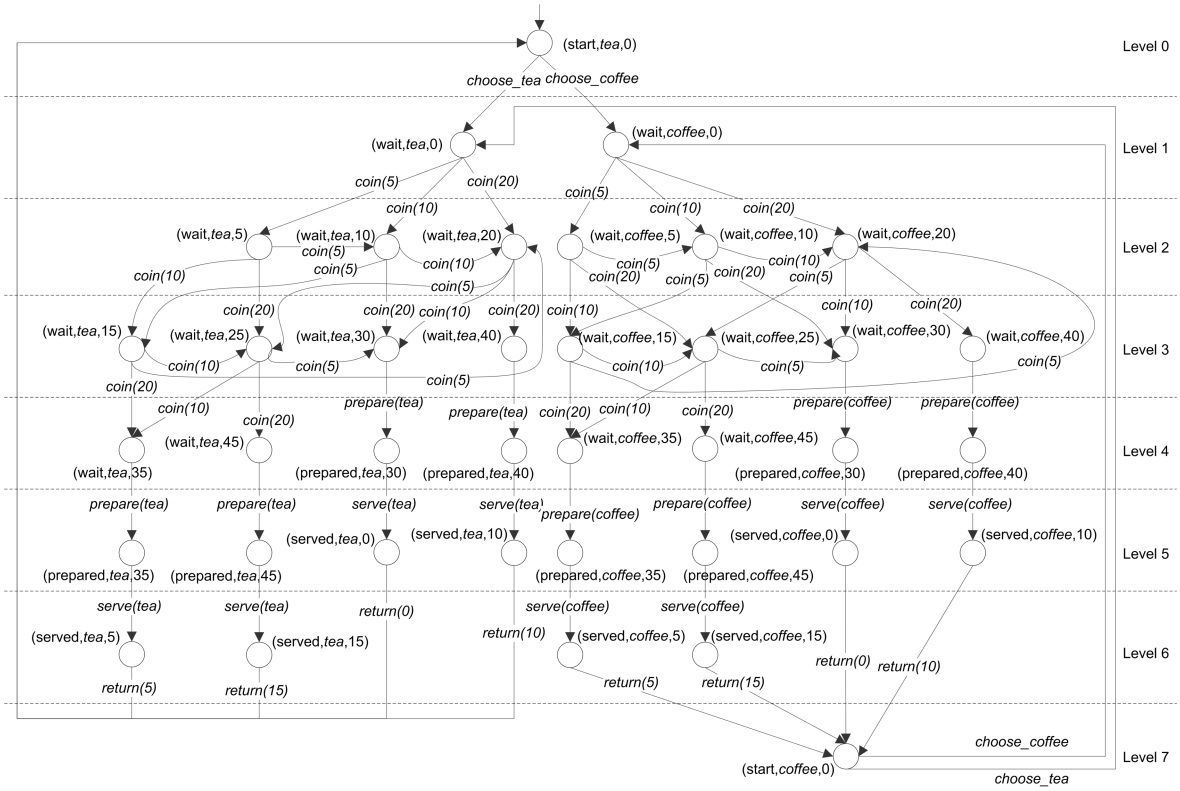


Figure 2.4: State space of the coffee machine system

Although the state space of the coffee machine model is finite, its size can be even more reduced by improving the system's state transition diagram of figure 2.2. The money that is paid too much is returned after serving the requested drink through which there are eight states in the state space for the machine state 'served'. If the too much paid money should be returned before serving the drink there would be only two states (one for each possible drink) for 'served' in the resulting state space. This is a small example of how the architecture of the modeled system influences the size of the state space.

The generated state space is explored by the model checker in order to verify the given requirement to the system. Basically it traverses all states consecutively and continuously checks whether the current state satisfies the requirement. Sometimes it is not required to

traverse all states, but this depends on which kind of property is verified. For example, if it is questioned whether action a is ever performed in the state space, the verification can stop on the detection of one single a action, which possibly can be very soon. But when the state space does not contain any a action the complete state space must be explored before the algorithm can give its judgment.

2.0.2 Formulating model requirements

As already noted, a system requirement must be formulated in some kind of logic before a model checker can verify it. Propositional formulas over state variables are the most basic formulas. In such formulas, it can for instance be specified that a certain variable is at most 10 in the entire state space. Presuming that the user of our coffee machine is not able to choose water can be the reason that we want to verify whether the variable *choice* ever gets the value *water* in the state space of the coffee machine model. The model checker then traverses all states to check whether $choice = water$ and will report that there is no state that satisfies the specified property.

Clearly propositional logic is not expressive enough, because we can not specify anything about actions that are performed by the modeled system. Therefore there are more expressive logics which can be classified into non-temporal and temporal ones; temporal logics are the more expressive of the two types. Which specific logics can be used in model checking depends on the used tool, but generally all model checkers accept formulas in propositional, non-temporal, and temporal logics. We give some examples in temporal and non-temporal logic in order to indicate their differences. It is important to notice that the initial state is the reference point of all formulas in temporal and non-temporal logics.

For explaining non-temporal logics and giving examples for these logics we use the Hennessy-Milner logic ([13]; also introduced in chapter 4 of [11]). Its syntax is given by the following grammar:

$$\phi ::= true \mid false \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi.$$

The primitive formula *true* is valid in all states of a model and *false* is never valid in any state. The logical operators have their usual meaning and can be used to compose logic formulas. The two special constructs of the Hennessy-Milner logic are the diamond modality and the box modality. The diamond modality $\langle a \rangle \phi$ expresses that there is a transition labeled with action a and leading to a state in which the formula ϕ holds. The box modality $[a]\phi$ expresses that all transitions labeled with action a lead to a state in which formula ϕ holds.

Now if we want to verify whether the user can choose coffee in our coffee machine system we formulate a formula as follows: $\langle choose\_coffee \rangle true$. The formula $[choose\_water]false$ expresses that whenever an action *choose_water* is done, a state is reached where *false* is valid. Because *false* is not valid in any state, the formula expresses that an action *choose_water* is not possible. The formula $\langle choose\_coffee \rangle \langle coin(20) \rangle \langle prepare(coffee) \rangle true$ expresses that the action sequence *choose_coffee*, $coin(20)$, $prepare(coffee)$ is possible from the initial state, which is clearly not valid, because the machine can not start preparing coffee before enough money is inserted.

As mentioned above, all logic formulas are evaluated in the initial state of the system. Because non-temporal logics do not provide the ability to specify an arbitrary action sequence, we can not specify that a certain requirement must hold during the entire lifetime of the system. Should the coffee machine model contain a flaw such that the coffee machine runs into a deadlock after serving the first drink in its lifetime, the truth values of the above

mentioned formulas will remain the same, although the model behavior certainly does not satisfy its requirements. You want to make sure that the coffee machine operates as required during its entire lifetime. For this purpose, the property must be specified in a temporal logic. For example, the property "after choosing coffee the machine *always eventually* serves my coffee" is not expressible in non-temporal logics, though it is expressible in temporal logics.

As the word temporal already says, a temporal logic deals with 'time', where performing actions must be understood as passage of 'time'. In a temporal logic it is possible to express that a formula is *always* valid, *eventually* valid or valid *until* some other formula is valid. We give some examples in temporal logic, using the same Hennessy-Milner logic extended with regular expressions. In this extension more than just a single action is allowed in a diamond or box modality. See [11] for a description of this extension. With this extension it is possible to formulate the above property. We give the corresponding logic formula for the interested reader:

$$[true^* \cdot choose\_coffee]\langle true^* \cdot serve(coffee)\rangle true.$$

Literally this formula says that if after a certain sequence of zero or more arbitrary actions (expressed by $true^*$) an action *choose_coffee* is possible, then after that an action $serve(coffee)$ is possible after a certain sequence of zero or more arbitrary actions. A stronger property that can be formulated in temporal logic would be "after choosing coffee the machine will not process any other request before serving my coffee". This is formulated as follows:

$$[true^* \cdot choose\_coffee]\langle \overline{choose\_coffee \cup choose\_tea \cup choose\_water}^* \cdot serve(coffee)\rangle true.$$

Literally this formula says that if after a certain sequence of zero or more arbitrary actions an action *choose_coffee* is possible, then after that an action $serve(coffee)$ is possible after a certain sequence of actions, not containing *choose* actions.

Sometimes the nature of the model makes it hard to translate a requirement into a logical formula. Then an alternative option is to add an action to the model which can only be performed if the requirement is not satisfied. Assume that we want to verify that the variable *paid* in the coffee machine model never becomes greater than 50 in state 'wait'; then we add a loop transition (a transition from a state to itself) to 'wait' with a guard '$paid > 50$' that performs the action *alarm*. Verifying the requirement then is a matter of specifying in a simple logic formula that the action *alarm* can be performed after zero or more arbitrary actions: $\langle true^* \cdot alarm\rangle true$. If the model checker reports that the formula is true, it is guaranteed that the requirement is not true; the model checker might provide a *witnessing trace* which specifies an action sequence that leads to the performance of action *alarm*. This trace can be inspected to find the flaw in the system. On the other hand, if the model checker reports that the formula is false, the model is guaranteed to satisfy the requirement. Some model checkers also can be instructed to detect certain actions during state space generation; then a message is printed to the screen or to an output file for each such action that is detected during state space generation. Whether this method is suitable heavily depends on the model and the requirement in question.

2.0.3 Model checking tools

As already said, there are various model checking tools available. Some of the most known tools are SMV, SPIN, CADP, mCRL2, and Uppaal. In this project mCRL2 and Uppaal are

investigated as a model checking environment for the formal models representing the firmware designs of Vitatron. In sections 2.1 and 2.2 these tools are introduced.

2.0.4 The state space explosion problem

The generated state space can become very large for many real-world problems. This is a well-known problem in the field of model checking and is called the state space explosion problem. Not only the time and space resources to generate such large state spaces can be a problem, but also the model checking on a large state space is a difficult job even for the most efficient algorithms.

Most model checkers provide techniques to enable model checking without generating the complete state space. Such techniques can have drastic influence on the time and space required for model checking and can be of various kinds:

- "on-the-fly"-verification of properties during state space generation. In some cases it is not even necessary to explore the complete state space when verifying a property; this happens when a witness or, on the contrary, a counterexample is found during the process of verification. This technique is used by the `pbes2bool` tool in mCRL2 (see section 2.1.3) and also in UPPAAL (see section 2.2). Also SPIN and CADP are "on-the-fly"model checking tools.

- Symbolic model checking; algorithms for symbolic model checking avoid ever generating and storing the state space. Instead, they generally represent the state space using a formula in propositional logic. On such a formula, properties can be verified purely by manipulations of the formula. See [16] for more information about symbolic model checking. Recently also the use of resolution provers has become popular in the field of model checking. Symbolic model checking forms the basis of the tool SMV and is also used by the model checking tools in the mCRL2 toolset (see section 2.1.3).

- Partial order reduction or τ-confluence can be used to reduce the number of independent executions of concurrent processes that need to be considered. The basic idea is that if it does not matter for the property that you want to check whether a or b is executed first, then it is useless to consider both the ab and the ba executions.

- Abstraction; when using this technique, you first simplify the model by abstracting from the internal behavior of the model that is not relevant for the property that is verified. The simplified system usually does not satisfy exactly the same properties as the original one, but generally, it is required that the abstraction is sound (properties that are proved on the abstraction are true of the original system). However, most often, the abstraction is not complete (not all true properties of the original system are true of the abstraction).

2.1 mCRL2

mCRL2 is a formal model specification language with an associated toolset. The toolset is available at [2]. mCRL2 stands for milli Common Representation Language 2 and is the successor of μCRL. The language and its toolset are developed at the department of Mathematics and Computer Science of the TU/e, in collaboration with LaQuSo and CWI.

This section introduces the main concepts of mCRL2, heavily based on [9] and [10]. For more information see these papers.

The mCRL2 language can be used to specify formal models, representing the behavior of distributed systems and protocols; using its accompanying toolset models can be analyzed and verified automatically. The 'about'-section of [2] contains a good description of the mCRL2 philosophy:

> "mCRL2 is based on the Algebra of Communicating Processes (ACP) which is extended to include data and time. Like in every process algebra, a fundamental concept in mCRL2 is the process. Processes can perform actions and can be composed to form new processes using algebraic operators. A system usually consists of several processes (or components) in parallel.
> A process can carry data as its parameters. The state of a process is a specific combination of parameter values. This state may influence the possible actions that the process can perform. In turn, the execution of an action may result in a state change. Every process has a corresponding state space or Labeled Transition System (LTS) which contains all states that the process can reach, along with the possible transitions between those states.
> Using the algebraic operators, very complex processes can be constructed containing, for example, lots of parallelism. A central notion in mCRL2 is the linear process. This is a process from which all parallelism has been removed to produce a series of condition - action - effect rules. Complex systems, consisting of hundreds or even thousands of processes, can be translated to a single linear process. Even for systems with an infinite state space, the linear process (being an abstract representation of that state space) is finite and can often be obtained very easily. Therefore, most tools in the mCRL2 toolset operate on linear processes rather than on state spaces."

It should be noticed that the mentioned built-in concept of time is somewhat premature and therefore this concept is not used in this project. Instead, the passing of time is explicitly modeled in all time-dependent processes. The mCRL2 model description in section 5.2 comes back to this point.

2.1.1 The process specification language

This section describes the mCRL2 language followed by the mCRL2 specification of the coffee machine model, which might clarify how the contents of the language relate to each other.

Processes and actions are the fundamental concepts in the mCRL2 language. A process in mCRL2 describes the behavior of a component in the modeled distributed system. Actions represent atomic events, which possibly can be synchronized with other actions. When two actions are declared synchronous, the actions can be performed by two different processes truly in parallel. By this synchronization mechanism communication and interaction between processes can be modeled.

Processes are defined by *process expressions*, which are compositions of actions using a number of operators. The basic operators are as follows (let a and b be arbitrary actions and let p and q be arbitrary process expressions):

- Deadlock or inaction δ, which does not display any behavior.

- Alternative composition, written as $p + q$. This represents a non-deterministic choice between p and q.
- Sequential composition, written as $p \cdot q$. This expression first executes p and then q (assuming p terminates).
- Recursion describes iterating behavior of a process. A simple example of recursion is the process expression $X = a \cdot X + b$. This process describes a sequence of a actions of arbitrary length (as long as the first alternative is chosen) followed by a single b action.
- Parallel composition or merge, written as $p \parallel q$, which interleaves and synchronizes the actions of p with those of q.
- Synchronization operator $p|q$, which synchronizes the first actions of p and q and combines the rest of p and q like the parallel composition.
- Multiaction $a|\ldots|b$, which is a special instance of the synchronization operator where the arguments are single actions (or multiactions). The meaning of a multiaction is that all actions occurring in it happen at the same moment (i.e. truly in parallel).

Furthermore there are some operators to restrict the behavior of process expressions in order to model the interaction between processes. The restriction operator(∇) and blocking operator(∂) can be used to specify explicitly which multiactions are allowed to occur or are not allowed to occur, respectively. With the communication operator(Γ) a set of allowed communications can be defined, for example a communication $a_0|\ldots|a_n \rightarrow c$ means that every group of actions $a_0, \ldots, a_n$ within a multiaction is replaced by c (this is expressed by $\Gamma_{\{a_0|\ldots|a_n \rightarrow c\}}(p)$ where p is the process expression on which the communication operator is applied).

When we for instance want to model communication from process X to process Y, we declare three actions *send*, *receive*, and *communicate* and we define the process expressions of X and Y as follows: $X = send$ and $Y = receive$. Then we use the communication operator to specify that the *send* action and the *receive* action synchronize to the *communicate* action ($send|receive \rightarrow communicate$). Because we do not want X to send without Y receiving and vice versa, we use the restriction operator to specify that only the *communicate* action can occur. This combination of the synchronization and restriction operators is often used in modeling communicating processes.

Before we move on to the data aspects of the language, the notion of abstraction has to be mentioned in the context of mCRL2. Usually the requirements of a system are defined in terms of *external* behavior (i.e. the interactions of the system with its environment), while one wishes to check these requirements on an implementation of the system which also contains *internal* behavior (i.e. the interaction between the components of the system). So it is desirable to be able to abstract from the internal behavior of the implementation. This is possible 'hiding' internal actions in a mCRL2 specification. This abstraction can be useful in reducing state space sizes. When internal actions are hidden and the state space is generated, reduction techniques based on bisimulation equivalence can be applied to the state space. More information can be found in [11].

Data

There are some standard data sorts and functions included in mCRL2. Unbounded positive ($\mathbb{N}^+$), natural ($\mathbb{N}$), integer ($\mathbb{Z}$) and real numbers ($\mathbb{R}$) are included with all relational and arithmetic operators. Note that real numbers are only used in the time functionality, which will not be covered here. Booleans ($\mathbb{B}$) are included with all elementary operators. Boolean expressions can be used in *conditions*, which are used together with the conditional operator, written as $c \to p \diamond q$, where c is a boolean data expression. This process expression behaves as an if-then-else construct: take for example the process expression $X = (n \geq 10) \to a \diamond b$, which expresses that action a is performed for n at least 10, otherwise action b is performed. Note that the else-branch is optional; $c \to p$ is a valid expression which expresses that p is only performed if c holds.

Besides these standard data sorts it is possible to declare custom sorts together with functions on these sorts. This can be done by declaring constructors, functions and their definitions manually or by using one of the type constructors. There is a type constructor for structured types, for list types, for sets, and finally one for bags. These type constructors automatically provide some standard functions to the type, for example comparing functions.

Data can be used to parameterize actions and processes. As an example, actions can be declared as follows:

act a;
$b, c, d : \mathbb{N}$;
$e : \mathbb{B} \times \mathbb{N}^+$;

This declares parameterless action a, actions b, c and d with a data parameter of sort $\mathbb{N}$, and action e with two parameters of sort $\mathbb{B}$ and $\mathbb{N}^+$, respectively. For the above declaration, a, $b(0)$ and $e(\mathit{false}, 6)$ are valid actions. For all operators, except the communication operator, the data parameterization doesn't change much; the data parameters are just retained when an operator is applied. The communication operator $\Gamma_C(p)$ has become stricter: for each communication $a_0|\ldots|a_n \to c, n \geq 1$, multiactions $a_0(\ldots)|\ldots|a_n(\ldots)$ in p are only replaced by $c(\ldots)$ when the data parameters of all a_i are equal (both the number of parameters and their values). The data parameters are retained in action c. For example $\Gamma_{\{b|c\to d\}}(b(0)|c(0)) = d(0)$, but also $\Gamma_{\{b|c\to d\}}(b(0)|c(1)) = b(0)|c(1)$. Furthermore $\Gamma_{\{b|c\to d\}}(b(1)|b(0)|c(1)) = b(0)|d(1)$.

Processes can be parameterized as follows:

proc $P(d : \mathbb{B}, e : \mathbb{N}^+) = a \cdot P(d, e)$
$+ b(d) \cdot P(\neg d, e + 1)$
$+ c(d, e) \cdot P(\mathit{false}, \mathit{max}(e - 1, 1))$;

This declares the process P with data parameters d and e of sort $\mathbb{B}$ and $\mathbb{N}^+$, respectively. Note that the data parameters can be seen as local variables in the process, which are declared in the left-hand side of the equation. In the process references on the right-hand side it is specified how the values of the data parameters are changed when the corresponding action(s) in front of it are performed.

There is also the ability to quantify over (possibly infinite) data types in process expressions. This is provided through the *summation* operator $\sum_{d:D} p$ where p is a process expression in

which data variable d may occur. The corresponding behavior is $p[d_0/d] + \cdots + p[d_n/d] + \cdots$, $n \geq 0$, for all elements $d_i \in D$. Here, $p[d_i/d]$ stands for p in which each free occurrence of d (i.e. not bound by another $\sum_{d:D}$) is replaced by d_i.

Summations over a data type are particularly useful to model the receipt of an arbitrary element of a data type. For example the following process is a description of a single-place buffer, repeatedly reading a natural number using action name r, and then delivering that value via action name s.

act $r, s : \mathbb{N}$;
proc $\mathit{Buffer} = \sum_{n:\mathbb{N}} r(n) \cdot s(n) \cdot \mathit{Buffer}$;

After this overview of the mCRL2 process specification language, we give an mCRL2 model of our coffee machine system. This model includes a process that represents the machine and a process that represents a user of the machine, which communicate with each other. As all mCRL2 models this model is structured as follows: first the custom sorts and all used actions are declared, next the process expressions that define the machine process (line 13-32) and the user process (line 34-49) are given and finally the initial state of the model is defined. Note that alternative composition (+) is used in combination with conditions to specify all transitions of the state transition diagram in figure 2.2 in process *Machine*. The communication and restriction operator are used in the initial state definition to specify the possible communications between the processes. In the model we have given comments on some lines by stating them after the comment operator %.

```
1.  % line 2-4 declare three structured data types
2.  sort      sort_choice: struct tea | coffee | water;
3.            machine_state: struct start | wait | prepared | served;
4.            user_state: struct ready | chosen;
5.  act       r_choose_tea, s_choose_tea, c_choose_tea;
6.            r_choose_coffee, s_choose_coffee, c_choose_coffee;
7.            r_choose_water, s_choose_water, c_choose_water;
8.            r_coin, s_coin, c_coin : Nat;
9.            prepare : sort_choice;
10.           r_serve, s_serve, c_serve : sort_choice;
11.           r_return, s_return, c_return : Nat;
12.
13. proc      Machine(state : machine_state, choice : sort_choice, paid : Nat) =
14.               % line 15-19 define the behavior of Machine if state equals start
15.               (state ≈ start) →
16.               (   % line 17: receive of choose_coffee changes state to (wait, coffee, paid)
17.                   r_choose_coffee · Machine(wait, coffee, paid)+
18.                   r_choose_tea · Machine(wait, tea, paid)
19.               )+
20.               (state ≈ wait) →
21.               (   (paid < 30) →
22.                       % accept receive of coin with any natural valued parameter y
23.                       % such a message changes state to (wait, choice, paid + y)
24.                       Σ_{y:Nat} r_coin(y) · Machine(wait, choice, paid + y)+
25.                   ◇
26.                       % paid at least 30, so perform prepare action
27.                       prepare(choice) · Machine(prepared, choice, paid)
```

```
28.             )+
29.             (state ≈ prepared) →
30.                 s_serve(choice) · Machine(served, choice, paid − 30);
31.             (state ≈ served) →
32.                 s_return(paid) · Machine(start, choice, 0);
33.
34. proc    User(state : user_state) =
35.             (state ≈ ready) →
36.             (   s_choose_coffee · User(chosen_coffee)+
37.                 s_choose_tea · User(chosen_tea)+
38.                 s_choose_water · User(chosen_water)
39.             )+
40.             (state ≈ chosen_coffee ∨ state ≈ chosen_tea ∨ state ≈ chosen_water) →
41.             (   % drink is chosen, perform s_coin action for a possible coin value
42.                 % or perform r_serve action with any parameter of sort sort_choice,
43.                 % followed by r_return action with any natural valued parameter
44.                 s_coin(5) · User(state)+
45.                 s_coin(10) · User(state)+
46.                 s_coin(20) · User(state)+
47.                 ∑_{drink:sort_choice} r_serve(drink) · ∑_{remainder:ℕ} r_return(remainder)·
48.                 User(ready)
49.             );
50.
51. % line 52-65 define the initial state of the complete model
52. init    % line 53 uses the restriction operator to allow only c_ actions and prepare
53.         ∇_{c_choose_tea,c_choose_coffee,c_choose_water,c_coin,prepare,c_serve,c_return}(
54.         % line 55-60 uses the communication operator to specify the communications
55.         Γ_{r_choose_tea|s_choose_tea→c_choose_tea,
56.         r_choose_coffee|s_choose_coffee→c_choose_coffee,
57.         r_choose_water|s_choose_water→c_choose_water,
58.         r_coin|s_coin→c_coin,
59.         r_serve|s_serve→c_serve,
60.         r_return|s_return→c_return}(
61.             % line 63 and 64 give the initial states of the individual processes
62.             % parallel composition (||) is used to place the processes in parallel
63.             Machine(start, tea, 0)||
64.             User(ready)
65.         ));
```

2.1.2 Modal μ-calculus

The mCRL2 model checking tools accept properties that are formulated in the modal μ-calculus, which is an extension to the Hennessy-Milner logic as described in section 2.0.2. Although the Hennessy-Milner logic with regular expressions is already very expressive and most behavioral properties can be specified in this logic, the modal μ-calculus is even more expressive through the addition of minimal and maximal fixed point operators. However, this addition also makes it rather difficult to formulate correct properties in the modal μ-calculus. Note that all formulas in the Hennessy-Milner logic can also be used in mCRL2, because this logic is a true subset of the modal μ-calculus. We give a brief introduction of the modal μ-calculus for the interested reader.

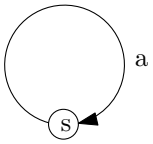
The syntax of the modal μ-calculus is as follows:

$$\phi ::= \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \langle a \rangle \phi \mid [a]\phi \mid \mu X.\phi \mid \nu X.\phi \mid X.$$

The formula $\mu X.\phi$ is the minimal fixed point and $\nu X.\phi$ is the maximal fixed point. X is just a variables which is typically used in fixed points together with other capitals Y,Z,... A fixed point $\mu X.\phi$ or $\nu X.\phi$ only exists if X occurs positively in ϕ, meaning that X in ϕ must be preceded by an even number of negations. For counting negations $\phi_1 \rightarrow \phi_2$ should be read as $\neg\phi_1 \vee \phi_2$. So, for instance $\nu X.\neg X$ and $\nu X.\neg([a]\neg X \vee X)$ are not allowed; in the first formula X occurs with one preceding negation, in the second formula only the former occurrence of X is fine because it is preceded by two negations.

To understand fixed point modalities it is useful to consider X as a set of states. Formula $\mu X.\phi$ is valid for all states in the smallest set X that satisfies the equation $X = \phi$, where X generally occurs in ϕ. Here we abuse notation, by thinking of ϕ as the set of states where ϕ is valid. Similarly, $\nu X.\phi$ is valid for the states in the largest set X that satisfies $X = \phi$. As an example we can look at two simple fixed point formulas $\mu X.X$ and $\nu X.X$. Because $\phi = X$, we are interested in the smallest and largest set of states X that satisfies the equation $X = X$, respectively. Of course every set satisfies this equation. Hence, the smallest set to satisfy it, is the empty set. This means that $\mu X.X$ is not valid in any state, which is equivalent to saying that $\mu X.X = \mathit{false}$. The largest set X that satisfies $X = X$ is the set of all states, so $\nu X.X$ is valid in all states which means that $\nu X.X = \mathit{true}$.

As another example we look whether the formulas $\mu X.\langle a \rangle X$ and $\nu X.\langle a \rangle X$ are valid in the following transition system:



The only sets of states to be considered are the empty set $X = \emptyset$ and the set of all states $X = \{s\}$. Both satisfy the 'equation' $X = \langle a \rangle X$. Namely, if $X = \emptyset$, then the equation reduces to $\mathit{false} = \langle a \rangle \mathit{false}$, which is valid. If $X = \{s\}$ it is also clear that this equation holds. So, $\mu X.\langle a \rangle X$ is valid for all states in the empty set. Hence, this formula is not valid in s. However, $\nu X.\langle a \rangle X$ is valid for all states in the largest set, being $\{s\}$ in this case. So, $\nu X.\langle a \rangle X$ is valid.

This example concludes our small introduction to modal μ-calculus. For more details, see [11].

2.1.3 mCRL2 toolset

Now we have seen the basics of both the mCRL2 process specification language and the property language modal μ-calculus. Using these two languages as recipes one could specify a formal model of the considered system and a property of the system that must be verified by means of model checking. Model checking is one of the purposes of the mCRL2 toolset, but besides that the toolset can be also be used to analyze and explore the model by simulation and visualization of the model's state space. Furthermore the toolset contains tools to optimize the model automatically. It should be noted that the toolset is subject to continuously development; here we give a description of the toolset version (SVN revision 3626) that we have used in this project.

Figure 2.5 gives an overview of the use of the mCRL2 toolset. On the left the figure shows the inputs of the toolset, which are a model of a system expressed in the mCRL2 language and possibly a formula that must be verified. The first step in model analysis is always the generation of a Linearized Process Specification (LPS) for the model by using the linearizer tool `mcrl22lps`. This tool first performs a syntax check and then starts the linearization, which can be seen as compiling the model to a more strict format that is well suited for analysis and manipulation by the tools in the toolset. An LPS can be seen as a symbolic representation of the transition system of a model. For more information about linearizing mCRL2 process specifications, see [10].

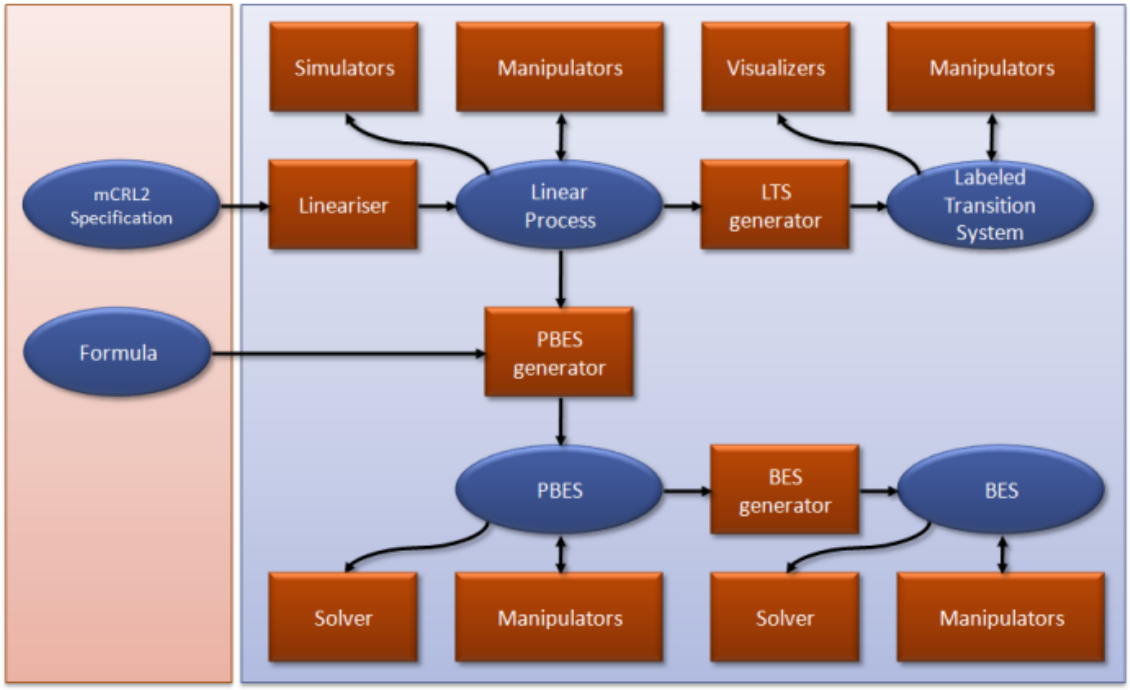


Figure 2.5: Overview of the mCRL2 toolset (picture taken from [2])

An LPS is unreadable for humans, but sometimes it is needed to inspect the LPS. For this purpose the toolset contains a tool to pretty print the LPS and a tool which provides some basic information about the LPS (these tools are not present in figure 2.5). This basic information contains for example the number of summands in the LPS, which is often indicative for the size of the model's state space. The LPS can be optimized by some manipulator tools; there is a tool for removing process parameters which are constant in the (reachable) state space, a tool for removing unused process parameters, and a rewriting tool. This rewriting tool evaluates conditions in the LPS by rewriting and then decides which alternatives are unused and can be removed. These optimization tools can have a substantial effect on the final size of the state space and/or the resources needed for the generation of the state space.

Besides these manipulator tools there is a simulator tool available which can be used to get more insight in a model without generating its state space. The user explores the model with this tool by manually performing individual (multi)actions. After an action has been performed the simulator shows a representation of the new state (a state vector), and the actions that are possible from this state. The simulator also offers the ability to save action

traces and load them again on a future moment.

To generate the state space from an LPS the tool `lps2lts` can be used. This tool generates a Labeled Transition System (LTS) that represents the complete behavior of the model. On the LTS the manipulator tool `ltsconvert` can be used to convert between different formats of LTSs and to minimize the state space modulo (branching) bisimulation equivalence. This minimization is effective when several internal actions are hidden (abstraction) in the model. For example, if the state space contains a sequence of internal actions, this sequence can be collapsed in many cases. Then all intermediate states in the sequence are removed, which leads to a smaller state space. Furthermore there are several tools to visualize the state space, which again can help in finding specification errors or in getting more insight in the model.

When analysis and simulation show that the formal model conforms to the modeled system well enough, one can start model checking. Model checking in mCRL2 is provided using Parameterized Boolean Equation Systems (PBES), which can be generated by the tool `lps2pbes` from an LPS and a modal μ-calculus formula. For PBESs there is again a tool to print it in human readable format and using `pbes2bes` one can try to remove data from the PBES in order to get a BES, which is often easier to solve. The actual model checking can be done by the tool `pbes2bool`, which internally uses `pbes2bes` and then solves the BES by rewriting the boolean equations in the BES. If requested, the tool returns a trace that indicates how `pbes2bool` came to the validity or invalidity of the PBES. Such a trace can again be useful to improve the model or the modeled system itself.

This description of the mCRL2 toolset concludes the introduction to mCRL2. In later chapters we will see how mCRL2 can be useful in formal modeling the pacemaker firmware design and in verification of the resulting models.

2.2 Uppaal

Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems. The tool has been developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark; the tool is available at [3]. In this project, we have used version 4.0.6 of Uppaal. Based on [4], this section gives an introduction to Uppaal.

In Uppaal it is possible to model the behavior of real-time systems as concurrent processes via a (Java-based) graphical interface. The tool contains a graphical simulator which constantly shows the actual state of the model and enables the user to explore the possible executions of the model by manually performing actions. During the design or modeling of the system this can provide useful information to improve the model or the modeled system itself. Then there is the model checker which covers the complete behavior of the system by exploring all possible executions. The model checker can check invariant properties (properties that must hold in every reachable state of the model) and reachability properties (properties that must hold in at least one reachable state of the model) that are formulated in (temporal) logic.

2.2.1 Timed automata

Uppaal is designed to verify systems that can be modeled as a composition of *timed automata*. Timed automata are especially useful to model components of real-time systems. A timed automaton is a finite state machine extended with clock variables which evaluate to a

nonnegative real number. The timed automata in a model and the way they interact with each other together define a *timed transition system*. Besides ordinary action transitions that can represent input, output and internal actions, a timed transition system has time passage transitions. Such time passage transitions result in synchronous progress of all clock variables in the model. This makes it possible to base decisions on the moment in time on which they are taken, which is done by so-called clock constraints. A clock constraint depends on one or more clock values and can either be an invariant of a component state (defining the time period(s) in which the component may reside in this state) or a guard on a transition (defining under which time conditions this transition may be taken).

Figure 2.6 shows a timed transition system that consists of one process. We see that `state_A` has an invariant, which is a clock constraint specifying that the clock variable `c` may be at most 10 in that state. Furthermore, the transition to `state_B` has a guard, which is a clock constraint specifying that this transition may only be taken if `c` is at least 5. Because `state_A` is the initial state of this process (indicated by the double circle) and `c` is initialized to 0 (clock variables are always initialized to 0), the system first performs a time passage action in `state_A` through which time passes by any value between 5 and 10. After this time passage action the value of `c` has been increased to a value between 5 and 10, hence the system can perform the ordinary transition to `state_B`.

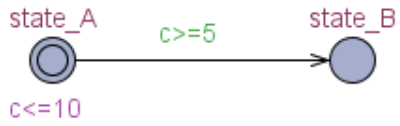


Figure 2.6: Example of time passage action

[5] contains a useful description of the formal semantics of timed automata and of algorithms and tools that are developed to analyze and verify timed automata.

2.2.2 Model specification

In Uppaal, timed automata are specified as graphs by putting locations, and edges between those locations, onto a worksheet and by editing the special properties of the locations and edges. Note that Uppaal depicts process states as locations and process transitions as edges to make clear the difference with the state of the complete system and a transition in the system.

Besides clock variables, an Uppaal model can contain bounded discrete variables that are part of the system state. These variables are used as in programming languages: they are read, written, and are subject to common arithmetic operations.

In the model, one can declare local constants, variables and functions for each process and global constants, variables and functions, which can be used by all processes. Uppaal provides predefined types for these constants and variables: integers, booleans and clocks and also custom types can be used, including (multidimensional) arrays, records, and structured types. The notational style of the declarations is based on that of C.

A state of the system is defined by the locations of all automata, the clock constraints, and the values of the discrete variables. The system as a whole moves to another state in three cases: if a time passage action is performed in the complete system (which increases all clock values with the same value and possibly changes the truth value of the clock constraints),

if an (ordinary) action is performed by one of the automata, and if two (or more) different automata synchronize over a channel, which means that transitions of these automata are performed in parallel.

The behavior of each individual automaton is defined by the locations and the edges of the automaton and their (optional) properties. We first describe the properties of edges:

- **Selection**: Selections non-deterministically bind a given identifier to a value in a given range. The other three properties of an edge are within the scope of this binding. It is also possible to bind multiple identifiers. This mechanism corresponds to quantifying over a data type.

- **Guard**: An edge is *enabled* in a state if and only if the guard evaluates to true. A guard must be a conjunction of simple conditions on clocks, differences between clocks, and boolean expressions not involving clocks. The bound must be given by an integer expression.

- **Synchronization**: Processes can synchronize over channels. The channel, on which there is communication, must be globally declared. The intuition is that two processes can synchronize on enabled edges annotated with complementary synchronization labels, i.e. two edges in different processes can synchronize if the guards of both edges are satisfied, and they have synchronization labels *e1*? and *e2*! respectively, where *e1* and *e2* evaluate to the same channel. When two processes synchronize, both edges are fired at the same time, i.e. the current location of both processes is changed. The update expression on an edge synchronizing on *e1*! is executed before the update expression on an edge synchronizing on *e2*?.

  It's also possible to define a channel as a broadcast channel. On a broadcast channel there can be multiple receivers synchronizing with one sender.

- **Update**: When executed, the update expression of the edge is evaluated. The side effect of this expression changes the state of the system. An update expression can consist of variable assignments and function calls.

Locations can have the following (optional) properties:

- **Name**: a unique name of the location. Necessary when a location must be referenced in a property (which is verified by model checking).

- **Invariant**: constraint on clock variables that must always hold when the timed automaton is in this location. System states which violate the invariants of locations are undefined; by definition, such states do not exist. An invariant is an expression that satisfies the following conditions: it is side-effect free; only clock, integer variables, and constants are referenced; it is a conjunction of conditions of the form $x < e$ or $x \leq e$ where x is a clock reference and e evaluates to an integer.

- **Initial**: boolean property that indicates whether or not the location is the initial location of the process.

- **Urgent**: boolean property that indicates whether or not a location is 'urgent'. Urgent locations are semantically equivalent to adding an extra clock x, that is reset on all

incoming edges, and having an invariant $x \leq 0$ on the location. Hence, time is not allowed to pass when the system is in an urgent location.

- **Committed**: boolean property that indicates whether or not a location is 'committed'. Committed locations are even more restrictive on the execution than urgent locations. The system state is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.
  Note that a location can not be both urgent and committed.

Figure 2.7 presents nearly all properties of edges and locations in an example model; comments are added to explain the elements of the model. The initial locations of the models are indicated by a double circle and all locations are labeled with their name. Note that the files in which the variables and functions are declared are included as rectangles in the figure.

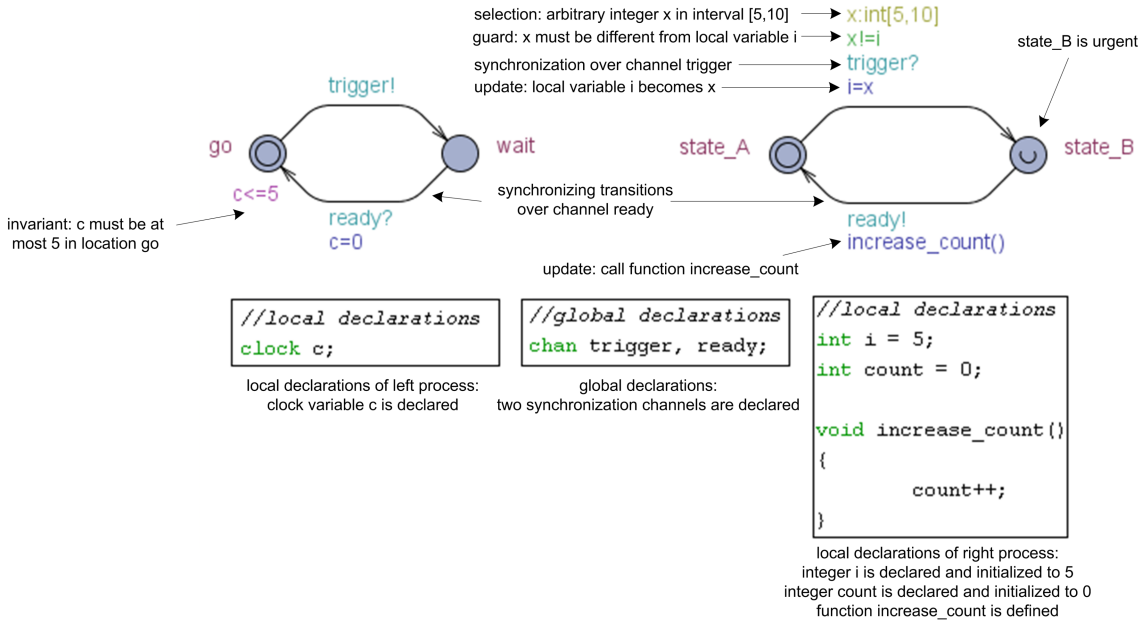


Figure 2.7: example model containing most UPPAAL constructs

In figure 2.8 an UPPAAL model specification of the coffee machine system is given. The left hand side shows a timed automaton that represents the state transition diagram of figure 2.2. To give an example of the use of clock variables, we have added a timeout feature: after choosing a drink, the user must insert enough money within 30 time units; if this does not happen, all inserted money is returned and the machine is reset. On the right hand side a timed automaton is included that models the user expectations of the system. Without this user model, verification is impossible since the purpose of the coffee machine model is reacting to 'external' inputs. Note that we have omitted the declaration files from the picture.
The edges are labeled with their properties in the same order as described above (which is also the order in which they occur in figure 2.7). Some edges synchronize over a channel from an array. For example the loop transition of location `wait` synchronizes over one of the channels from array `coin`; selection is used to quantify over the possible index values, which indicate the value of the coin that is inserted by the user. The edge from location `wait` to

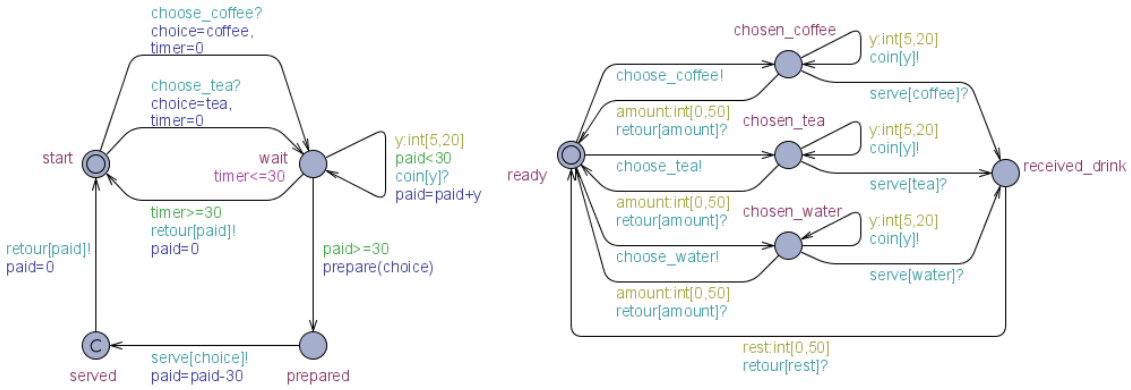


Figure 2.8: UPPAAL model of coffee machine (left) and user (right)

location `prepared` calls a locally defined function `prepare` with argument `choice`, which is a local variable. Location `served` is made committed so that too much inserted money is immediately returned after serving the drink.

An invariant is added to the location `wait` which specifies that the value of the clock variable `timer` must always be at most 30 in this location. Because the clock variable is reset on entering the location `wait`, this invariant specifies that the automaton can reside at most 30 time units in this location. The edge from `wait` to `start` that returns all inserted money is only enabled if the value of `timer` is at least 30; together the guard on this edge and the invariant of `wait` make sure that the system is reset when 30 time units are passed in location `wait`.

Note that the combination of this coffee machine model and user model implies that the user only accepts serving of the chosen drink. Should the coffee machine (due to a modeling fault) try to serve the wrong drink, the system will end up in a deadlock situation, because on a certain moment the two models try to synchronize over two different channels from the array `serve`.

2.2.3 Simulation and verification

The built-in simulator of UPPAAL can be used to examine the specified model by manually or automatically (and randomly) stepping through the model. It has already been noted that there are two types of transitions in a network of timed automata: time passage transitions which model the passing of time without changing the current location and action transitions which model changing of location in one or more (when synchronization is involved) processes. In the simulator only action transitions are stated explicitly, but delay transitions are combined with the possible action transitions when it does make sense *when* the action transition is performed. For example, when performing edge e before $t = 5$ has a different result than performing the same edge on or after $t = 5$ then the simulator will show two possible transitions: edge e in the time interval $[0, 5\rangle$ and edge e in the time interval $[5, \infty\rangle$.

The verifier can be used to perform the actual model checking of a desired property. The verifier uses a symbolic technique that reduces verification problems to that of efficient manipulation and solving of constraints, which are conditions on clock variables (see section 2.2.1). Furthermore the verifier uses "on-the-fly" exploration of the state-space of the specified model. For the verification, various options are available; these include possible state space

reduction techniques, different search orders and the possibility to request a diagnostic trace. A diagnostic trace can be a *counterexample* or *witnessing trace* (depending on the kind of property), that is generated during verification. The trace is loaded into the simulator after verification, so that one can check why the property is (not) true.

Properties are specified in UPPAAL's requirement specification language, which is a restricted temporal logic (see section 2.0.2) and has the following BNF-grammar:

```
Prop ::= A[] Expression | E<> Expression | E[] Expression
       | A<> Expression | Expression --> Expression
```

Expressions can contain all global and local variables and can be built up from all well-known boolean and arithmetic operators. All expressions must be side effect free. Furthermore it is possible to test whether a certain process is in a given location using expressions of the form `process.location`.

The semantics of the property constructs are as follows:

- Possibly `E<>`:
  The property `E<> p` evaluates to true for a timed transition system if and only if there is a sequence of alternating time passage transitions and action transitions `s0-->s1-->...-->sn`, where `s0` is the initial state and `sn` satisfies `p`.

- Invariantly `A[]`:
  The property `A[] p` evaluates to true if and only if every reachable state satisfies `p`.

- Potentially always `E[]`:
  The property `E[] p` evaluates to true for a timed transition system if and only if there is a sequence of alternating time passage and action transitions `s0-->s1-->...-->si-->...` for which `p` holds in all states `si` and which either is infinite or ends in a state with no outgoing transition (and until infinity this state never gets an outgoing transition either).

- Eventually `A<>`:
  The property `A<> p` evaluates to true if and only if all possible transition sequences eventually reach a state satisfying `p`.

- Leads to `-->`:
  The syntax `p --> q` means that whenever `p` holds eventually `q` will hold as well.

The special property `A[] not deadlock` can be verified to make sure that the model can't get stuck in a *deadlock state*.

After specifying a property the verifier can be started. Unfortunately, the graphical interface of UPPAAL does not give any measure of progress during verification of the property. Although, when the command-line verifier is used, information about the throughput (number of processed states per second) and the load (number of explored states) is given during verification. When the verification has finished, a statement is returned about the validity of the property and possibly a trace is sent to the simulator which can be inspected.

On the UPPAAL model of the coffee machine system, we can for instance check the following properties:

- "The user can choose for water" which is translated into UPPAAL's requirement specification language as follows: `E<> User.chosen_water`. Literally this formula expresses that state `chosen_water` in the user model is reachable. The result of the verifier on this formula will be `Property is not satisfied` because there is no transition in the machine model that synchronizes over the channel `choose_water`. If we request a diagnostic trace for this property, we do not get one because such a trace does not exist.

- "When a drink has been prepared, always at least 30 cents are inserted" is translated as: `A[] Machine.prepared imply Machine.paid>=30`. This expresses that the local variable `paid` is always at least 30 if the machine model is in state `prepared`. Verification will result in `Property is satisfied` because of the guard on the transition from state `wait` to state `prepared` and for this property again no diagnostic trace exists.

- "After choosing coffee, the user always gets his/her coffee" is translated as: `User.chosen_coffee --> User.received_drink`, which is clearly not satisfied by the model. Requesting a diagnostic trace will give us a trace in which the user chooses a drink, does not insert any coins and after 30 time units the machine is reset without serving the coffee to the user.

We have seen how to model a system in UPPAAL and how to check some properties on this model. Also we have seen how simulation and diagnostic traces can provide us feedback on the model.

Chapter 3

The human heart

The purpose of an artificial pacemaker is to maintain an adequate heart rate when the heart's own pacemaker is too slow or when there is a block in the heart's electrical conduction system. Since both of these reasons why patients need an artificial pacemaker reside in the electrical conduction system of the heart, this chapter focuses on this system. The presented information is based on Vitatron's internal education modules [7] and [8]; more information can be found in [15] and [17].

3.1 The heart function

The heart is a muscular organ, responsible for pumping blood through the blood vessels. This is done by rhythmic contractions of the *atria* (the upper chambers of the heart) and the *ventricles* (the lower chambers of the heart). The flow of blood through the heart is visualized in figure 3.1: de-oxygenated blood flows into the right atrium, which upon its contraction pumps this blood into the right ventricle. The ventricle contracts a short interval after the atrial contraction and pumps the blood to the *lungs*. The short interval is typically around 0,2 seconds and is necessary to fill the ventricle completely to maximize the effectiveness of the heart's pump function. The lungs provide oxygen to the blood and the blood is returned to the heart, where it flows via the left atrium and the left ventricle (analogous to the flow through the heart's right side) into the aorta to the rest of the body.

3.2 The electrical conduction system

The electrical impulses that create and coordinate the contraction of the atria and ventricles are generated and conducted by the conduction system of the heart. Before we discuss this conduction system, we must know more about the cellular structure of the heart's muscle tissue.

3.2.1 Cellular structure

Each cell consists of a nucleus surrounded by a jelly like substance called cytoplasm. The nucleus and cytoplasm are surrounded by a cellular membrane. The cell membrane separates the inside of the cell from the outside and preserves its integrity. An important function of

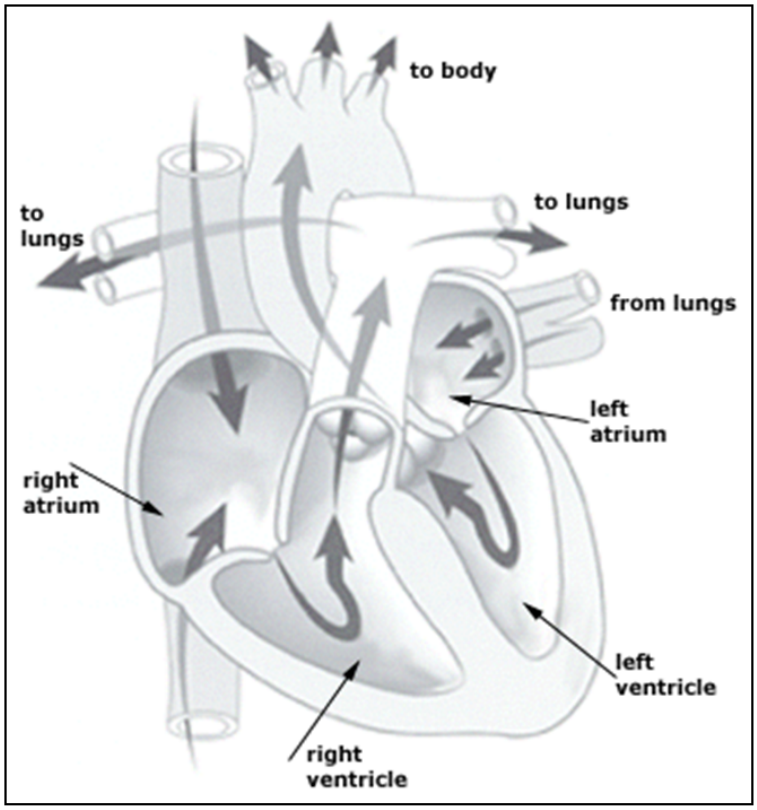


Figure 3.1: Blood circulation in the heart
(picture taken from Vitatron internal training program level B, 29-03-2005)

the cell membrane is to keep the concentrations of ions at certain values. The concentrations of sodium(Na^+), potassium(K^+), and calcium(Ca^{++}) inside the cell are different from the concentrations of these ions in the surrounding fluid. The difference in concentrations would normally disappear, because of diffusion through the semi permeable membrane. But the membrane has an active mechanism, the sodium pump, which constantly maintains the concentrations of the ions inside and outside the cell at different levels.

The difference in ion concentrations results in a negative charge inside the cell and a positive charge outside. The difference in potential is called the resting membrane potential, and ranges from -70 to -100 mV. Stimulation of the cell can cause a change in the cell's membrane potential. This can be an electrical stimulation from an artificial pacemaker or an electrochemical and mechanical stimulation, caused by contraction of a neighbor cell. Such a stimulation of the cell causes the permeability of the membrane for sodium and calcium ions to increase, allowing an influx of these ions into the cell. When the sodium influx is able to reduce the membrane potential to the threshold level (approximately -40 to -60 mV), a massive influx of sodium ions brings the transmembrane potential to approximately +20 mV. The change in potential is called depolarization. The state of depolarization is quickly transferred to the adjacent muscle cells, causing a depolarization of the entire muscle mass.

After the depolarization the muscle contracts immediately. At this moment an inflow of chlorine ions (Cl^-) initiates the repolarization process and the inside becomes negative again.

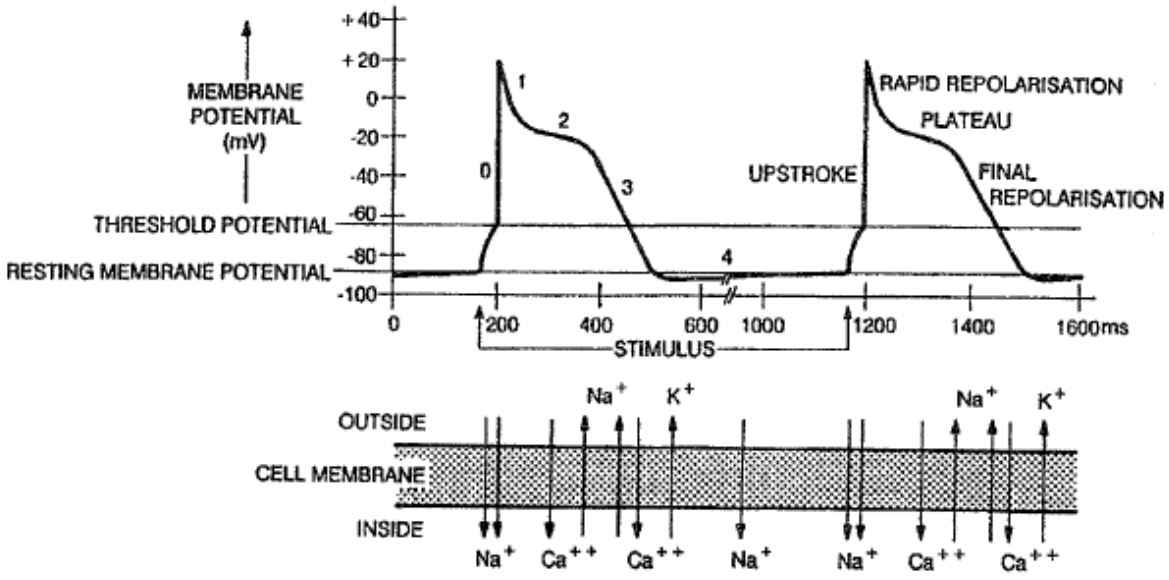


Figure 3.2: Phases in depolarization and repolarization process
(picture taken from [7])

Figure 3.2 gives a schematic overview of the five phases in the depolarization and repolarization process of the cardiac muscle cell. The graph gives the membrane potential as a function of time during two complete cycles of the depolarization and repolarization process. In phase 4 the resting potential is maintained by the sodium pump and phase 0 represents the depolarization as a result of the rapid sodium and calcium inflow. In phase 1 the cell repolarizes rapidly through the chlorine inflow and phase 2 represents a period of relative stability of the membrane, which is called the plateau phase. Finally in phase 3 the repolarization process completes through sodium and potassium outflow.

An important part in the process of depolarization and repolarization is the *refractory period*. The refractory period starts immediately after the depolarization and stops gradually during the repolarization phase. Hence, in figure 3.2 this period covers phases 1, 2 and partially phase 3. During the refractory period the tissue is less (relative refractory) or not at all (absolute refractory) excitable, meaning that it does not react to stimulations. The refractory period prevents that a stimulus starts cycling, because of the mechanism that a depolarized cell transfers its depolarization state to all the adjacent cells.

Actually the potential in phase 4 is not completely constant, because all heart cells show an interesting phenomenon in phase 4, caused by the slow influx of sodium ions. Through this influx the transmembrane potential slowly decreases towards the threshold potential (see figure 3.2). In most cells, long before the threshold has been reached, the next externally stimulus arrives and the depolarization is started as a result of this stimulus. Without the appearance of this stimulus, the potential of the cell will eventually reach the threshold value and depolarize automatically. This phenomenon explains the automaticity of some specialized cardiac cells, the pacemaker cells. Tissues made from these specialized cells can be found in the sinus node (appointed SA node in figure 3.3) and in some other elements of the conduction system. In these specialized cells the slow depolarization in phase 4 is faster than in other heart cells, although there are also different types of these specialized cells. The cells in the

sinus node are of the slowly responding type. Cells of the slowly responding type have a low conduction velocity but a relatively high automatic depolarization rate. Cells of this type are therefore suitable for the generation of signals. Next there are cells of a fast responding type which have a high velocity of conduction and a relatively low automatic frequency. Cells of this type are suitable to transport signals from one part of the heart to the other.

3.2.2 Functional description of the conduction system

In figure 3.3, a schematic picture of the conduction system is given.

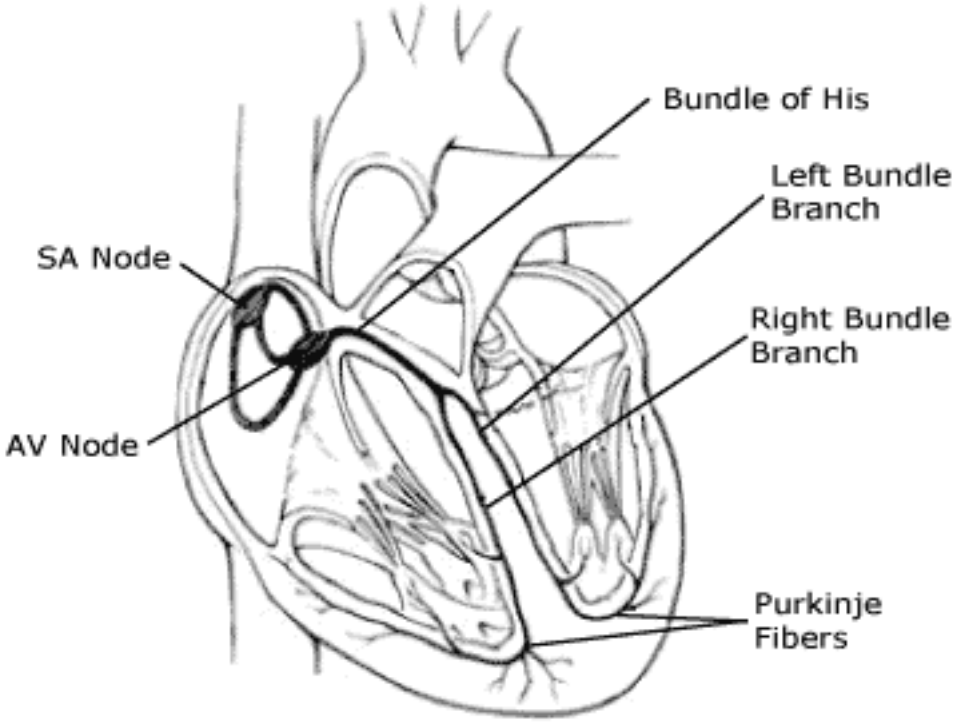


Figure 3.3: The heart's conduction system
(picture taken from www.onevalveforlife.com; ©St. Jude Medical, Inc.)

The sinus node (also called SA node or sino-atrial node) consists of a group of specialized pacemaker cells and is located in the upper part of the right atrium. The sinus node cells are of the slowly responding type and thus have a high automatic depolarization frequency (typically between 60 and 100 min^{-1}, but dependent of the person's exercise level), which is the highest of all pacemaker tissues in the heart. Because the sinus node has the highest depolarization frequency it determines the rate of the complete heart which is the reason why the sinus node is called the natural pacemaker.

The electrical signal from the sinus node is transferred to the right atrium and left atrium and causes there an atrial contraction. This happens because a chain of stimulations is started by the sinus node signal: the sinus node's neighbor muscle cells depolarize and spread the signal to all their neighbor cells, etcetera.

The sinus node signal is also quickly transported via three pathways of conductive tissue to the atrio-ventricular or AV node located in the lower part of the atrium (see figure 3.3).

The AV node is activated and, because its cells are also of the slow responding type, delays the further conduction of the signal by approximately 0,08 to 0,2 seconds. The delay prolongs the filling time of the ventricles before the ventricular contraction is initiated.

After the delay in the AV node, the signal is passed to the only channel of communication between atria and ventricles, the His bundle. The His bundle divides into the right and left bundle branches, which divide further into the finer Purkinje fibers forming the final contact with the muscle cells of the ventricles. The His bundle, right and left bundle branches and the Purkinje fibers are all made of cells of the fast responding type. Hence, the speed of conduction through these channels is faster than in the muscle cells of the ventricles itself. This is important, because it causes the depolarization of the muscle cells, where the Purkinje fibers are present, to start all at the same moment. This optimizes the ventricular contraction into the direction of the outflow valves.

3.3 Artificial pacemakers

An artificial pacemaker (also called implantable pulse generator to avoid confusion with the heart's natural pacemaker) is a medical device which uses electrical impulses, delivered by leads contacting the heart muscles, to regulate the rhythm of the heart. The primary purpose of a pacemaker is to maintain an adequate heart rate, either because the heart's own pacemaker is too slow, or there is a block in the heart's electrical conduction system. Modern pacemakers are externally programmable and allow the cardiologist to select the optimal pacing modes for individual patients. The cardiologist has a lot of freedom in setting these modes, because the pacemaker contains several therapies that can be switched on or off.

As already said, leads form the connection between the pacemaker and the heart. These leads can be attached to either the right atrium only, to the right ventricle only, or to both the right atrium and the right ventricle. Figure 3.4 shows the position of the leads of a dual chamber pacemaker, which has leads in both the right atrium and the right ventricle.

When the muscle cells in the neighborhood of a lead depolarize spontaneously, this can be measured by the pacemaker as electrical activity on this lead. The measurement of electrical activity on a lead is called *sensing* and this actually forms the input of the pacemaker. Furthermore the pacemaker is able to deliver *paces* on the same lead, which are short electrical pulses to stimulate the muscle cells in the neighborhood of the lead. The electrical pulse will cause these muscle cells to depolarize and hence causes the start of a chain of depolarizations. So by delivering a pace on a lead that is attached to the muscle cells of the ventricle, the pacemaker causes a ventricular contraction to occur. Basically we can define the operational interface of a pacemaker as follows: its two possible inputs are *atrial senses* and *ventricular senses*, which are measured spontaneous contractions of the corresponding heart chambers and its two possible outputs are *atrial paces* and *ventricular paces*, which are electrical stimulations to force a contraction to occur.

The pacemakers of Vitatron have different modes, which are described by a three-letter-code:

- the first letter indicates in which chamber(s) the pacemaker can deliver paces. There are four options: O=None, A=Atrium, V=Ventricle, D=Dual, in which Dual means that pacing is possible in both the atrium and the ventricle.
- the second letter indicates in which chamber(s) the pacemaker can sense spontaneous contractions. (O=None, A=Atrium, V=Ventricle, D=Dual)

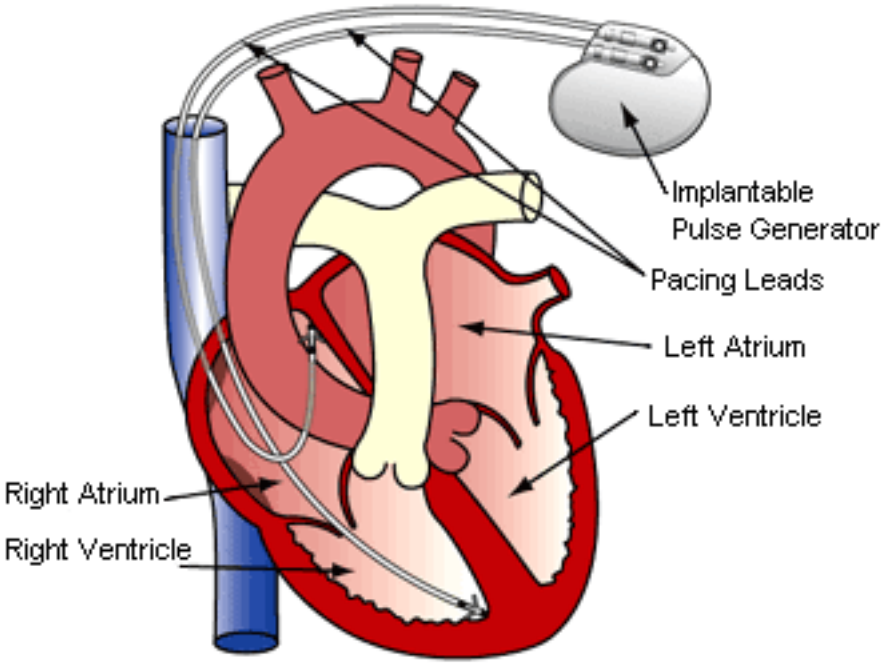


Figure 3.4: Dual chamber pacemaker
(picture taken from www.londonafcentre.co.uk; ©2005 Dr Richard Schilling)

- the third letter indicates how the pacemaker responses to sensing. O=None, T=Triggered, I=Inhibited, D=Dual(=both Triggered and Inhibited). In case of 'Dual' the pacemaker delivers a ventricular pace some time interval after an atrial sense. 'Inhibited' means that a planned atrial pace is not given when an atrial sense has been detected.

Some of the most used modes are for instance AAI, VVI and DDD. In this project we focus on the DDD pacemaker, which can be seen as the most sophisticated mode of Vitatron's pacemaker and therefore is the most challenging when it comes to model checking.

3.4 Arrhythmias

As already indicated, the purpose of an artificial pacemaker is to maintain an adequate heart rate when the electrical activity of the heart itself is irregular or slower than normal. Such conditions in which the electrical activity of the heart is not appropriate are called *cardiac arrhythmias*. In this section we will describe a few important arrhythmias and the possible measures that can be taken by the artificial pacemaker to handle these arrhythmias.

- Sinus tachycardia:
  The sinus node depolarizes faster than normally (typically > 100 min^{-1}). An artificial pacemaker can't do anything about this, because it can't slow down the sinus node. However it tries to maintain a normal ventricular rate by pacing the ventricle. Then the phasing of the atrial and ventricular contractions will not be correct which decreases

the effectiveness of the heart's pump function. But because a normal ventricular rate is maintained there is at least a regular blood supply to the body.

- Sinus bradycardia:
  The sinus node depolarizes slower than normally (typically < 60 min^{-1}). The artificial pacemaker will deliver paces at a certain minimum rate in the atria when sinus bradycardia is discovered.

- AV block:
  The AV node function is disturbed. There are different types of AV block, varying from only a prolonged conduction interval through the AV node (first degree AV block) to a complete block (third degree AV block). In case of a complete AV block, the sinus node determines only the contraction frequency of the atria and the contraction frequency in the ventricles is determined by a focus somewhere in the ventricular tissue. This focus generally is the group of cells in the ventricle with the highest automatic depolarization frequency. A depolarization in this group of cells will cause the entire ventricle to contract. Since the automatic frequency of the cells generally is lower than the atrial contraction frequency, there is no relationship between the atrial and ventricular rhythms. The pacemaker will follow the normal atrial rate by pacing the ventricle a certain AV-delay after each atrial sense as to mimic the normal function of the AV node. This is called tracking.

- Premature ventricular contractions (PVC):
  A PVC is a ventricular contraction which is not the result of an earlier atrial contraction or a stimulus from an artificial pacemaker, but originates from a spontaneously depolarizing muscle cell of the ventricle. A ventricular contraction will be classified as PVC when it is not preceded by an atrial event (sense or pace). After a PVC the artificial pacemaker can deliver an atrial pace in order to make the atrium *refractory* and thus preventing that the PVC is conducted in reverse direction to the atrium (see below).

- Retrograde conduction:
  Retrograde conduction is the conduction of a stimulus from the ventricles via the AV node to the atria (so via the 'retrograde' path, as opposed to the normal 'anterograde' path from the atria to the ventricles). One of the causes of retrograde conduction can be a PVC that occurs when the AV node and the atria are not *refractory*. The depolarization signal then is conducted from the ventricles to the atria and the atria contract a short interval after the ventricular contraction. This is the wrong order and must be avoided or at least the effects must be minimized. When the artificial pacemaker is not aware of the retrograde conduction, the effect could be that it tracks each (retrograde conducted) atrial sense by a ventricular pace, which again is retrograde conducted, and so on. This results in a too high heart rate and is highly undesirable. When the artificial pacemaker detects this effect, it stops tracking the atrial sense in order to break the cycle of retrograde conduction.

- Atrioventricular crosstalk:
  It may happen that shortly after an atrial pace a ventricular event is sensed. This sense may not be induced by an actual depolarization, but may be caused by atrial crosstalk artifacts (the atrial pace is detected in the ventricle and interpreted as a ventricular depolarization). Such a ventricular sense is called a crosstalk ventricular sense. This

may result in the inhibition of the ventricular pace which is highly undesirable. To avoid inhibition a Ventricular Safety Pace (VSP) is delivered by the artificial pacemaker to ensure ventricular contraction in case of crosstalk detection.

Chapter 4

Pacemaker architecture

In this project the firmware design of Vitatron's DA+ pacemaker is used as a case study to investigate the feasibility of model checking at Vitatron. This chapter gives an overview of the architecture and the behavior of the firmware and its environment.

An artificial pacemaker or implantable pulse generator (IPG) consists of the following components(see figure 4.1):

- A can that surrounds the IPG and physically isolates the electronics from the environment (the patient's body when the IPG has been implanted).
- A hybrid that consists of digital and analog hardware and firmware (in ROM on the microprocessor). The firmware is a piece of embedded software that controls the behavior of the hardware such that the IPG maintains an adequate rate in the patient's heart.
- A battery that powers the electronics.
- An antenna for wireless communication with the programmer, which is an external device through which the cardiologist can program patient dependent parameters.
- A connector to which the leads are attached.

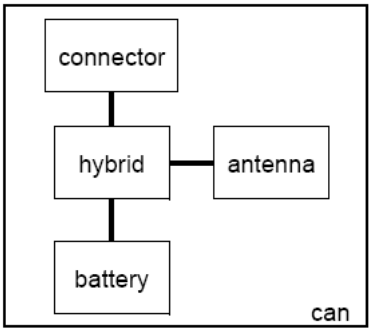


Figure 4.1: IPG overview

The hybrid forms the intelligent core of the IPG. Figure 4.2 gives an overview of how the firmware is situated with respect to the hardware in the hybrid. The hybrid contains a hardware module that provides a telemetry interface to program an read firmware parameters; this telemetry hardware collaborates with a firmware component (Physician Programmer in

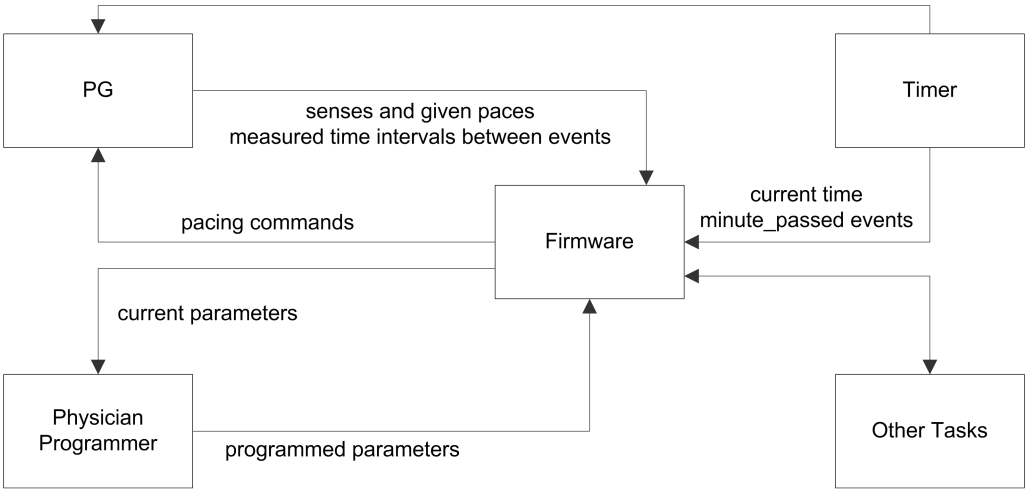


Figure 4.2: Firmware environment

figure 4.2). Then there is a module that provides the current time and events for each passed minute. Furthermore there is a number of modules for other tasks, e.g. a battery measurement module.

But the most important hardware module from firmware perspective is the pulse generator (PG). The PG can be seen as the interface between the pacemaker leads and the firmware; it delivers paces to the heart on request of the firmware and provides heart diagnostics back to the firmware. These diagnostics include notifications when a contraction is sensed and when a pace is given (called sense and pace events) and time intervals that inform the firmware about when these events happened. The PG contains an atrial timer and a ventricular timer that measure the elapsed time since the last atrial or the last ventricular event, respectively. When a contraction is sensed or a pace is given, these timers contain the time interval between the sense or pace and the last atrial or last ventricular event, respectively. Together with the sense or pace event, the values of the timers are sent to the firmware. In the firmware these values are denoted by aa_int (representing the time interval between the last atrial event and the last but one atrial event), av_int (representing the time interval between the last ventricular event and the atrial event before this ventricular event), va_int and vv_int.

Here some text is removed for confidentiality reasons.

This text describes how the firmware requests the PG to deliver a pace and how the PG ensures that this pace is given on the right moment in time.

After this overview of the environment in which the pacemaker firmware operates, we come to the description of the firmware itself. The pacemaker firmware is an event-driven system; processing occurs only in response to an external event by executing an interrupt service routine, after which the event is put in a queue of the operating system. The sense and pace events from the PG are the most important external events when the pacemaker is in operation; other examples of external events are parameter update events and the minute_passed

events.

In the firmware development process, the Hatley and Pirbhai structured analysis method (H&P method in the sequel) is used to specify the functional firmware requirements and the firmware design. Also the DA+ firmware design is created using the H&P method. In the following section the H&P method is introduced for a better understanding of the mCRL2- and UPPAAL-model structures. For more detailed information about the H&P method itself, see [12].

4.1 The Hatley and Pirbhai method

The Hatley and Pirbhai method consists of modeling tools and techniques to specify both a system's function and its physical partitioning. The system specification must define what problem the system is to solve (its requirements) and how that system is to be structured (its architecture or design structure). To this end the system specification is separated into two models; the requirements model and the architecture model.

In this project, only the system requirements model of the DA+ pacemaker firmware is used as a basis for the formal models. The system requirements model specifies the flow of events and data between the system components and how these events and data are processed in the components. All processing in the model is assumed to be infinitely fast. The requirements model is built as a layered set of Data Flow Diagrams (DFDs) with associated processing specifications, called PSPECs. Each successive level of DFDs expresses a refinement of the higher-level DFDs and the most detailed DFDs only contain PSPECs. On each level this results in a network of processes, representing DFDs on the successive level and PSPECs, interconnected by data flows and control flows.

The system requirements model shows how each process transforms its input data flows into output data flows, and how the processes are related to each other. To this process structure a control structure is added which determines what the process structure must do under any given external or internal conditions or operating modes. To this purpose process controls are generated from the logic in the so-called CSPECs (described below) which activate and deactivate DFDs and trigger PSPECs in the process structure. The control structure also receives information about the status or mode of external systems and transmits similar information about itself.

After this overview of the system requirements model, we give a slightly more detailed description of the basic elements of a system requirements model:

- Data flows: A data flow is a pipeline through which data of known composition flows. This data may consist of one element or a group of elements. Data flows are always continuous in the time domain and can be read and reread any number of times by a receiving process until its source process stops generating it.

- Control flows: A control flow is a pipeline through which control information of known composition flows. Just as in a data flow, this control information may consist of one element or a group of elements. Control flows are time transient and can be seen as events that activate a certain calculation in the receiving process.

- Data stores: A store is simply a data flow frozen in time. The data and control information a store contains may be used any number of times by any number of processes, and remains available until replaced with something new.

- PSPEC: A Process SPECification is the elementary process that is specified by a sequence of statements in pseudo-code. A PSPEC can be triggered by a CSPEC (event triggered) or by a change of an incoming data flow (data triggered). When a PSPEC is triggered, it executes its pseudo-code which transforms the incoming data flows into outgoing data and control flows.

- CSPEC: Each DFD possibly contains a Control SPECification to control the handling of incoming control flows. CSPECs convert input control flows into output control flows and process controls (through which PSPECs and child DFDs can be triggered). The CSPEC can be a process activation table (PAT), a state transition diagram (STD) or a combination of these two.

  If the CSPEC is only a PAT, it specifies the consequences of the receipt of each possibly incoming event in a table. The possible consequences can include activations and deactivations of child DFDs, the output of events on a control flow, and triggering PSPECs and child DFDs. In the latter case also the order in which the PSPECs and child DFDs are triggered can be specified.

  In an STD, the consequences of an incoming event are dependent of the current state of the STD. An STD is just a simple labeled transition system whose transition labels define on which incoming event the transition is performed and what the consequences of the transition are. In an STD these consequences can be triggering PSPECs and child DFDs, sending an event to another STD of the diagram, and sending an event to a corresponding PAT. In the latter case the corresponding PAT specifies the actual consequences of the transition which keeps the STD readable. Figure 4.3 shows the format of an STD in an H&P model.

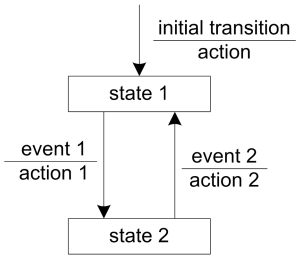
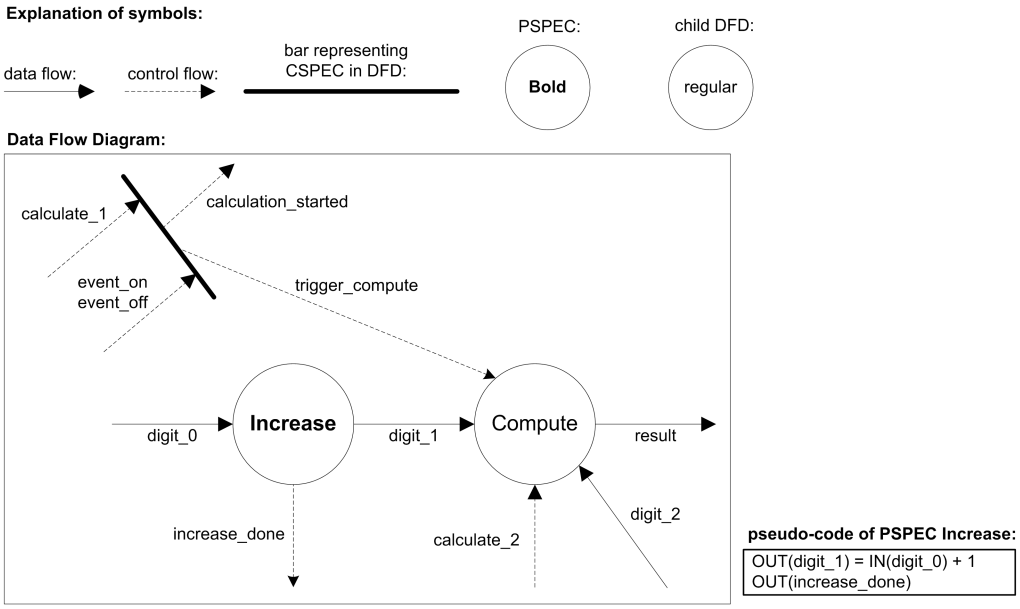


Figure 4.3: State Transition Diagram

Figure 4.4 shows an example of a DFD and its corresponding CSPEC. The DFD shows that the CSPEC has three incoming control flows and two outgoing control flows. The CSPEC defines how the receipt of the events `event_on`, `event_off` and `calculate_1` is handled; in this case a PAT is used for this purpose. We see that `event_on` results in the activation of the child DFD `Compute` and `event_off` deactivates `Compute`. Furthermore the CSPEC defines that on the receipt of event `calculate_1`, first the PSPEC `Increase` is triggered and then the outputs events `trigger_compute` and `calculation_started` are delivered (both outputs are indicated by a '2' in the PAT, meaning that the sending order of these events does not matter).

The behavior of the PSPEC `Increase` is defined by the pseudo-code that is listed beside the DFD. Above we have seen that the CSPEC triggers this PSPEC, so this PSPEC is event



CSPEC (PAT specifying the actual contents of the bar in the DFD):

	Inputs			*Outputs*		*Process activation*	
Name	event_on	event_off	calculate_1	calculation_started	trigger_compute	Increase	Compute
1	x						on
2		x					off
3			x	2	2	1	

Figure 4.4: Example: Hatley & Pirbhai requirements model

triggered, meaning that its pseudo-code is only 'executed' upon receipt of a trigger from the CSPEC. On receipt of a trigger, the value on the data flow `digit_0` is incremented and put on the data flow `digit_1` and subsequently the output event `increase_done` is delivered (to a higher-level diagram).

As long as the child DFD `Compute` is active (no event `event_off` has been received after the last `event_on` event), the data flow `result` contains the result of the last calculation. An incoming event `trigger_compute` or `calculate_2` will only be processed by `Compute` if it is activated and will probably cause a change of the value on the output data flow `result`. After deactivation of `Compute` the value of `result` will not change until its next activation.

4.2 Pacemaker H&P model

Figure 4.5 shows a high-level flow diagram of the pacemaker (therapy related) firmware requirements model. The behavior of the pacemaker firmware is largely determined in this flow diagram and its child diagrams. Besides this flow diagram there are only some diagrams that are responsible for interfacing between the PG and the displayed diagram, synchronization of parameter changes, and some mode specific interfacing between the firmware and other hardware modules.

Of each process diagram that is contained in the diagram of figure 4.5 we will briefly describe its functionality.

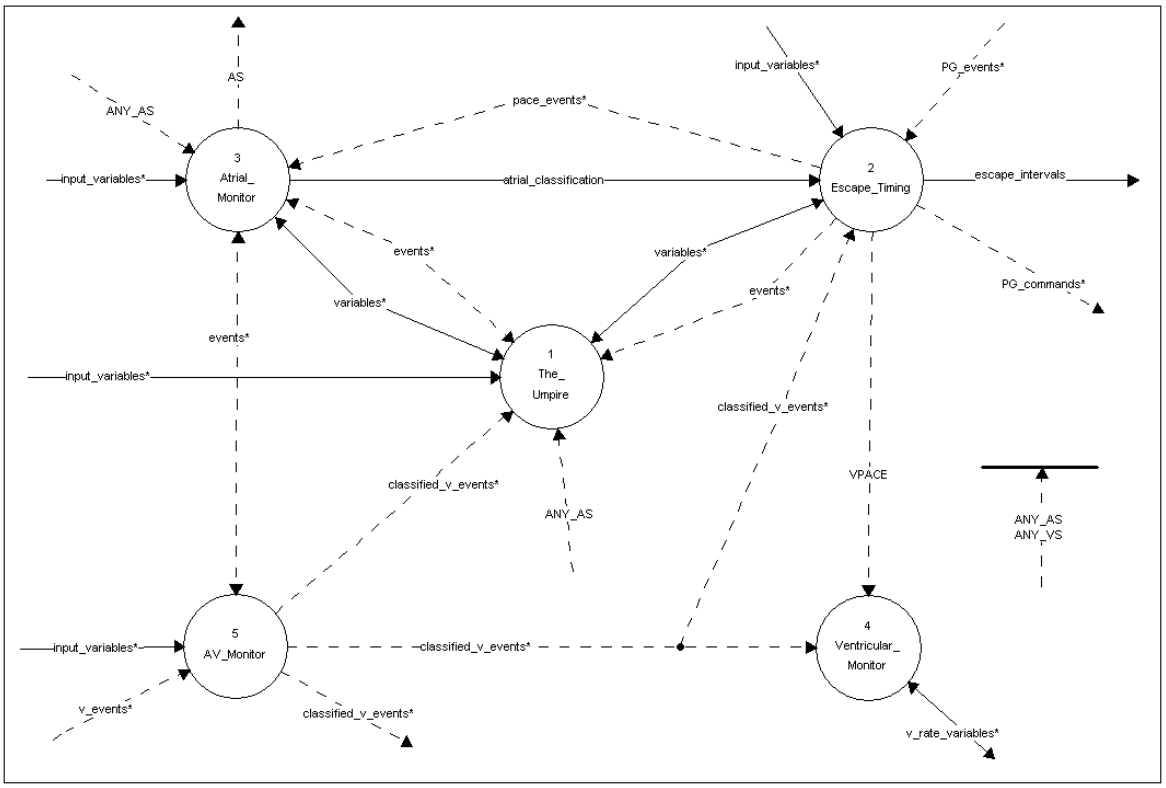


Figure 4.5: Flow diagram of pacemaker firmware requirements model

- The_Umpire:
  The_Umpire contains a number of subprocesses, which mainly represent pacemaker therapies. Each pacemaker therapy can be switched on or off by the patient's cardiologist and controls the pacing rate. Various therapies can be active simultaneously and therefore The_Umpire must decide which therapy is controlling the pacing rate. Therapy selection is based on the priority of the therapies and the rate that they suggest. Furthermore The_Umpire controls the interactions between the various therapies. Examples of therapies are the RC_Test, which controls a test to check if retrograde conduction (RC; see section 3.4) is present or absent and the Lower_Rate therapy that maintains a certain minimum rate.

- Atrial_Monitor:
  This process classifies each incoming atrial event based on the timing of the event itself and the recent heart behavior.

  A so-called physiological band is maintained by the Atrial_Monitor; based on this band atrial events are classified. Figure 4.6 shows an overview of the physiological band. The band consists of the physiological rate, an upper bound, and a lower bound. These bounds are set to a predefined distance from the physiological rate. The physiological rate is the center of the band and follows changes of the atrial rate. As figure 4.6 shows, the band (together with the parameters Tachy Rate and Brady Rate) determines five classes of atrial events (note that the rate of the received atrial event is defined as the

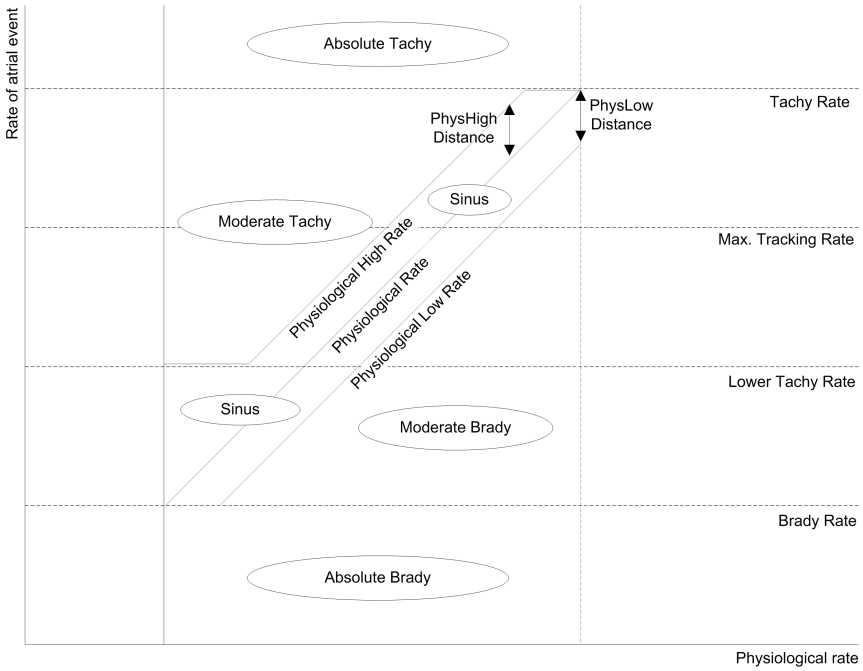


Figure 4.6: Physiological band of Atrial_Monitor

inverse of the time interval since the last atrial event):

- Absolute Tachy for atrial events that are received in a rate that is higher than the Tachy Rate.
- Moderate Tachy for a rate between the upper bound and the Tachy Rate.
- Sinus for a rate between the lower and the upper bound.
- Moderate Brady for a rate between the Brady Rate and the lower bound.
- Absolute Brady for a rate that is lower than the Brady Rate.

Upon each receipt of a Sinus atrial event, the band is updated; the physiological rate and the corresponding bounds are increased or decreased in steps of 2 if the rate of the received event is higher or lower, respectively than the current physiological rate.

Besides this, the Atrial_Monitor contains the RC_Monitor that classifies whether an atrial sense is caused by retrograde conduction or not. This RC_Monitor works in close collaboration with the therapy RC_Test.

- Ventricular_Monitor: A simple process that updates a moving average of the ventricular rate on each incoming ventricular event.
- Escape_Timing:

In the description of Escape_Timing, some small changes are made for confidentiality reasons.

This process plays a crucial role in the pacemaker's goal to mimic (and restore) the mechanical contraction sequence of the normal healthy heart. If an adequate atrial

rate is sensed, according to the patient's activity level, the pacemaker is said to be in synchronous mode and it will base its behavior on this atrial rate. In synchronous mode, each trackable atrial sense or atrial pace is said to be tracked by a ventricular pace and a subsequent atrial pace. Whether an atrial sense is trackable is decided in a child diagram of Escape_Timing based on the classifications from the Atrial_Monitor and a variable that depends on the ventricular rate average from the Ventricular_Monitor.

In case an atrial contraction is sensed that is not trackable, which means that it comes too fast after the previous atrial event, the pacemaker switches to asynchronous mode. The pacemaker prevents that the ventricular rate will suddenly increase too much, and tries to stabilize the ventricular rate by pacing the ventricle independently from the atrium. In asynchronous mode, the atrial rate is no longer tracked, but the pacing rate will be based on the ventricular rate. During asynchronous mode the system will continuously strive to return to synchronous mode.

- AV_Monitor:
  The AV_Monitor classifies all ventricular senses and paces according to their order with respect to previous events. For instance, the AV_Monitor detects whether or not a ventricular sense is a PVC (see section 3.4). The classifications are sent to Escape_Timing, Atrial_Monitor, and The_Umpire whose decisions depend on them.

Notice that the CSPEC is displayed in the bottom right corner of the diagram in figure 4.5 and that it only has four incoming control flows. For these control flows (the upper two are the atrial sense event and the ventricular sense event), the CSPEC defines their processing order through a process activation table. The other incoming control flows in the diagram are the input for only one child diagram and therefore they are not addressed by the CSPEC.

From the described H&P diagram all elements are formally modeled except some parts of the Atrial_Monitor and many parts of The_Umpire. In the next chapter we will see how the various H&P constructs are translated to a formal model.

Chapter 5

Formal modeling of pacemaker firmware

During the project several formal models have been constructed to investigate the feasibility of model checking on the firmware designs of Vitatron. These models differ in which firmware components are contained in the model and in which environmental processes are modeled besides the firmware processes.

For example, we initially started with a model that contained only a small part of the firmware, a simple process simulating the behavior of the hardware module PG (see chapter 4), and a group of processes simulating some basic heart behavior. In this model, the firmware process was a model of some core functionality to deliver paces at a certain rate. Later on, we added more firmware functionality to the models and concurrently also the PG process and the heart processes were extended to support the firmware extensions in the formal models.

This chapter gives a description of some technical details of the formal models and of the design decisions that were made during the formal modeling of the pacemaker firmware. These design decisions mainly concern the translation of the H&P model constructs into the formal languages of the model checking tools. The described choices and modeling techniques slowly emerged during the construction and the verification of the models; they are described in this report because they form the basis of the models. By keeping these choices and techniques in mind, the construction of the models was actually just a matter of straightforward translation of the H&P model. During this translation we tried to deviate as little as possible from the H&P model; the only necessary deviations that were made are described in this chapter. Because we think all these deviations are justified, we have high confidence that the level of conformance between the H&P model and the formal model is rather high.

Section 5.1 describes the design decisions and technical details that are applicable on all formal models of the pacemaker firmware. Because some technical aspects of the models are dependent of the model checking tool that is used, sections 5.2 and 5.3 describe the tool specific technical aspects of the models.

Section 5.4 describes the design of the heart simulation processes and the PG process, because they form the environment in which the firmware model is verified in some models. Finally in section 5.5 we mention some techniques that have been applied to reduce the state space of the models. Note that these state space reducing techniques decrease the level of conformance between the H&P model and the formal model, but this is inevitable if we want

to make model checking possible. The actual model checking approaches and their results will not be described in this chapter, but in chapter 6.

5.1 General aspects of the formal models

When translating (a part of) a Hatley and Pirbhai system requirements model to a formal model, the first design decision concerns the grouping of the H&P processes (DFDs and PSPECs) into concurrent processes in the formal model. If this grouping is chosen too coarse-grained, i.e. too much H&P processes are put in one formal concurrent process, the concurrent processes become too complex and also the benefits that concurrency in the formal model gives are lost. If for example a concurrent process is created that represents two different (but related) DFDs with corresponding STDs, the concurrent process must take all state combinations of these two STDs into account. Modeling the two DFDs each as a concurrent process avoids this and makes the modeling easier, because each process only has to bother about the state of its own STD, thus resulting in a decrease of the total number of 'state handling routines'.

On the other hand, in a too fine-grained grouping, the overhead of communication becomes large which is also undesirable. These considerations lead to the decision that each DFD that does not contain child diagrams (but only PSPECs) is modeled as a separate concurrent process. The formal process description defines the actions that are performed on receipt of an event from each incoming control flow to the diagram. Usually the diagram's CSPEC is the basis of the process description; it executes the pseudo-code of the PSPECs that are called by the CSPEC.

Besides the concurrent processes for these lowest level H&P DFDs, the formal models also contain concurrent processes for most intermediate level H&P diagrams. Often such an intermediate level diagram does not contain any PSPECs and its function is only the redirection of incoming events to its child diagrams. Then the diagram's CSPEC, which manages these redirections, is directly translated to a concurrent process. If an intermediate level diagram also contains some PSPECs, the functionality from these PSPECs is combined with the redirection of the incoming control flows.

However, to this latter statement there is one exception in the DFD Atrial_Monitor which contains a relatively complex PSPEC Atrial_Rate_Classifier. This PSPEC performs a calculation on receipt of an event; the possible incoming events are not only coming from higher level diagrams but also from child diagrams of the Atrial_Monitor. Modeling this PSPEC as a separate concurrent process is more straightforward since this creates a more clear interface to the processes that must send an event to the PSPEC.

In the H&P model all communications between DFDs, PSPECs and CSPECs run via data flows and control flows. The way the data flows are modeled happens to be tool dependent, hence we describe this aspect in the tool specific sections 5.2 and 5.3. On the contrary, for the control flows a modeling construct is designed that is used in the models for both tools. To motivate this control flow modeling construct, we consider the following example:

> Consider a system, consisting of two concurrent processes A and B that can interact by synchronous communication. Process A sends an event to process B, then performs n local actions and then terminates. Process B waits on any incoming event; on receipt of an event it processes this event by performing m local actions.

> When process A after sending the event to process B, simply continues its action sequence and ignores the behavior of process B, the resulting system state space will contain states for all combinations of internal states of process A and B. So in this case the order of magnitude of the model state space would be $n * m$, meaning that the state space size is quadratic in n and m.

Because the firmware design contains many such situations, we had to invent an event passing mechanism that avoids an explosion of the state space as described in the example. When we look at the implementation code of the pacemaker firmware, we see that event sending in the H&P model is actually implemented as doing a function call. Because the pacemaker firmware has no parallel threads, a component that calls a function must wait on execution of the function and can only resume its behavior when the program counter is set back to its address on termination of the called function.

> Back to the example this corresponds to the behavior in which process A performs its local actions after process B has finished processing the received event. If this behavior should be modeled, the order of magnitude of the model state space would be $n + m$, meaning that the state space size is only linear in n and m.

So to conform the formal models to the firmware design and for performance reasons, the sending of events is formally modeled as follows. Every process has its own unique ID and a process always sends its own ID and the ID of the recipient along with an event. After sending an event, the sending process waits on a special acknowledgment message that contains the ID of the event recipient and its own ID. The event recipient sends this acknowledgment message as soon as it finishes the processing of the received event. In this way, function calling and returning are modeled as separate events in the formal model. The event and acknowledgment messages with their arguments are modeled in mCRL2 and Uppaal as synchronizing parameterized actions or synchronizations over channels from an array, respectively.

An event from the PG that reaches the firmware system is firstly processed by the highest level process (corresponding to the highest level H&P diagram). This process sends the incoming event consecutively to one or more child processes as defined in the CSPEC of the highest level diagram. Each time the highest level process waits on acknowledgment while its child diagram processes the event. This behavior repeats itself on lower levels and usually the real processing of the event is mainly done in the lowest level processes by the PSPECs. This typical behavior is visualized in a message sequence chart (MSC) of an imaginary system in figure 5.1.

In this MSC we see that process Root and process A just redirect the event to child processes and wait for acknowledgment. Process B and C do some processing on receipt of the event and send an acknowledgment to the process that sent the event. On processing the event, process B sends an output to the environment.

Of course such modeling can be improved by for instance dropping process A as intermediate process in sending an event from Root to C, but in many cases we have not done this to stay close to the H&P model and to provide a clear interface to processes that must send an event to process A. In other cases we have applied this improvement; for example when a certain process always redirects all incoming event to one particular subprocess.

However this mechanism can cause problems if a certain process during processing of a received event sends an event to a higher level process that is currently waiting on an acknowledgment. The waiting process only accepts an acknowledgment from the process to

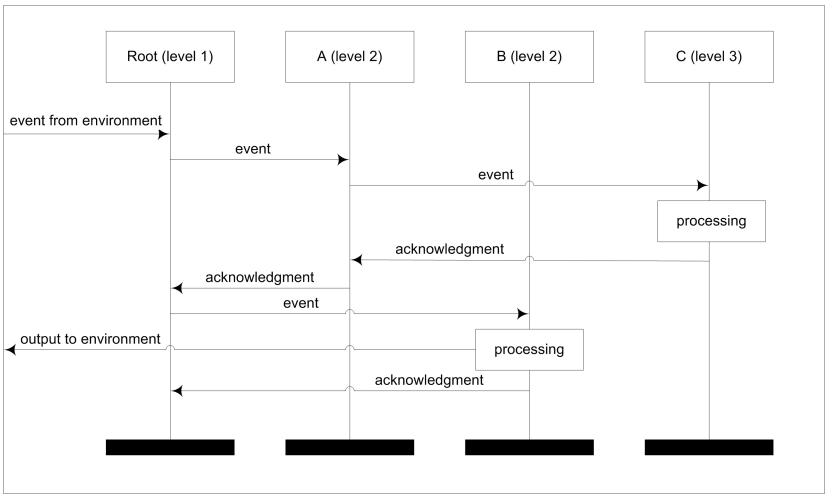


Figure 5.1: Message sequence chart of typical event passing in firmware system

which it has sent an event and thus will not be able to receive any events. This scenario will most likely cause a deadlock in the complete system, because the root process also does not get an acknowledgment and will not be able to receive any events from the environment anymore.

During construction of the models in this project, such scenarios did indeed cause a deadlock to occur when trying to generate the state space of the model. By inspecting the trace to the found deadlock, the scenario could be explored and the problem could be solved. Because the problem does not occur too often, it was possible to solve them by explicitly modeling the possible incoming events during waiting for an acknowledgment. When process A is waiting for an acknowledgment from process B and it receives an event from process C, it postpones processing of the event until the acknowledgment from process B is received. This prevents that process A and process B will be concurrently processing an event which can have bad consequences on the state space as we have seen above.

An H&P system requirements model can contain state transition diagrams (STD) in the context of a CSPEC (see section 4.1). These STDs differ from common state transition diagrams in the sense that they implicitly ignore events from incoming control flows if the STD does not include an outgoing transition from the current state for this control flow. So if an event is received when the STD resides in a state without an outgoing transition for this event, the STD just stays in this state.

In the formal models this aspect is also modeled. In mCRL2 this was just a matter of adding an else-branch to each process that sends an acknowledgment to the sender of an event for which no outgoing transition is defined in the current state. But Uppaal does not support such an else-branch construct so we had to add all non-defined transitions as loop transitions that send an acknowledgment to the event sender.

5.2 Technical aspects of the mCRL2 models

As described in section 4.1, data flows in an H&P model are time continuous and can be read and reread any number of times by a receiving process until the source of the data flow stops generating it. To model this aspect in mCRL2, for each data flow x of type T three parameterized actions $(r\_x, s\_x, c\_x : T)$ are declared. By means of the mCRL2 communication and restriction operators, it is specified that these three actions communicate and that only the action c_x is allowed to occur. The source process A of x contains a parameter representing the current value of x and A's process expression contains an alternative that performs the action s_x("value of x"). This enables each process that needs the value of x to perform $\sum_{y:T} r\_x(y)$ and use the value of y in subsequent actions and conditions as the value of the data flow x.

This modeling of the data flows only causes some problems when the source process is performing a series of actions in collaboration with other processes. During the execution the process is not able to perform the s_x action. An example of this problem is the situation in which process A has sent an event to one of its child processes B and waits on acknowledgment of B. While the event is being processed in lower level processes, the value of A's data flow may be needed. But A is waiting on acknowledgment and can not send the data flow value. This is a typical deadlock situation in which process A waits on the lower level processes and one of the lower level processes waits on process A.

One nasty solution for this problem could be enabling A to perform the data flow send action during waiting on an acknowledgment. But we have chosen to create a co-process in such cases that is responsible for storing the values of the data flows of process A, i.e. the data flows are modeled as a process. When for example, a calculation in process A results in a change of its data flow x, A performs a s_set_x("new value of x") action that communicates with a r_set_x action in A's co-process which stores the new value. The co-process only contains alternatives for receiving changes in A's data flows and for sending the current values to other processes. Because these are all unit-actions, the co-process can never be busy executing a series of actions and causing a deadlock situation as described above.

Next we discuss the modeling of time in the mCRL2 models. In the H&P firmware requirements model time is present through the minute_passed events from the hardware module Timer (see figure 4.2). In the part of the model that is formally modeled, these minute_passed events are actually only used in the process Band_Update that updates the physiological band (see section 4.2) at least every minute. In the processes that simulate the heart and PG behavior time is a more crucial factor, but here time steps of one minute are far too large. Because the PG does some processing every 5 ms, special tick-actions are introduced that represent a time passage of 5 ms.

All processes that use time (i.e. the heart and PG processes, the Band_Update process and also the root process of the firmware (see below)) can perform their own tick-action if their local state allows it. For example, if t represents the current time and the PG must deliver a pace to the heart if $t = 15$ then when $t = 15$ the PG may only perform its tick-action after delivering the pace. All the tick-actions communicate to one global tick-action which represents a global time passage of 5 ms. Because individual tick-actions are not allowed to occur independently of all others, global time can only pass if the states of all time dependent processes allow it.

As said above, the root process of the formal model also participates in performing the

tick-action. Not the fact that this process is time dependent is the reason for this, but this models the property that all processing in an H&P model is infinitely fast. All external events enter the formal model via this root process which redirects them to its subprocesses. Hence, the root process is always waiting for an acknowledgment of one of its subprocesses during processing of an event. When the root process is waiting until the event is entirely processed, it can not perform its tick-action. Therefore time can not pass during processing of an event in the formal model which means that this processing is modeled infinitely fast.

It must be noted that because event handling is modeled like this to conform to the H&P principle of infinitely fast event processing, flaws in the firmware implementation can potentially remain undiscovered. During processing of an external event, the firmware implementation of course spends runtime which could influence the later external behavior of the system. Since we only verify the firmware design, we stick to the infinitely fast modeling of event handling, but one has to be aware of the risk of missing flaws that are caused by spent runtime. Modeling this spent runtime will certainly not result in a feasible model checking approach, because this introduces too much non-determinism into the model.

5.3 Technical aspects of the UPPAAL models

In UPPAAL time is explicitly present through clock variables. All time dependent processes in the formal UPPAAL model are equipped with one or more clock variables. Because these clock variables automatically show synchronous progress, the only concern is to restrict this progress when time is not allowed to pass. In the previous section we described two situations in which time is not allowed to pass, i.e. if a process must perform some action sequence on a specific moment in time or when an external event is being processed in the model.

In the first situation the passage of time is restricted by a combination of a guard on the first transition of the action sequence and an invariant on the state wherein the process waits until it may perform the action sequence. The guard is a clock constraint which controls that performing the action sequence is not allowed before the clock reaches the time on which the action sequence must occur. On the other hand, the invariant is a clock constraint which specifies that the same clock may not be greater than the time on which the action sequence must occur. The invariant makes sure that the model can not reside in the corresponding state after the action sequence must have been performed. Hence the guard and the invariant together imply that the action sequence starts exactly on the specified moment in time and is performed infinitely fast.

To disallow passage of time during processing of an event, we make use of the notion of the committed locations (see section 2.2). Event processing is just an action sequence which includes redirection of events to other processes and waiting for acknowledgments. By making all intermediate locations in such action sequences committed, the action sequence must be finished once it is started before time can pass again. The same effect would be reached when we made all these locations urgent, but this would leave the option open to perform other (non time passing) actions during the processing of an event. This would affect the model state space size negatively and furthermore it could cause interference in the model when these other actions affect variables on which decision are made in processing the event.

Because synchronizations can not be parameterized in UPPAAL[1], data flows can not be modeled as this is done in mCRL2. Instead we use global variables for modeling data flows. The source of the data flow sends its output to a global variable which can be read by any receiving process. This is dangerous because the global variable can potentially be written by any process, but simplicity is the reason why this modeling has been chosen. Furthermore, to avoid unwanted writing to global variables, naming conventions are used in the UPPAAL models.

We encountered a type problem when modeling the hardware module PG in UPPAAL. On an atrial or ventricular event, the current values of the atrial timer and the ventricular timer must be stored in global variables (representing the data flows aa_int, av_int, va_int and vv_int, see chapter 4). The atrial timer and the ventricular timer are represented by two clock variables which are reset on each atrial or on each ventricular event, respectively. But storing of the current clock value in a global integer variable is not just a matter of assigning the clock value to the global variable, because this results in a typing error. Clock variables are real typed and UPPAAL does not provide a type casting function. To store the value of the clock variable in an integer variable anyhow, we created a committed state with looping transitions which increase the global variable as long as it is smaller than the clock value. Of course this has unfavorable effects on the model state space, because the system runs through a rather lot of intermediate states while performing these looping transitions. The effects on the state space are reduced by creating three kinds of these looping transitions which increase the variable with 1, 10, or 100 (each time only the maximal possibility is enabled).

5.4 Formal model of heart behavior and PG

As an environment in which the formal pacemaker model can be verified, a formal model is developed that simulates the heart behavior. This model is constructed such that it can show various rhythms, PVCs, retrograde conduction and crosstalk (refer to section 3.4 for these terms). Interaction between the pacemaker model and the heart model happens by sense and pace events between these models. The formal heart model has been developed through consultation with Vitatron.

This section briefly describes the heart model to indicate against which behavior the pacemaker model has been verified. Creating this awareness is important, because any heart model always is an abstraction of the reality. If the pacemaker model is found to be correct based on verification against this heart model, it can always be the case that flaws occur for heart behavior that is outside the scope of this heart model.

The formal heart model consists of two processes that simulate the behavior of the atrium and the ventricle and 5 processes that together simulate the behavior of the AV node. In figure 5.2 an overview of the heart model is given. The left hand side of this figure shows the architecture of the heart processes where the possible events between the processes are represented by dashed lines. The right hand side shows the behavior of one single AV node process in a kind of state transition diagram that partially depends on time (its meaning is described further on).

[1]Parameterization of synchronizations can be simulated by introducing arrays of synchronization channels, but this is rather awkward.

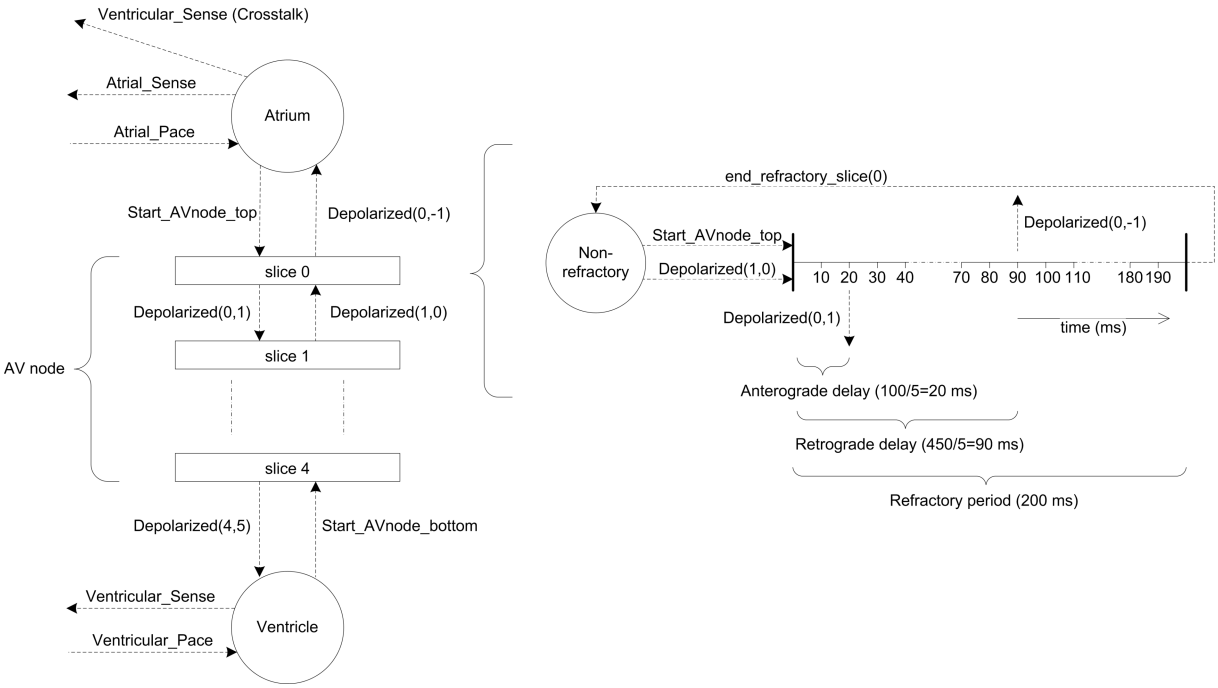


Figure 5.2: Formal heart model overview

The atrium can contract upon receipt of an atrial pace from the PG, upon receipt of a stimulus from the AV node (simulating retrograde conduction) and it can generate a spontaneous contraction to simulate the function of the sinus node. After each contraction, the atrium process determines when its next spontaneous contraction will take place by making a non-deterministic choice from a number of rates; this spontaneous contraction will of course only take place if no stimulus from the PG or the AV node arrives before the determined moment in time. Upon each contraction an event is sent to the AV node processes to start the conduction to the ventricle and the atrial refractory period is enabled in which all external stimuli to the atrium are ignored. If the contraction was not caused by an atrial pace, an atrial sense event is sent to the PG. Atrioventricular crosstalk is modeled by allowing the atrium process to possibly send a ventricular sense event to the PG shortly after an atrial pace is received (a ventricular sense is classified as crosstalk ventricular sense if the PG receives it within 90 ms after the atrial pace has been delivered).

The ventricle process behaves analogously, except that crosstalk does not have to be addressed in this process and that it only has two choices for its spontaneous contraction rate. One choice is a very low rate, which corresponds to the normal heart behavior in which the ventricle does not show spontaneous contractions. The second choice is a relatively high rate to enable the ventricle process to show PVCs.

Next we address the processes that simulate the behavior of the AV node (refer to section 3.2.2 for a purpose description of the AV node). Each process represents a slice of the AV node and is responsible for delaying all conduction signals that travel through the AV node. The total delay that a conduction signal encounters in anterograde direction (100 ms) is smaller than the delay in retrograde direction (450 ms) which is according to the natural behavior of the AV node (although these values can vary in the heart's natural behavior, we have modeled

them as constants to limit the state space size). During delaying a signal and a period here after each slice is refractory meaning that any incoming signal is not handled.

This behavior is illustrated in the right hand side of figure 5.2: the slice is non-refractory until any external stimulation arrives (either from another slice, the atrium or the ventricle). Upon receipt of an external stimulation, the slice starts its anterograde delay; after this delay the stimulus is 'conducted' to the slice below (or the ventricle in case of the lower slice). Then the retrograde delay starts after which the stimulus is send to the slice above (or the atrium in case of the upper slice). Finally the slice remains refractory for a while after which it returns to its non-refractory state.

The AV node is modeled in several slices to take the behavior of the natural AV node in the following situation into account:

> The atrium contracts and triggers the AV node (at the upper side) and before the signal is conducted to the ventricle, the ventricle also contracts and triggers the AV node (at the lower side). In this situation two conduction signals approach each other in the AV node. On the moment that the two signals meet each other in a cell somewhere halfway the AV node, they will extinguish. This happens because the first signal that arrives at this cell makes this cell refractory which causes that the secondly arriving signal will not be conducted anymore.

In reality the AV node contains too many cells to model them all as a separate process. Therefore it is chosen to model 5 slices which all represent one fifth of the AV node cells.

As described in chapter 4, the PG is the interface between the pacemaker leads and the firmware. Therefore the PG is also formalized in the models that contain the heart model. The atrial and ventricular timer are modeled as two parameters in the PG (by clock variables in Uppaal and by naturals that are increased on every tick-action in mCRL2).

Here some text is removed for confidentiality reasons.

This text describes how some PG details are formally modeled.

5.5 State space reducing modeling techniques

During the actual model checking several adaptations were made to the models in order to limit the state space. These state space reductions emerged from a kind of iterative procedure; each time the model state space size appeared to be too large we tried to invent a reduction that would be helpful in the specific model. The adaptations vary from just changing some constants to adding transitions and to making the ordering of events more strictly. This section clarifies these reductions and explains why they are justified, i.e. why the behavior of the model still conforms to the behavior of the modeled system.

The H&P model of the pacemaker firmware specifies that a command for an atrial pace is given on the detection of a PVC (in case of a sequence of PVCs this is only done for the

first PVC in the sequence). As described in section 3.4 this PVC-A stim is given to prevent that the ventricular contraction will be retrogradely conducted to the atrium. This is formally modeled by allowing the process PVC_Astim_Generator to send a message to the PG that upon receipt of this message will directly deliver an atrial pace.

Furthermore the accompanying document of the H&P model specifies the following:

> "The time between the PVC and PVC-A stim has to be less than 40 ms. Note: The specified value of 40 (ms) is a maximum. The implementation should strive for an as short as possible delay in order to optimize the effectiveness of this feature."

Initially this was modeled by allowing an arbitrary time passage (but less than 40 ms) after the process PVC_Astim_Generator has decided that a PVC-A stim must be given. This resulted in an enormous state space explosion, because each possible time passage between the PVC and the PVC-A stim caused a different value to be stored by the PG in va_int and aa_int. Although the precise moment in time on which the PVC-A stim is given does not have great influence on the behavior of the system, the many different values of va_int and aa_int that are stored in the model state result in an explosion of states. Therefore we skipped the implementation of the 40 ms delay between the PVC and the PVC-A stim and just give the command for the PVC-A stim directly after detecting the PVC. Because the firmware implementation should strive for an as short as possible delay, the formal model does not deviate from the specified system too much in this way. But it is important to realize that this is a real simplification of the formal model which potentially can hide certain flaws in the firmware design that are caused by the time interval between a PVC and the PVC-A stim.

Recall from section 4.1 that in a process activation table (PAT) in an H&P model the order of triggering PSPECs and child DFDs and sending output events (PAT-consequences in the sequel) can be specified. Figure 4.4 shows that this order is specified by numbering the PAT-consequences. But often a PAT does not specify the exact order of the PAT-consequences by numbering them all with the same number. This is usually done because the order does not matter for the behavior of the system. Then the choice for a specific order is just left open for the ones that implement the system.

Again in a try to deviate as less as possible from the specified system, we modeled such 'unordered' PAT-consequences by triggering (sending the event or executing the PSPEC's pseudo-code) these processes all consecutively, followed by waiting on an acknowledgment message from all these processes. But as you can imagine, the fact that a number of processes can be processing an event simultaneously, results in many possible interleavings in the model state space and hence in an explosion of the state space.

To reduce the state space we decided to fix the ordering of the PAT-consequences according to the choices that are made in the implementation code of the firmware system (the implementation of the firmware system must choose a specific ordering because the system has no parallel threads). Because the model checking purpose is verification of the system specification and not of the system implementation, this is not preferred. It could be the case that the choices that are made in the translation of the system specification to its implementation accidentally have made the system behavior satisfying its requirements, while other choices would have resulted in incorrect system behavior. But keeping this in mind, we have made this decision to be at least able to verify a model that corresponds to the system implementation.

Next, we applied some state space reductions in the processes that are responsible for detection and handling of *retrograde conduction*. The DFD VP_AS_Pattern maintains two counter variables; Total_PRAS_cntr that counts the number of possibly retrogradely conducted atrial senses and Successive_VP_cntr that counts the number of successive ventricular paces. Based on the values of these counter variables this process might decide that retrograde conduction is probably present and can start the RC_Test therapy in order to verify this and to handle appropriately. Basically said, this decision is made if Total_PRAS_cntr crosses a certain threshold value while Successive_VP_cntr is yet under another threshold value.

The factory settings of these two mentioned threshold values are 26 and 32, respectively. Therefore it takes a rather large sequence of atrial senses and ventricular paces before the RC_Test is started, which results in many intermediate states in the state space. By decreasing the threshold values, the amount of intermediate states can be reduced which also reduces the state space significantly.

In the occurrence of a certain event sequence[2] the counter variable Successive_VP_cntr keeps being increased without ever checking whether it already crossed the threshold value. If this event sequence holds on infinitely Successive_VP_cntr is also being increased infinitely, resulting in a infinite state space. This is highly undesirable, so we have added an upper bound in the specification of VP_AS_Pattern which avoids that Successive_VP_cntr becomes much greater than the threshold value.

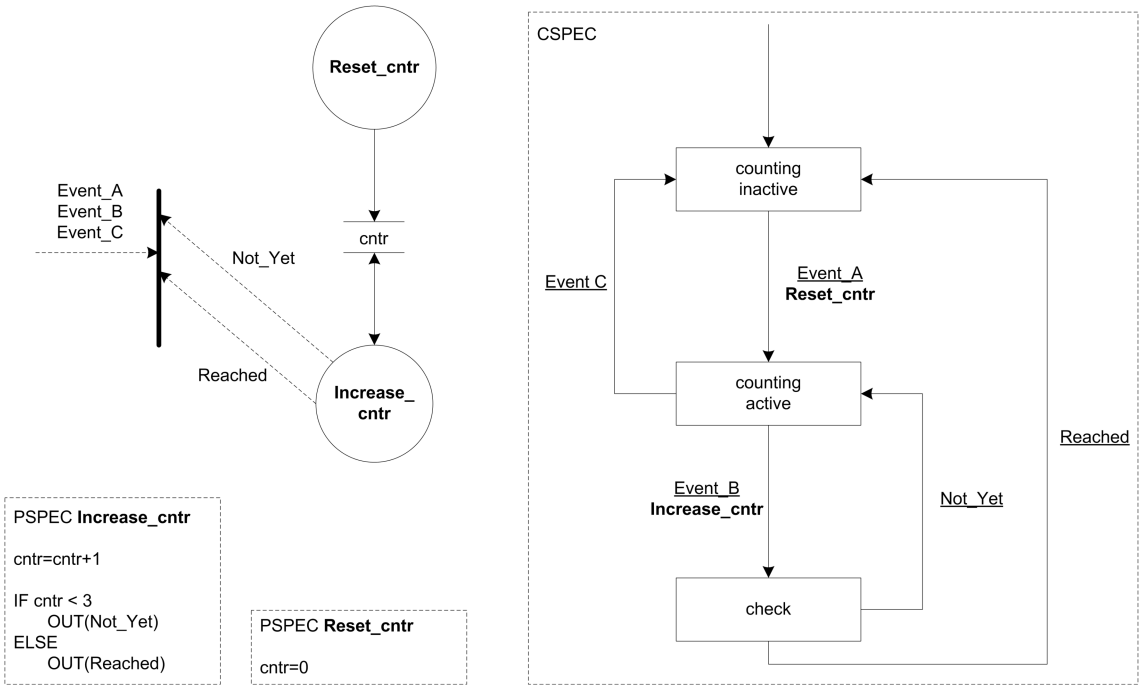


Figure 5.3: STD structure of Multiple_Successive_VS and NOA_Detection

[2]A sequence of alternating ventricular paces and PFASs is present. PFAS stands for Potential Farfield Atrial Sense, which means that the time interval between this atrial sense and the preceding ventricular event is too short to be conducted retrogradely.

The DFDs Multiple_Successive_VS and NOA_Detection both maintain a counter variable and are driven by a state transition diagram which is similar for both processes. The structure of these state transition diagrams is shown in figure 5.3. Note that the labels on a transition in the state transition diagram define the received action above the bar and the triggered PSPEC below the bar. The word 'cntr' between two horizontal lines represents a data store, containing a counter variable.

The counter variable is only used when the STD is in the state 'counting active'; in this state the number of received 'Event_B' events is counted and on the third 'Event_B' the STD moves to state 'counting inactive'. Although the counter variable is not used in state 'counting inactive', it is not reset upon the transition to this state. Hence the last used value of the counter variable stays in the variable, which has negative consequences on the state space. Potentially the size of the state space is multiplied by 4 for all 4 values that the counter can have; because there are 2 such processes, this factor can even become 16.

Therefore we added a reset of the counter variable to the transitions to state 'counting inactive'. So both on receipt of 'Event_C' as on receipt of the third 'Event_B' in state 'counting active' the counter is reset. Because the nature of Multiple_Successive_VS and NOA_Detection implies that they are only counting in a limited part of the state space, the values of their counter variables only differ from 0 in this part of the state space. Experimentally it has been encountered that this adaptation reduces the size of the state space significantly, while the behavior of the model clearly is left unchanged.

As described in section 4.2, the Atrial_Monitor updates a physiological band on every incoming atrial event and the Ventricular_Monitor updates a moving average of the ventricular rate on every incoming ventricular event. This causes that a lot of information about the rate history is stored in the state space which has dramatic results on the size of the state space. To limit these results the following changes have been made:

Normally the physiological band is increased or decreased in steps of 2 if the current atrial rate is higher or lower, respectively than the current physiological band. This step size is now increased to 20 so that the number of possible band values decreases. In this way the band still follows the current atrial rate; although this happens a lot coarser now, incoming atrial events can still be classified based on the physiological band.

The value of the ventricular rate average has been fixed, which is of course the best reduction that can be applied on a varying parameter. The only decision that is based on the average ventricular rate is whether an incoming atrial sense should be tracked or not; by fixing the average ventricular rate this tracking decision is made less history dependent, but because the tracking decision is also based on the physiological band the behavior will not be radically changed. The value to which the average ventricular rate has been fixed is 110, which is chosen to enable the occurring of repeatedly retrograde conduction; if the value would be lower, any retrogradely conducted atrial sense would not be tracked by ventricular paces which would stop the retrograde conduction sequence.

Chapter 6

Model checking approaches and results

As has already been mentioned in chapter 5, several model checking approaches have been exploited to investigate the feasibility of model checking on Vitatron's firmware designs. In this chapter, these approaches are described together with the results and experiences that were found by using the approaches. Note that we only describe verification results here; the development of the formal models itself can also be regarded as a result, but this has been described in chapter 5. Because any model checking approach comes down to applying abstractions and eliminations to certain parts of the system behavior, we indicate for each approach what the reliability level is of any results that potentially can be obtained by using this approach.

Because the duration of the project did not permit us to translate the complete firmware design to a formal model, no approach is exploited that verifies the complete firmware design. In order to obtain a good perception of the feasibility of model checking at Vitatron, it is at least necessary to verify a model that contains the most characteristic firmware design elements, the firmware design elements that have the largest influence on the external behavior of the firmware, and firmware design elements with a large flaw risk. Elements with a large flaw risk are elements with a high complexity (e.g. firmware parts consisting of many collaborating and interdependent processes) through which it is hard to evaluate their behavior 'manually' by exploring the firmware design specification. Which specific approaches have been exploited is mainly invented through consultations with Vitatron.

These consultations for example resulted in the decision that we omitted including many pacemaker therapies (pacing rate controlling subprocesses of The_Umpire that can be switched on or off by the patient's cardiologist; see section 4.2) in the formal model. Although verifying the complete firmware process The_Umpire would be a very interesting approach because this is a complex process that decides which of the 'competitive' therapies controls the pacing rate, we omitted this approach in favor of being able to verify other parts of the firmware design. Instead of modeling the complete process The_Umpire, only the Lower_Rate therapy has been modeled that maintains a certain minimum rate under all circumstances.

Furthermore the decision has been made to abstract away from all pacemaker parameters that are programmable by the patient's cardiologist. In reality most of these parameters can vary within some range (taking interdependencies into account). It could be the case that the firmware behavior contains a flaw for some extraordinary parameter configuration; to verify

the firmware extensively, one should have to vary all these parameters within their ranges and verify the formal model for each combination of the parameters. But this would have dramatic effects on the model state space; in fact the state space would be multiplied by the number of possible parameter value combinations. In this project we have chosen to set all pacemaker parameters to their factory settings in order to investigate model checking on the default firmware design. How model checking could be improved for systems with many programmable parameters might be a topic of future research.

Before we describe the three model checking approaches and their results, it must be noted that two of the three approaches were not fully investigated in UPPAAL. Halfway in the project, we experienced relatively large verification times for the verifications in UPPAAL, compared to the same verifications in mCRL2. This was even the case for simple models with state spaces of a few millions of states (if the state spaces of these models were generated by mCRL2). Because UPPAAL does not give any information about the number of already verified states during the verification, it was very hard to find reasons for these large verification times. At a certain stage of the formal modeling, all optimizations were invented through analysis by the mCRL2 toolset. Carrying these optimizations and other changes in the mCRL2 models through in the UPPAAL models was a time intensive job considering the lack of results that we achieved via verification of the UPPAAL models. These considerations made us decide to stop modeling in UPPAAL and to focus completely on verifications of the mCRL2 models. However, we have reproduced one important result from verification with mCRL2 in UPPAAL; this is described in section 6.4.2. Note that we do not have completely turned down UPPAAL as a model checking tool for pacemaker firmware by making this decision. The prior knowledge of mCRL2 and the available support from the mCRL2 developers might have influenced the comparison between the two model checkers. Whether the use of UPPAAL as a model checker for firmware designs could be improved might also be a topic of further research.

6.1 Verified requirements

It has appeared to be rather hard to invent appropriate requirements to the firmware design which were suitable to be verified by means of model checking. Because we had only limited knowledge of the pacemaker firmware, the verified requirements have been invented through consultations with Vitatron. This resulted in only three requirements to the complete firmware model; these requirements are verified in the approaches that are described in section 6.2 and section 6.3. Note that this set of requirements is by no means complete, but at least it is a representative subset of all kinds of requirements. Verifying these three requirements will probably give us a clear picture of the feasibility of verifying requirements to the formal firmware model in general.

Besides these three requirements to the complete firmware model, it would probably be more easy to invent requirements to parts of the firmware design. It is desirable that each component of the firmware design is designed with a clear and specific purpose. If this purpose is well documented, model checking approaches can be developed in which it can be verified whether the design component indeed accomplishes this purpose. We describe one example of such an approach in section 6.4.

Next we describe the three requirements to the complete firmware model, followed by

their formalizations that are used in the verifications with mCRL2 and UPPAAL:

- **Absence of deadlocks:**
  Absence of deadlocks is of course essential for the correct behavior of the pacemaker firmware, but besides that we experienced that modeling errors often cause deadlocks to occur in the models. E.g. if a process in the formal model contains a flaw so that it tries to send an event to another process that can not receive that event, a deadlock state will be reached because the first mentioned process can not proceed. So checking whether the formal models are deadlock free also acts as a debugging facility, although the absence of deadlocks is of course no guarantee for the absence of modeling errors.

  mCRL2:
  Verifying this requirement on the mCRL2 models can either be done during state space generation by `lps2lts` or by formulating the simple logic formula $[true^*]\langle true\rangle true$ which can be verified by the tool `pbes2bool`. If the first method is used, `lps2lts` must be instructed to print a message for every deadlock that it detects in the state space. We have used this `lps2lts` method, because we ran `lps2lts` anyway to determine the size of the model state space.

  UPPAAL:
  Note that section 2.2.3 mentions the special property `A[] not deadlock` in UPPAAL's requirement specification language that can be used to verify this requirement on the UPPAAL models.

Here some text is removed for confidentiality reasons.

This text describes the 'second requirement' to the complete firmware model.

- **High-Rate-Limiter (HRL):**
  On a higher level, the external behavior of the pacemaker can be observed by inspecting the behavior of the heart. The pacemaker must maintain an adequate heart rate; one of the basic requirements to this heart rate is that it may not become too high. In other words, the time interval between two ventricular contractions must be higher than a certain threshold value called the High-Rate-Limiter (310 ms). Because this is a requirement to the ventricular behavior, this requirement has only been verified in those models that contain the formal heart model.

  mCRL2:
  The ventricular contraction itself does not have a distinctive action and can furthermore have a number of causes. It can be a contraction that is caused by anterograde conduction of an atrial contraction, a contraction that is caused by a ventricular pace, or a spontaneous contraction of the ventricle itself (a PVC). Therefore we have formulated a logic formula which specifies that the time interval between two *end_refractory_v* actions must always be larger than the High-Rate-Limiter. The *end_refractory_v* action marks the end of the refractory period after each ventricular contraction; because the

refractory period has a constant length, the time interval between two such actions equals the time interval between the corresponding contractions.

To formalize this requirement, we need the modal μ-calculus extended with data, because the time interval between two consecutive *end_refractory_v* actions must be 'measured' after which it can be compared to the High-Rate-Limiter. Unfortunately we experienced serious difficulties in obtaining a correct logic formula representing this requirement. Therefore we based it on the following example from section 4.4 of [11]:

> "A very powerful feature is the ability of putting data as parameters in fixed point variables. Using this it is possible to for instance count the number of events. Saying that a buffer may never deliver more messages than it has received can be done as follows:
>
> $\nu X(n : \mathbb{N} := 0).[\overline{deliver \cup receive}]X(n) \wedge [receive]X(n+1) \wedge [deliver](n > 0 \wedge X(n-1))$
>
> Here n counts the number of received messages that have not been delivered yet. The notation $n : \mathbb{N} := 0$ says that n is a natural number that is initially set to 0. The core of the formula is in its last conjunct, which is false if a deliver is possible while $n > 0$. This conjunct then becomes false, turning the whole modal formula into false."

Inspired by this example formula, we created the following modal formula, saying that an *end_refractory_v* action may only occur when at least 62 *tick_ALL* actions (which equals 310 ms, because every *tick_ALL* action represents 5 ms) have occurred since the preceding *end_refractory_v* action.

$$\begin{aligned}\nu X(n : \mathbb{N} := 0).&[tick\_ALL]X(n+1) \wedge \\ &[end\_refractory\_v](n \geq 62 \wedge X(0)) \wedge \\ &[\overline{tick\_ALL \cup end\_refractory\_v}]X(n)\end{aligned}$$

Because we have based this formula on the mentioned example, we have high confidence that the formula is a valid representation of the HRL-requirement. Nevertheless, it is unsatisfactory that we were not able to invent a formula from scratch. From our opinion, this is mainly due to the complexity of the modal μ-calculus, which will probably also give some trouble if Vitatron would decide to use mCRL2 as a model checking environment.

We have verified this logic formula on the mCRL2 models by the tool `pbes2bool`.

UPPAAL: Since there are no tick-actions involved in the UPPAAL models we can not use the above described formula to verify this requirement in these models. Instead we have created a formula that depends on the clock variable of the `Ventricle` process:

`A[]((Ventricle.Conducted_contraction` $\vee$ `Ventricle.PVC` $\vee$ `Ventricle.Pace)` $\Rightarrow$ `Ventricle.c`$\geq$`62))`

Literally this formula states that the value of the clock variable `c` in the process `Ventricle` must always be at least 62 if the process is in the location `Conducted_contraction`, `PVC`, or `Pace`. Note that these three locations all indicate that a ventricular contraction has just passed and that `c` is reset upon leaving these locations. Hence the value of `c`

in one of these three locations represents the length of the time interval since the last ventricular contraction. Upon each contraction this time interval is compared to the HRL value.

6.2 Heart model approach

The main approach that has been exploited during the project consists of verifying the formal firmware model in the presence of models that simulate the heart and the hardware module PG; these environmental models are described in chapter 5. As has already been indicated briefly in that chapter, any results that are obtained by verifying the firmware model in the presence of this heart model must be judged by keeping the modeled heart behavior in mind.

Our heart model can show some of the mostly occurring arrhythmias (sinus tachycardia, sinus bradycardia, PVCs, retrograde conduction and crosstalk; see section 3.4) besides the natural heart behavior, but the various types of AV block and more rare types of arrhythmias are not included in the heart model. If the firmware model contains a flaw which only results in undesired external firmware behavior when the heart would show behavior that is not modeled, we certainly can not detect this flaw by verifying in the presence of this specific heart model.

The modeled heart behavior strongly influences the state space size of the verified model (the composition of the heart model, the PG model, and the firmware model); each modeled arrhythmia introduces non-determinism into the heart model. E.g. after each received atrial pace, the formal atrium process non-deterministically decides whether or not the atrial pace results in a ventricular crosstalk sense that is sent to the PG within 90 ms after the pace (see section 5.4). Therefore the modeled heart behavior must be limited to make model checking feasible. The behavior that is present in our heart model has emerged from taking its effects on the state space into consideration and again through consultation with Vitatron.

Recall that the atrium process in the formal heart model non-deterministically chooses its new rate after each contraction. By allowing a choice for a very slow rate, sinus bradycardia is modeled and likewise a very high rate models sinus tachycardia. In the approaches that include the formal heart model, we have varied the possible atrial rates that are specified in the atrium process to test their effects on the state space size and the model behavior. Note that we have left the conduction time intervals through the AV node (anterogradely 100 ms and retrogradely 450 ms) constant; also varying these time intervals would again mean a significant impact on the state space size.

In this main approach the following firmware processes are included (refer to section 4.2 and figure 4.5 for a more detailed description of the mentioned firmware processes):

- The processes Accept_R_Channel and PVC_Astim_Generator that are responsible for detecting and handling crosstalk ventricular senses and PVCs, respectively.
- The_Umpire is included as a driver process that only provides some parameters to other processes in order to maintain the minimum rate of the Lower_Rate therapy.
- The functionality of the process Escape_Timing that is applicable for DDD pacemakers.
- The processes AV_Monitor and Ventricular_Monitor are entirely included in the model.

- The Atrial_Monitor is included, except the Tachy_Classifier that provides a detailed classification of too high atrial rates. In the formal model, this classification has been simplified and included in another subprocess of the Atrial_Monitor that is responsible for classifying all atrial rates. Furthermore the RC_Monitor is omitted from the Atrial_Monitor in this approach.

6.2.1 mCRL2 results

On the mCRL2 model[1] of this approach, we have verified all requirements from section 6.1 for several combinations of rates that can be shown by the atrium process. Basically we can conclude that all verified models satisfy all formulated requirements.

Unfortunately, we were not able to verify the model for all combinations of atrial rates, because the state space size exploded dramatically when certain combinations of rates were possible. Because a model must first have a reasonable state space size before any requirement can be verified on the model, it is probably more interesting to consider how the state space size relates to these atrium configurations.

Being aware of this relationship might enable us to draw conclusions about the feasibility of model checking firmware designs by this heart model approach. The atrial rate corresponds to the sinus rate in the human heart which has the greatest influence on the complete heart behavior. Therefore the test coverage of this model checking approaches is directly proportional to the number of possible atrial rates. To maximize the test coverage of the firmware design the atrium process must be able to perform spontaneous contractions in rates that vary from very low to very high, but of course the state space explosion problem is inevitable here.

Possible atrial rates (bpm)	Model state space size (# states)	HRL verified
40/80/120	24.539.395	yes (valid)
40/60/80/100/120	43.837.681	yes (valid)
40/80/120/160	476.503.913	no
140/180	116.692.481	no
80/180	3.865.364	yes (valid)

Table 6.1: State space sizes of some mCRL2 models

Table 6.1 shows the state space sizes of some model configurations. What immediately draws our attention in this table is the fact that allowing high rates causes a dramatic increase to the model state space size, although that is only the case if the gaps between the possible rates are not too large. An explanation for this is that information about the current rate is stored in variables that represent the physiological band on which tracking decisions[2] are made. As described in section 4.2 this band is only updated for limited rate changes to avoid too sudden changes in the ventricular rate. The physiological rate is initialized to the minimum rate of 60 bpm that is maintained by the Lower_Rate therapy. In the model that shows atrial rates

[1] This model can be found in the electronic appendix: `mCRL2/Heart_model_approach_section_6.2.mcrl2`

[2] An atrial sense is tracked by planning a ventricular pace a certain time interval after the sense. This is only done if the atrial rate is found to be appropriate by the firmware.

of 80 and 180 bpm, the 80 bpm senses will cause the physiological rate to become 80. Any 180 bpm atrial sense that is shown by the heart model does not cause an increase of the band, because the jump of 100 bpm is far too sudden. Therefore the value of the physiological rate is 80 bpm in almost the complete state space which explains the relatively small size of the state space.

On the other hand we see that the state space of the 40/80/120/160 model is quite large; here the gaps between the possible rates are smaller and probably the physiological rate can become any value between 40 and 160 bpm in the model state space. For every value of the physiological rate, other tracking decisions are made for the four possible atrial senses which again enlarge the state space.

But this latter explanation still does not explain the difference in state space size between the 40/60/80/100/120 model and the 40/80/120/160 model. In the former model the physiological rate can become any value between 40 and 120 bpm, but the consequences on the state space size are not just as much as in the latter model. Most probably this difference is explained by the fact that the pacemaker firmware performs a lot of calculations in the presence of high atrial rates; planned paces are sometimes advanced in time to ensure that safety critical paces can still be given if needed. These calculations determine new values for variables which influences the behavior of the model in handling later events. Any new variable value possibly multiplies the state space and in combination with the possible atrial rates this soon leads to an exploding state space as we have experienced. Note that this effect can also be seen in the state space size of the 140/180 model.

On the first two models in table 6.1, in which the atrium can show a range of rates from very low to 'average', all three requirements are verified completely and found to be correct. Unfortunately, we were not able to verify the model completely for atrial rates that vary from very low to very high. We also tried to verify the 40/60/100/140/180 model, but we killed its state space generation after the generation of more than a billion states (which took more than a week on a machine with 128 GB of memory). Based on this fact and the state space size of the 40/80/120/160 model, the state space size of the 40/60/80/.../180 model (providing an even more complete range of atrial rates) would probably be extremely large, thus making model checking of such a model infeasible.

In this approach the process The_Umpire has been included as a driver process that provides the so-called escape interval of the Lower_Rate therapy which maintains a minimum rate of 60 bpm. To investigate the feasibility of extending The_Umpire such that it can provide escape intervals for more different rates, we adapted the driver process of The_Umpire (in the 40/80/120 model) slightly such that it non-deterministically would deliver the escape intervals for a rate of 60 bpm or a rate of 100 bpm. To verify the behavior of the firmware design extensively, many more options must of course be provided, but as a start we considered these 2 options. Unfortunately, the state space of the resulting model exploded dramatically again (more than 500 million states), thus making clear that varying these escape intervals in this approach is infeasible.

Another point of concern that we have about the large state spaces that are generated by this approach is that visualization techniques can not be applied on these state spaces. The mCRL2 toolset contains a set of visualization tools which have been proved to be rather useful in several investigations in the past. By visualizing the state space of a model, it can be visu-

ally explored. By trying to explain the visible patterns in the state space one can potentially obtain much insight in the modeled system and in the consequences of the made choices while formally modeling the system. This insight can again lead to improvements to the model and to the discovery of possible state space reductions that can be applied. Unfortunately using these visualization tools is only feasible to state spaces up till approximately 1 million states.

6.2.2 UPPAAL results

Because we prematurely stopped modeling in UPPAAL, some parts of the UPPAAL model[3] for the heart model approach do not entirely conform to the corresponding mCRL2 model. The mCRL2 model contains a more elaborated version of a subprocess of Escape_Timing whereas the UPPAAL model contains a depleted version of this subprocess. Nevertheless we have verified the UPPAAL model, because this difference between the models certainly does not make the state space of the UPPAAL model bigger than the state space of the mCRL2 model. We have chosen to verify the UPPAAL model for possible atrial rates of 40, 80 and 120 bpm because the state space of the mCRL2 40/80/120 model was relatively small. Note that we have only verified the 'absence of deadlocks'-requirement and the HRL-requirement.

UPPAAL's verifier reported that the 'absence of deadlocks'-requirement holds on the model after approximately 5,5 days of verification time (compare this to the verification time of a few hours that the mCRL2 toolset needed). The HRL-requirement was validated on the model after approximately 14 days of verification time, which is again a rather lot more than the verification time that mCRL2 needed to verify the requirement (approximately 1 day).

These large verification times for this relatively 'simple' model explain the reason why we stopped modeling in UPPAAL on a certain moment during the project. Although model checking firmware designs is certainly possible by using UPPAAL, our investigations have shown that mCRL2 can verify more complicated models while requiring less runtime.

6.3 Heart model approach extended with RC functionality

The H&P firmware design contains a rather cohesive group of processes that is responsible for the detection and the handling of retrograde conduction (RC; see section 3.4). Because the group of collaborating processes is rather large and this group is a typical example of the firmware design style, Vitatron suggested including this RC functionality in the verifications.

Therefore we extended the heart model approach with the processes from the RC_Monitor and the RC_Test. The RC_Monitor is a collection of subprocesses of the Atrial_Monitor that constantly monitors all incoming senses and paces to check whether the RC_Test should be started; this is the case when a certain amount of ventricular paces and suspected retrogradely conducted atrial senses (atrial senses that are received within 450 ms after a ventricular event) have been received. The RC_Test is a pacemaker therapy which has been modeled as a subprocess of The_Umpire. If the RC_Test therapy is active, it alters the AV-delay between the atrial sense and the ventricular pace that tracks this sense in order to verify whether retrograde conduction is really present. The RC_Test reports its findings to the RC_Monitor again, so that the RC_Monitor can classify each incoming atrial sense as being a normal atrial

[3]This model can be found in the electronic appendix: `Uppaal/Heart_model_approach_section_6.2.xml`

sense or a retrogradely conducted atrial sense. This classification is used by other processes like Escape_Timing in combination with the other classifications from the Atrial_Monitor.

Because through the addition of the RC_Test there appears more than one therapy in the formal model, The_Umpire had to be extended as well. The priority of the RC_Test therapy is always higher than the priority of the Lower_Rate therapy, hence we modeled The_Umpire so that it outputs the Lower_Rate therapy data if the RC_Test is inactive and the RC_Test therapy data if the RC_Test is active.

The reliability level of this approach is similar to the one of the previous approach, because we verify in the environment of the same heart model. Hence once again the results that possibly are obtained by this approach must be judged by taking the modeled heart behavior into account.

Although we have produced an UPPAAL model[4] for this approach, we have only verified the models in this approach by mCRL2, because of the limited UPPAAL results in the previous approach.

6.3.1 mCRL2 results

Verification of the mCRL2 model[5] that correspond to this approach actually shows the same patterns as the verifications in the heart model approach from the previous section:

The successful verifications (for possible atrial rates that vary from low to 'average') of the heart model approach are repeated in the models of this extended approach; this has shown us that all verified models satisfy the three requirements.

Furthermore, the model state spaces are even dependent on the possible atrial rates, allowing high rates also causes a dramatic increase of the state space size in this approach. The sizes of the state spaces are multiplied by a factor 2 due to the addition of the RC functionality. Because this is only a constant factor, this extended heart model approach is just as much feasible as the heart model approach from the previous section:

For possible atrial rates that vary from low to 'average', the firmware design can be verified by this approach, but providing a more complete heart behavior by allowing higher atrial rates soon makes the approach infeasible, because the state space size increases too much.

6.4 Isolated RC functionality approach

Because the RC functionality is such a typical example of collaborating processes in the firmware design and because Vitatron indicated that in the past a flaw had been found in this part that can cause a deadlock[6], we investigated the application of model checking on this RC part separately. This means that we have constructed a formal model that besides the RC_Monitor and RC_Test processes only contains some driver processes and stub processes that represent the processes in the context of the RC part. The driver processes provide the

[4]This model can be found in the electronic appendix: `Uppaal/Heart_model_RC_approach_section_6.3.xml`

[5]This model can be found in the electronic appendix: `mCRL2/Heart_model_RC_approach_section_6.3.mcrl2`

[6]Vitatron has successfully identified the root cause of this problem. The problem occurs only for very specific parameter settings and therefore the probability that this problem occurs is very small. Note that the problem has been solved successfully; also all already implanted pacemakers with this problem have been updated to fix the problem.

RC part with its input events and input variables and the stub processes accept its output events. In its normal context in the firmware design, the RC part processes 7 different input events and needs one time interval variable from the PG; see figure 6.1.

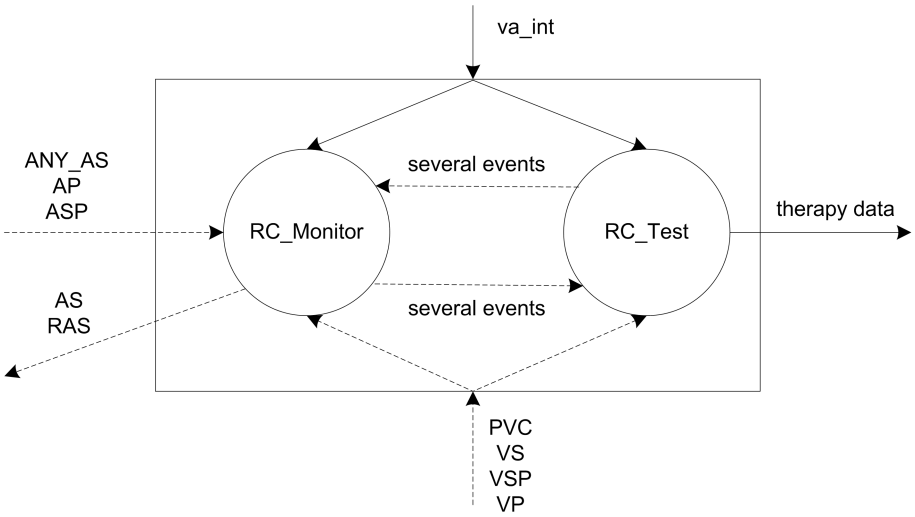


Figure 6.1: Context overview of RC part

One of the driver processes repeatedly sends one of the 7 possible events to the RC part. Some of these events only must be sent to the process RC_Monitor and others must be sent to both processes RC_Monitor and RC_Test. After sending an event to RC_Monitor or RC_Test, the driver process waits on an acknowledgment message from this process as to simulate the original sender of the message in the complete firmware model. After handling an event completely (the acknowledgment message is received from the processes to which the event has been sent), the driver process non-deterministically decides the next event it shall send. In this decision some dependencies between the possible events are taken into account, e.g. according to its definition a PVC can only occur if the previous sent event was a ventricular sense, a ventricular pace or also a PVC.

The external variable va_int is provided by a separate driver process. In practice, this variable can have a lot of different values (any value between 0 and approximately 1000 ms in steps of 5 ms), but we have limited the number of possible values in the formal model to reduce the model state space size. Because va_int is compared to two thresholds (a lower threshold of 150 ms and an upper threshold of 450 ms), we have modeled a possible value of va_int for each of the three possible equivalence classes of the variable. So va_int can be 100 ms (lower than the lower threshold), 300 ms (between the lower and the upper threshold) or 600 ms (higher than the upper threshold). How the variable value is precisely provided to the RC part depends on the tool specific modeling of data flows. So in mCRL2 the driver process can perform an *s_va_int* action parameterized with one of the three values. In the UPPAAL model the global variable va_int is non-deterministically changed into one of the three values by the first process in the RC part that needs va_int (there are only two processes that use the variable and one of them is always the first one that uses it).

The driver processes can arbitrarily send one of the input events and one of the va_int values in each iteration, respectively. Therefore all combinations of input events and variable values end up in the state space in any event processing transition sequence where the combi-

nation can possibly occur. In this way the RC part is extensively tested and flaws that reside in the RC part must be found by this approach.

However note that we still do not vary the programmable pacemaker parameters that are used in the RC part, hence only flaws for the factory settings of these parameters can be found.

Although the driver processes take some interdependencies into account while providing all combinations and sequences of input events and variable values to the RC part, impossible event sequences can occur if these interdependencies are modeled incorrectly or incompletely. Note that modeling all interdependencies between the input events is hard, because the occurrence of many events depends on the behavior of processes that are not included in this approach.

This does not have to be a problem; the set of all *provided* event sequences includes the set of all *possible* event sequences and hence the RC part is at least verified for all possible behavior of its natural environment. If any provided impossible event sequence results in a violation of the requirements to the RC part, the trace that corresponds to this violation can be inspected. When this inspection makes clear that the provided input sequence is impossible with respect to the natural behavior of the RC part environment, the violation can be ignored because it will not occur during normal operation of the RC part.

If the behavior in this violation trace makes clear that a dependency must be added to the driver processes and adding this dependency is also feasible, this can be done. This is preferred because adding dependencies to the driver processes will decrease the number of input combinations which also means a reduction of the state space size. Otherwise, when adding a dependency is not possible or too complicated, this can be omitted; but then one has to be alert on the occurrence of impossible event sequences which can cause requirement violations.

Determination of the RC part requirement

The basic purpose of the RC part is classifying whether an incoming atrial sense is a normal atrial sense (output AS event) or a retrogradely conducted atrial sense (output RAS event). Besides these two events the only output of the RC part is diagnostic information that is not responsible for the behavior of the firmware (not in figure 6.1) and therapy data of the RC_Test. This latter therapy data is only used if the RC_Test therapy is active and even then it is just an adaptation of the therapy data that was used before RC_Test was made active. So model checking of the isolated RC part must be focused on verifying the requirement that **each incoming atrial sense is classified by the processes in the RC part**.

At the start of this approach we were not aware of the need of focusing on this requirement (which was mainly due to a lack of understanding of the purpose of the RC processes) and we modeled a separate stub process for receiving the output events AS and RAS. Because Vitatron indicated that the present flaw in the RC part could result in a deadlock, we only verified the 'absence of deadlock'-requirement (note that the other two requirements from section 6.1 also make no sense here, because the formal models of the heart and PG are not included in this approach). From this verification, the model appeared to be deadlock free which normally would be great, but not in this case because we wanted to find the known deadlock by means of model checking.

After this unsatisfactory result, some consultations with Vitatron gave awareness of the mentioned requirement to the RC part. Although it may seem like we were aiming at changing

the model in order to find the specific flaw by means of model checking, these consultations were more meant to invent the requirement(s) that must be verified on the RC part.

Getting aware of the 'AS/RAS-classification requirement' to the RC part led to changing the driver processes and stub processes. Instead of receiving the AS/RAS-classification by a separate stub process, we changed the event sending driver process such that it expects the classification after sending an atrial sense to the RC part. So when this driver process has sent an atrial sense event to the RC_Monitor, it waits on the AS/RAS-classification (coming from some subprocess of the RC_Monitor) before it accepts an acknowledgment message of the RC_Monitor (note that the AS/RAS-classification is normally sent to another firmware process that is not present in this approach, so the driver process acts as this process to receive the classification). If the model does not satisfy the requirement, because on a certain moment the atrial sense event is not classified in the RC part, then the driver process will keep waiting on the classification while the RC_Monitor waits until it can send an acknowledgment message to the driver process. This is a deadlock situation which can be detected by the model checker; the model satisfies the formulated requirement if and only if the model is deadlock free.

6.4.1 mCRL2 results

We have verified the constructed mCRL2 model[7] by instructing the tool `lps2lts` to detect deadlocks in the model during state space generation. Furthermore we have requested traces to any found deadlocks, which enabled us to inspect them in order to check whether a found deadlock is caused by a problem in the firmware design or by an error in the formal model.

This resulted in finding the deadlock that is known at Vitatron. The deadlock state can be reached by various traces[8], but is always caused by the fact that the AS/RAS-classification is not delivered by the RC part under certain circumstances. The verification has been executed by the mCRL2 tools in approximately four minutes on a state space of less than a million states. Compare this to the state space sizes in the heart model approach in section 6.2.1; the big difference can be explained by the absence of many processes that store information about the rate of recent events in their variables (e.g. the physiological band) and by the absence of time representing tick-actions in the RC part.

An essential element in all found deadlock traces was the occurrence of a PFAS after a ventricular pace. PFAS stands for Potential Farfield Atrial Sense and depicts that the received atrial sense event has not been caused by a real atrial contraction, but by ventricular activity that is sensed on the atrial lead of the pacemaker. In the RC part an atrial sense is classified as PFAS if it is received at most 150 ms after the preceding ventricular event (this 150 ms is the lower threshold that is mentioned on page 69). Figure 6.2(a) shows when an atrial sense is classified as PFAS.

But in the natural environment of the pacemaker firmware, the PG hardware contains a mechanism to prevent that farfield atrial senses are sent to the firmware, which is called farfield blanking (FF-blanking). By default this FF-blanking mechanism ignores a sense on the atrial lead if it is received within 150 ms after the last ventricular pace or within 50 ms after the

[7] This model can be found in the electronic appendix: `mCRL2/Isolated_RC_approach_section_6.4.mcrl2`

[8] Five of these witnessing traces can be found in the electronic appendix: `mCRL2/RC_dlk_0.trc` ... `mCRL2/RC_dlk_4.trc`

last ventricular sense. This is shown in figure 6.2(b). So in the standard configuration of the FF-blanking mechanism, no PFAS can ever occur after a ventricular pace in the pacemaker, because the blanking time interval hides all atrial senses in the time interval below the lower threshold. However, the blanking time interval after a ventricular pace (by default 150 ms) is programmable by the cardiologist in a range from 50 to 300 ms. Clearly if this time interval would be programmed smaller than 150 ms, PFASs are possible after a ventricular pace, thus making it possible that the found deadlock traces occur. This is shown in figure 6.2(c) wherein the blanking time interval after a ventricular pace has been set to 50 ms.

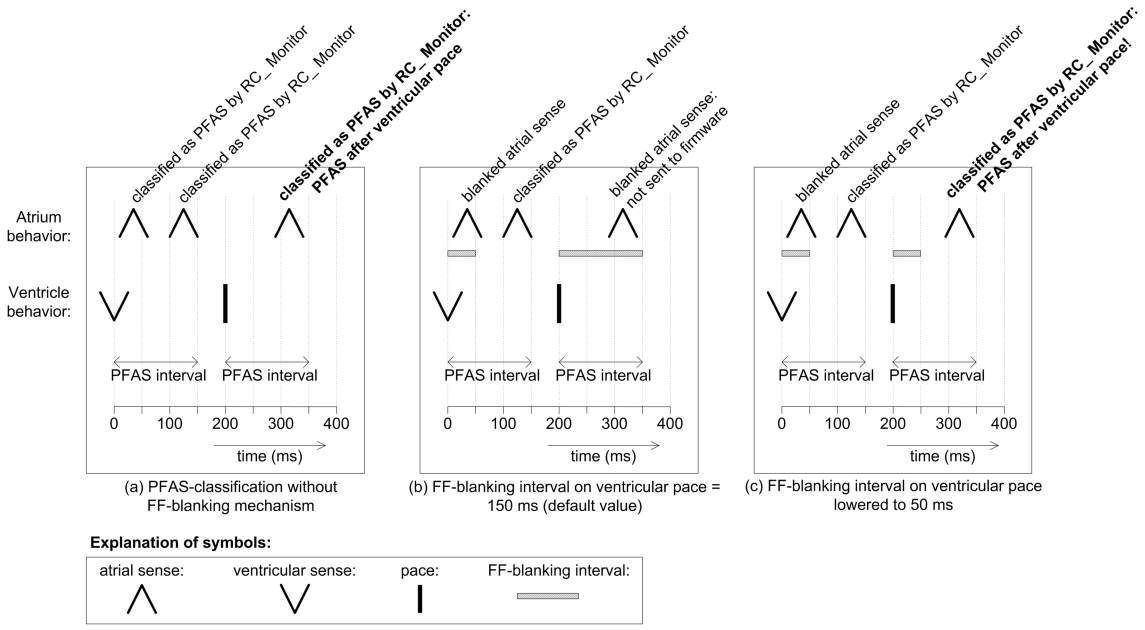


Figure 6.2: Relationship between PFASs and FF-blanking mechanism

In our driver process that is responsible for sending the variable va_int to the RC part we had not taken the FF-blanking mechanism into account. Because of that we got the mentioned deadlock traces in which a PFAS does occur after a ventricular pace. When we realized that these traces could actually not occur in the standard configuration of the pacemaker firmware, we modeled the FF-blanking mechanism into the driver process by disallowing sending the value 100 for the variable va_int if the last ventricular event was a ventricular pace. Recall that the driver process could send three different values for va_int, namely 100, 300 or 600 ms; from these three values 100 is not possible if the last ventricular event has been a ventricular pace and the FF-blanking time interval has been programmed to its default value of 150 ms. After this change to the driver process, we verified the model once again.

This resulted in finding other deadlock states in the model, whose witnessing traces[9] all contained the processing of at least three successive ventricular paces (without any intermediate atrial event) that have been sent by the driver process. This is highly unlikely behavior in the natural RC part environment, because the process Escape_Timing always plans an atrial pace after a ventricular pace; there is one exception in which a new ventricular pace is planned after a ventricular pace, but this planned pace itself will certainly cause Escape_Timing to

[9]All these witnessing traces can be found in the electronic appendix: `mCRL2/RC+FF_dlk_0.trc` ... `mCRL2/RC+FF_dlk_3.trc`

plan an atrial pace. So the maximum number of successive ventricular paces without any intermediate atrial event is two. Therefore the deadlocks that are found in this approach, wherein we added the FF-blanking mechanism to the driver process, can be safely ignored.

While reflecting on these results, we discovered a potential risk in model checking by this approach. If one would not be familiar with the known deadlock and would only have verified a model containing the modeling of FF-blanking (in its default configuration where the blanking time interval after a ventricular pace is 150 ms), only the highly unlikely deadlock states, resulting from action sequences containing at least three successive ventricular paces, would have been found. Because PFASs can not occur in this model with default FF-blanking, the more likely deadlocks are not present in this model. Furthermore, the length of the FF-blanking time interval after a ventricular pace is just one of the numerous programmable parameters, so it is plausible that this parameter would not have been lowered in this verification which would also have been an option to find the likely deadlock.

Finally by ignoring the found deadlock states (resulting from the traces that contain at least three ventricular paces) that have been classified as highly unlikely, no deadlock states are left in the state space of the model that includes the FF-blanking mechanism in its default configuration (blanking time interval after a ventricular pace of 150 ms). Therefore, one would wrongly draw the conclusion that the RC part behavior satisfies the 'AS/RAS-classification requirement'.

Although this observation depends on the specific system that is verified, we can conclude that it is recommendable to start a model checking approach like this with as much abstraction (i.e. allowing the driver processes to send all combinations of input events and input variables, without taking interdependencies into account) as possible (with respect to modeling techniques and feasibility of the state space size) to avoid potential missing of flaws in the behavior. Note that adding interdependencies to the model decreases the abstraction level and vice versa; before adding the FF-blanking mechanism to the driver process in the isolated RC approach the abstraction level was higher than after this addition. However, the addition of the FF-blanking mechanism hid crucial aspects of the behavior, which would not be known if the approach was started on this level of abstraction.

Initially we also applied much abstraction to the driver process that sends the possible events to the RC part; this was done by allowing this driver process to send all events without taking dependencies on preceding sent events into account. This high abstraction level resulted in a rather unnecessary state space explosion, because the state space contained impossible traces from pacemaker perspective (for instance traces occurred in which the driver process sends a PVC immediately after an atrial sense, while a PVC can only occur after another ventricular event according to its definition). Through getting aware of the presence of such impossible traces and the increasing effect on the model state space size, we encountered the need for modeling dependencies between events in the driver process.

This means that we have constructed the isolated RC model by using a kind of top-down approach, with respect to the abstraction level. Decreasing the abstraction level means decreasing the state space size but also a potential disguise of crucial flaws in the system behavior, so any decrease of abstraction must be considered carefully.

6.4.2 UPPAAL results

After we have found the known deadlock in the RC part with mCRL2, we tried to reach the same result with UPPAAL. Because the state space of the mCRL2 model was relatively small and the verification was done in just several minutes, we were good-tempered in achieving this goal.

So we verified the 'no deadlock'-property `A[] not deadlock` on the constructed model[10]; FF-blanking has not been included in this model in order to find the likely deadlock. After approximately 4,5 days of verification time, UPPAAL reported that the model does not satisfy the 'no deadlock'-property and delivered a witnessing trace[11]. This trace conforms to one of the (likely) traces that we found with mCRL2.

Note that the UPPAAL model of this approach does not contain any clock variables, because there is no time involved in the RC part. Because UPPAAL actually is designed to verify timed transition systems, this fact might have influenced the verification time negatively in this case.

6.4.3 mCRL2 verification of the deadlock consequences

In this section some small changes are made for confidentiality reasons.

In the original text, too many details of the 'second requirement' are mentioned.

At Vitatron it is known that the found deadlock in the RC part can result in a situation in which the 'second requirement' is violated, which is undesirable. This known consequence of the RC deadlock has even been the inspiration to formulate the 'second requirement'.

Therefore after we found the RC deadlock we were curious whether this undesirable consequence could also be found in the formal model that includes the heart model and the RC functionality. In the approach that is described in section 6.3 we clearly have not found this consequence because there can not occur a PFAS after a ventricular pace in this default configuration of the FF-blanking mechanism, which is included in the formal PG process. Therefore we adapted the model such that PFASs could occur after a ventricular pace[12] and verified the 'second requirement' on the resulting model.

We found several traces to a violation of this requirement in the model state space. All these traces correspond to one of the traces that have been found in the isolated RC model. After the atrial sense event that is not classified by the RC part (causing the deadlock in the isolated RC model) the next ventricular sense event causes a violation of the 'second requirement'. Explaining the exact reason why this violation occurs after the absence of the AS/RAS-classification is beyond the scope of this report, but it is important to notice that the reason resides in the cooperating core firmware processes AV_Monitor and Escape_Timing.

[10] This model can be found in the electronic appendix: `Uppaal/Isolated_RC_approach_section_6.4.xml`

[11] This witnessing trace can be found in the electronic appendix: `Uppaal/RC_dlk-1.xtr`

[12] This has been done by decreasing the FF-blanking time interval to 50 ms, fixing the sensed AV delay to 200 ms, and allowing the atrium to show a high rate of 180 bpm (the latter is necessary to make PFAS occurrences possible).

Because we have found this violation of the 'second requirement' through adapting the formal model by using knowledge of the deadlock, this result is not that important from model checking perspective. But on the other hand it is a nice example of investigating the consequences of a found flaw (in a part of the model) on the external behavior of the complete model. Finding a flaw in an isolated part of a model is generally much easier, because often a model part does have fewer dependencies on external variables and parameters which makes it possible to vary these variables and parameters in order to verify the model part more extensively. After finding a flaw in a model part, the precise external variables and parameters for which this flaw occurs can be put in the complete model in order to discover the consequences of the found flaw on the external behavior of the complete model.

Chapter 7

Recommendations to Vitatron

From the found results, we can state that applying model checking to firmware designs is certainly not straightforward. Based on our experiences with formally modeling the pacemaker firmware, the actual model checking and its results, we give some recommendations for the future application of model checking at Vitatron. These recommendations also include some open questions that could be a topic of future research in the academic world.

In section 7.1 we give recommendations about the 'model checking aimed'-design of future firmware systems and the formal modeling of such designs. Recommendations about how and whereupon model checking should be applied are sketched in section 7.2. Section 7.3 gives some recommendations about resources and skills that are needed in order to apply model checking and finally section 7.4 suggests some topics for future research.

7.1 'Model checking aimed'-design

During formally modeling the pacemaker firmware we met some constructs in the H&P model which caused problems in our formal models. This was the case either because a modeling construct had dramatic effects on the state space size, or because it was hard to invent a correct formal translation of the modeling construct. In this section we will address these problems together with recommendations to prevent such problems in future firmware designs in order to make model checking of such systems more feasible.

Deterministic design

Because some process activation tables (PATs) in the H&P model do not specify the ordering in which child DFDs and PSPECs are triggered and output events are sent, we have modeled the ordering that is chosen in the implementation code of the firmware (see section 5.5). We described that this is undesirable, because we want to verify the design specification itself instead of its implementation. Therefore we recommend specifying an explicit order of such PAT-consequences in the specification of future firmware designs, if such a design would be the basis for a formal model on which model checking is applied. This is recommended for two reasons: in section 5.5 we already discussed its state space reducing effect and besides that it eases the creation of formal models with a better conformance to the design specification.

We realize that leaving the ordering of triggers and output events unspecified has advan-

tages from implementation perspective. If the ordering in which certain actions must be performed has not been specified in the system design, because their ordering really does not matter for the behavior of the system, the implementor can choose the most efficient ordering from the possible options.

But from model checking perspective, this is undesirable. To obtain the best conformance between the system design and the formal model of this design, one would generally also leave the ordering of such actions undetermined in the formal model. In fact, all possible orderings then can occur in the model state space, which causes a dramatic explosion of this state space as we have experienced. To reduce the state space, one could choose a specific ordering during the translation of the design to a formal model; but this will harm the conformance between the design and the formal model and it could be the case that certain design flaws are disguised by this specific choice. If during the implementation stage later on, another choice for this ordering would be made for which these flaws do occur, the implementation will contain errors while model checking resulted in a validation of the design. Of course this is highly undesirable and therefore it is clear that determinism in the design is always preferred above non-determinism if this design will be verified by model checking.

Related to this point we have another concern about the style in which the pacemaker firmware is specified. In some large DFDs it is hard to find out which behavior is exactly specified for each incoming event. Sometimes this is caused by a lack of order specification as mentioned above, but also the way certain H&P constructs are used sometimes makes the meant behavior unclear. For instance, triggering of a child DFD is often used where sending an event to the child DFD would provide much more clarity about the meant behavior of this child DFD (process the specific event instead of react to a general trigger).

These arguments have to be considered carefully while designing future firmware systems, if model checking will be used as a verification method of these designs. For future firmware designs that are verified by model checking, we therefore recommend the following:

- Apply as much determinism as possible while specifying firmware designs.
- Specify future firmware designs in a more formal language or in a less ambiguous way to create more clarity about the meant behavior. The used language does not have to be a purely formal language that is suitable for model checking, but using a more formal design language or specifying the design in a less ambiguous way will certainly ease the translation of the design into a purely formal language.

  Note that specifying a system design directly in a formal language that is suitable for model checking, will generally not lead to a formal model on which model checking can be applied directly. While specifying a system design, you do not want to worry about the consequences on the state space size of the used design constructs. However, if the system design is specified in a formal language, the translation to a formal model on which model checking can be applied, becomes just a matter of applying simplifications and reductions to the system design.

Obeying these two recommendations will ease the translation of future firmware designs to formal models, but may have disadvantages from implementation perspective: implementing may become more difficult and the resulting implementations may become more inefficient.

State space reducing modeling

In section 5.5 we described two relatively simple state space reductions that are applied in the formal translation of the firmware design. From these reductions, we draw some recommendations.

The resets of the counter variables in the processes Multiple_Successive_VS and NOA_Detection that we added, avoid that any value 'stays behind' in such a counter variable while this value is no longer needed. Although this does not form a problem during runtime of the firmware, it does increase the state space size of a formal model that represents the firmware. Therefore a recommendation to force resetting of variables whose variable is no longer needed in future firmware designs would be too outrageous, but at least one has to be alert on this issue in the design specification during formally modeling of such a design.

The same arguments apply to the bounding of the counter variable Successive_VP_ cntr in the process VP_AS_Pattern that we added; during normal runtime of the firmware (and generally also in the model state space if the firmware model is placed in the context of the formal heart model) the infinite increase of the counter variable does not occur. But we met the problem of the possibly infinite increasing of the variable when we exploited the isolated RC approach (section 6.4), because the nature of this approach made the specific input event sequence, which causes the infinite increase, possible. Once again we recommend being alert on such issues during formally modeling future firmware designs, because they can have dramatic effects on the state space size.

Explicit purpose and interface specification of design components

In section 6.4 we described that during the exploitation of the isolated RC approach, we initially had some trouble in finding the right requirements to the RC part. At first we were not aware of the main purpose of the RC part, i.e. classifying each incoming atrial sense as normal atrial sense or retrogradely conducted atrial sense. Because of this, we only focused on the 'absence of deadlocks'-requirement, which did not result in finding the known deadlock.

We attribute the lack of awareness of the mentioned purpose of the RC part to the somewhat unclear modeling style in the H&P model, which is mainly inherent to the H&P method itself. Looking at the sphere that represents the process RC_Monitor (figure 6.1 shows a simplified version of this), we see several incoming and outgoing control flows and data flows. Among several control flows for communication to the RC_Test and data flows that provide diagnostic information to other processes and the 'outside world', there is an outgoing control flow on which the AS/RAS-classification is sent to other processes. But unfortunately this latter control flow does not attract attention as being the most crucial outgoing control flow of the process or as being linked to the incoming atrial sense control flow. Also the documentation and the model itself do not explicitly mention the requirement that the RC_Monitor must always deliver the AS/RAS-classification on receipt of an atrial sense event.

The described problem clearly is caused by a lack of an explicit definition of the purpose and the interface of the RC_Monitor. From these considerations about the design specification of the RC_Monitor, we think that it is justified to draw the following recommendations for the design of the complete firmware.

The H&P model of the firmware design describes *how* the behavior of each individual de-

sign component is accomplished (i.e. all actions that are performed upon receipt of a specific input event can be tracked), but does not define *what* exact purpose is intended by designing this behavior. The purpose of a design component is always (mainly textual) described in the accompanying documentation, but reading this purpose description mostly requires much background knowledge and linking this description to the design in the H&P model is sometimes quite difficult. Next to this, the H&P model contains small textual descriptions of the purpose of most components, but such textual descriptions are generally rather ambiguous. Hence extracting the intended purpose of a design component is quite hard and requires much detailed knowledge of the design and the domain in which the designed firmware operates.

To be able to verify whether the behavior of a design component is correct, its intended purpose must be absolutely clear. Therefore it is recommended to state an explicit and formal specification of the purpose and the interface of each component in future firmware design specifications. This specification should define the relationships between the input interface and the output interface (containing events and variables) of the component and must abstract away from the precise internal behavior that accomplishes these relationships. Because all interaction of the component with its environment takes place via its interfaces, it is always possible to define its purpose explicitly in such relationships. Preferably the specification is made purely formal, e.g. by using logical propositions for all interface relationships, so that the ambiguity of the design component purpose is minimized.

Preferably this purpose and interface specification is included in the design specification itself in order to make the design as complete as possible, but it is also an option to put it in the accompanying documentation. In the latter case, it must be guaranteed that the purpose and interface specification can be linked easily to the design specification.

Obeying these recommendations will probably ease the invention and the formulation of requirements to the models that can be verified by model checking. If each design component has a clearly specified purpose, it can be verified whether this purpose is really accomplished by the design component by means of model checking.

Low coupling/high cohesion design pattern

From the isolated RC approach we also have experienced that model checking parts of a system by such an approach can be feasible and can result in important knowledge about the behavior of the complete system. Because the behavior of the RC part is determined by only 7 incoming control flows and 1 incoming data flow (besides the programmable parameters that are constant during runtime) we were able to create a formal model of this behavior with a relatively small state space. The RC part itself consists of many processes that collaborate with each other. This is a clear example of how applying the well-known 'low coupling/high cohesion' design pattern to a firmware design can have positive effects on the feasibility of model checking. Of course the specific design of the RC part contained a flaw, so with respect to that flaw the design must not be copied onto other designs. But mainly because the RC part satisfies the 'low coupling/high cohesion' design pattern, we were able to discover the flaw by means of model checking.

Therefore we recommend designing future systems according to this design pattern as much as possible to make a kind of unit testing by model checking possible. We come back to the point of unit testing by model checking in the next section.

7.2 Application of model checking

During the project, we experienced that formalizing system behavior can greatly improve the comprehension of the modeled system. At the start of the project, many manual explorations of the firmware H&P model were required to get some first insight in the model. But this was totally insufficient to get a really precise understanding of the system details. Nevertheless we started modeling some parts of the system based on our limited knowledge.

Formalizing a system means that responsibilities of processes and communication between the processes are made explicit. Because of that, we were forced to invent formal modeling techniques for formally representing the firmware behavior. By exploiting these invented modeling techniques we translated all details of the firmware design, although we were not totally aware of the purpose of all design details that we were modeling.

After the first parts had been formally modeled, we used the formal analysis techniques to verify whether the formal model represented the system correctly: Exploring the formal model by simulating actions, trying to verify the 'absence of deadlocks'-requirement, and raising questions to Vitatron based on our observations from these simulations and verifications, led to correction of modeling errors but also gave much understanding of how events flow through the various firmware processes. This better understanding of the firmware functionality often resulted in ideas for improving the model and the used modeling techniques.

In this way we used an iterative procedure consisting of alternating stages of modeling the system details and analyzing the resulting formal model. This modeling approach gradually made clear the best way of modeling the firmware design and furthermore improved our understanding of the firmware behavior itself.

What do these experiences mean for the application of model checking in the development of future firmware systems? If formal modeling and model checking would be exploited during early design of future systems, a better understanding of the consequences of made design decisions can possibly be obtained before the complete design has been finished. We recommend a parallel development of the formal model that is suitable for model checking besides the development of the firmware design.

Although our heart model approach from section 6.2 and 6.3 has shown that verifying a firmware design extensively (by using a heart model showing many possible rates, including very high rates) is yet a rather infeasible task, it is possible to analyze the firmware behavior for some basic heart behavior (for example with a heart model that shows a constant spontaneous atrial rate). By providing such basic heart behavior to a formal model of the firmware design, at least big problems in the design can be detected.

Next to this, a formal model of the design in which the heart model can show an extensive set of behavior, is analyzable by using a simulator tool to explore the model by manually performing actions. This is because for such simulations of a formal model, it is not necessary to be able to generate the complete state space of the model. Sometimes such 'manual' simulations can be useful, for instance when one wants to analyze the firmware behavior for a specific sequence of heart behavior. However, the results of such simulations are also limited, because by simulation only a small part of the complete state space can be analyzed.

Furthermore, design parts of limited size can be verified extensively by using an approach similar to the one we have used to verify the RC part in isolation (section 6.4). We have already mentioned that model checking can act as a kind of unit testing technique in this way.

By these different model checking approaches, unwanted behavior in the firmware design can possibly be detected (and hopefully corrected) in an early stage of development. When such unwanted consequences should be discovered by unit tests at the end of the design stage, such corrections might be much more expensive.

Model checking separate design parts

Coming back to the point of unit testing by model checking, we have seen one successful example of such an approach in the isolated RC approach (section 6.4). In this approach the RC part was taken out of its normal context and placed in a formal model wherein its input events and input variables were provided by driver processes. Because the RC part only depends on one external variable, we were able to verify this part extensively by letting the driver processes provide all possible variable values and all combinations of input event sequences. Of course it highly depends on the specific design that is verified, whether some parts of that design can be verified by such an approach.

Although not described in chapter 6 because the results were very disappointing, we have also investigated applying similar approaches to the Escape_Timing part of the design and to the complete firmware design. In case of the Escape_Timing part the approach was not successful because this part depends on too much external variables (2 time intervals from the PG and several data variables from the Atrial_Monitor and The_Umpire); varying all these variables resulted in a huge state space explosion, even varying only the time intervals and keeping the other variables constant was certainly not feasible. It may be clear that the same problem appeared when applying the approach to the complete firmware design.

What can we conclude from these investigations? At least the results from the isolated RC approach show that model checking can be applied to design parts whose behavior does not depend too much on the values of external variables. It seems that the number of possible input events to a design part has less influence on the feasibility of model checking on this part; in the firmware H&P model the number of input events to a process is limited anyhow and usually there are lot of dependencies between them through which the number of possible event sequences is also limited. On the other hand, if the number of external variables or the size of their value ranges increases, the number of possible value combinations of these external variables increases exponentially, which soon causes the state space to explode and the model checking approach to become infeasible.

Therefore it is recommended to be careful in selecting on which parts of a design, model checking is going to be applied. A kind of bottom-up approach is recommendable, i.e. start with verification of the smallest parts of the design, followed by combining several parts in the verification, and so on until the number of external variable dependencies becomes too large and model checking starts being infeasible. Of course the function of the verified design parts and the requirements to them must be clear before model checking can actually verify these requirements (obeying the recommendations from section 7.1 should imply this). If the complete design would have only few dependencies on external variables, it could appear that model checking of the complete design is feasible by this approach, although the nature of the firmware for implantable heart devices does not make this plausible.

Another approach that can be applied in order to verify larger design parts can be the following. Validated design parts can be replaced by driver/stub processes that can show all possible external behavior of this design part, but whose internal behavior is omitted.

In this way these design parts do not contribute to the state space size too much, through which larger design parts that include these design parts, can be verified. We have not investigated such an approach in this project, but it is likely that such an approach would be feasible.

Recall from section 6.4.1 that it is recommended to start such approaches, in which design parts of limited size are verified, with as much abstraction as possible to avoid possible missing of flaws in the system. This high abstraction level can mainly be reached by initially modeling the driver processes such that they provide all combinations of input events and input variables, without taking interdependencies into account. However, if the resulting state space would become too large, the abstraction level can be lowered to make the approach feasible. The abstraction can also be lowered if it appears that the driver processes provide impossible input event sequences. So with respect to the abstraction level, a top-down approach is recommended.

Decreasing the abstraction level can be done by modeling extra dependencies between the several input events and input variables into the driver processes that provide these inputs. Decreasing the abstraction level however must be considered carefully, because it can also mean a disguise of crucial flaws in the system behavior.

Model checking tools

Besides these general recommendations on the application of model checking to firmware designs, we also try to give some recommendations on the choice of a tool for model checking firmware systems. Although this is rather difficult because we have only investigated the application of two specific model checkers from the several ones that are available.

mCRL2 has appeared to be a tool in which firmware designs can be very well modeled and that can deliver very useful results in analyses and verifications of the resulting models. Disadvantages of mCRL2 could be that using the language and the toolset requires rather much tool-specific knowledge and that the toolset and its documentation are still under development. The latter can of course also be an advantage, because the toolset is continuously improved. Furthermore the mCRL2 development team is always accessible to receive questions and comments on the mCRL2 language and its toolset.

In this project Uppaal has appeared as a tool in which formal designs can be constructed very easily because of its graphical interface. Through this interface the conformance between the firmware design and its formal representation is a bit clearer than when mCRL2 is used. We were able to analyze and verify some small models with Uppaal, but when the models grew larger, the runtimes for analyses and verifications on these models exploded a lot sooner than was the case with mCRL2. Because of this, the model checking results that we obtained by using Uppaal are unfortunately very limited.

Based on these experiences we would recommend using mCRL2 as a model checker at Vitatron, but it is certainly also recommended to investigate the usability of other available tools.

7.3 Model checking resources and skills

In this project we have used the TU/e compute server *elephant* to perform our analyses and verifications of the models. This machine has 128 GB of memory through which we could handle larger state spaces and perform quicker verifications than on a personal computer or

a 'normal' server. Being able to generate these huge state spaces was very useful in order to investigate to which extent the parameters of the model were responsible for the state space explosions. But as we have concluded earlier, generating and verifying these huge state spaces have not led to important results because we were only able to vary a limited set of parameters in the models. Even though, our isolated RC approach has a relatively small state space and has led to useful results.

Regarding the necessary computation power to apply model checking, we can therefore conclude that these do not have to be gigantic, although more resources are always useful to be able to cope larger states spaces. Because state spaces grow exponentially with the size of the verified models, more resources will only cause a limited shift to the model size boundary above which model checking starts being infeasible.

Engineers that are going to apply model checking need to have certain skills. General knowledge of model checking is required; [6] is a good source for this. An important skill that must be developed in order to apply model checking is the ability to use functional programming as contrasted with imperative programming which would be very familiar to Vitatron engineers. With functional programming it is defined *what* purpose a certain component must accomplish, instead of *how* this purpose is accomplished. Formal models on which model checking can be applied are constructed using a functional programming like language in most model checkers.

Constructing a formal model of a system and formulating requirements in logic formulas of course also requires knowledge of the specific model checker language and the available logics; for mCRL2 and Uppaal introductions are included in this report. For mCRL2 it is furthermore very useful to obtain general knowledge about process algebras; this is for instance provided in [11].

But probably the best way to learn using a model checker tool is practicing by trying to model and verify some simple example systems. Experience is also required for inventing and formulating the right model requirements. Sometimes a model requirement can be verified more easily by using an auxiliary action that is performed if the requirement is violated. To determine which approach must be used for verification of a certain requirement, experience is essential.

7.4 Future research

This section describes some points that can be a topic of future research, either in the academic world (perhaps as a master project) or in collaboration with Vitatron or other companies.

Because the formal modeling of firmware designs appeared to be a rather time intensive job (as probably is the case with most other system designs), it would certainly be useful to look for better methods for obtaining formal models from these designs. Perhaps tools can be developed for automatic translation of design specifications to formal models (note that such tools are already available for the conversion of UML diagrams to Uppaal models: [1] and for the conversion of Petri nets to mCRL2 models).

Note that we have already mentioned the suggestion to specify future firmware designs directly into a formal model checker language. Although model checking most often can not be applied directly on such a 'formal design', the translation of such a design to a feasible

(with respect to its state space size) formal model becomes much easier. In fact this translation then becomes only a matter of applying simplifications and reductions to the 'formal design'.

Another point of future research could be the investigation of more efficient techniques for applying model checking on system designs that depend on many programmable parameters. This is a point that we do not have addressed in this project, because it would certainly make model checking of the pacemaker firmware design infeasible. Perhaps techniques can be developed that reuse parts of the already generated state space when some parameters are changed and the state space is generated again. Changing some parameters does not necessarily mean that the complete state space is changed, so our intuition is that the parts which remain the same could be reused rather easy.

In section 6.1 we described that formulating a requirement in the modal μ-calculus for verification on the mCRL2 models was rather hard. After having obtained a logic formula of a model requirement in the modal μ-calculus, it is especially hard to verify whether this formula forms a valid representation of the model requirement. Perhaps improvements can be made on this point by TU/e researchers, for instance by inventing a higher-order language in which requirements can be easier formulated. The μ-calculus then might act as the underlying basis for this language.

Chapter 8

Conclusions

The feasibility of the application of model checking on Vitatron's firmware designs like the DA+ pacemaker firmware design as an independent verification technique is certainly not confirmed. Nevertheless the investigations from this project have shown that it might be valuable to use model checking as a supplementary test technique besides the current test techniques in the future.

The pacemaker firmware is highly time-dependent; besides the ordering in which external events are received, their timing has a great influence on the behavior of the firmware. Of course, this influence on the behavior also has an enlarging effect on the state space of any formal model that represents (a part of) the firmware. Furthermore, besides this direct influence on the state space, much timing information is stored in process variables whose values are used in determining the subsequent behavior. The model state space size thus is highly proportional to the timing diversity of the provided external inputs to the model.

In our heart model approaches the external inputs were provided to the firmware by the formal heart model. We regulated the timing diversity of these external inputs by allowing different sets of optional atrial rates. In this way the time dependency of the firmware model became very clear; increasing the number of possible atrial rates had an exponentially increasing effect on the state space size. Because verification of a formal model becomes infeasible when the model state space becomes too large, we were forced to limit the diversity of the provided heart model behavior in order to make verification possible. Although the verifications that have been carried out in the presence of the heart model all were successful (all defined system requirements appeared to be true in the verified models), the conclusions for the firmware design that can be drawn from these successful verifications are very limited. This not only results from the limitations to the heart behavior, but also from the state space reductions (described in section 5.5) that we applied. Furthermore the firmware models are only analyzed and verified for one particular setting of the programmable parameters. These three types of reductions in the heart model approach make that we have only reached a minimal test coverage of the firmware system.

Based on these considerations we can state that this model checking approach does not yet form a feasible and reliable verification technique for complete firmware designs. However, analysis and simulation of future firmware systems would be possible by using this approach, because these do not require the generation of the complete state space and may provide insight about the relations between the system components and about the external behavior.

For verification of future firmware designs this approach can probably only be used if the modeled input behavior is limited; this might be useful when the design must be verified for some specific heart behavior (e.g. verifying the firmware behavior for some particular cardiac arrhythmia). However, the reliability of verification results from such approaches always must be carefully judged.

Our investigation of the isolated RC approach (see section 6.4) has shown that the application of model checking on a part of a firmware design is probably always more feasible than any approach that verifies the complete design, either by means of a formal heart model or another environmental model. Models of design parts automatically have a smaller state space and less time dependencies which makes it possible to verify the parts more extensively. This approach appeared to be feasible on the RC part; we were able to place the RC part in the context of a complete environmental model that provides all possible timings of input event sequences and all possible values of input variables.

Our verification results show that this model checking approach can lead to the discovery of a requirement violation in a model of a design part and that the consequences of such a violation can be investigated in a model of the complete design afterwards. Furthermore, these verification results are very reliable, because of the complete environmental model that is used in such an approach. However note that we have only verified the RC part for one particular setting of the programmable parameters. To verify the part even more completely, these parameters should be varied, but whether this would be feasible is left as a topic of future research.

After this overview of our experiences from investigating the several model checking approaches, we give a summary of our recommendations to Vitatron and the academic world from chapter 7.

- 'Model checking aimed'-specification of firmware designs.
  - Apply as much determinism as possible (from implementation perspective) while specifying firmware designs. For instance, specify an explicit order of triggering processes and sending output events.
  - Specify future firmware designs in a more formal language or in a less ambiguous way in order to increase the clarity of the firmware design specifications. Make sure that the specified behavior for each possible incoming event to the system can be easily tracked in the design.
  - State an explicit and formal definition of the purpose and the interface of each component in the design specification. This will probably ease the invention and the formulation of requirements that can be verified by model checking.
  - Obey the 'low coupling/high cohesion' design pattern as much as possible in the specification of firmware designs. This will reduce the dependencies between design components which makes extensive verification of these components by model checking more feasible.
- Translation of firmware designs to formal models.
  - Be alert on variable values that potentially 'stay behind' in the model state space while they are no longer influencing the behavior.

  - Be alert on potentially infinitely increasing variable values. Such variables can cause the state space to become infinite as well.

- Model checking approach.
  - Apply model checking already during early design of firmware systems; this can either be done by formally modeling the firmware in the context of a simple heart model or by exploring the formal model of the firmware manually by using some simulator tool. This will provide early insight and understanding of the consequences of made design decisions and might prevent expensive corrections at the end of the design stage.
  - Apply model checking as a kind of unit testing technique. Selecting the analyzed and verified system parts is preferably done by a bottom-up approach; next it is recommendable that the applied abstraction level is initially chosen as high as possible. Later on the abstraction level possibly can be lowered if the resulting state space appears to be too large.
  - The comparison between mCRL2 and UPPAAL in this project has shown that mCRL2 is preferably chosen as a model checking tool for firmware designs. However, it is also recommended to investigate the feasibility of other available model checking tools in the future.

- Resources and skills.
  - Perform model checking on a machine with many resources. Although the availability of many resources is important, no super computer is required in order to apply model checking. The requirement violation in the RC part even had been found when the verification would have been carried out on a personal computer, because the state space of the corresponding model was relatively small.
  - Develop the skills of engineers that are going to apply model checking. General model checking knowledge is important, but knowledge about the specific model checker that is used is also very important.

- Future research topics for the academic world.
  - Investigate the development of more efficient methods for obtaining formal models from system designs, e.g. tools which translate a system design to a formal model automatically.
  - Investigate how model checking can be performed more efficiently on systems with many programmable parameters.
  - Investigate the development of a higher-order language in which model requirements can be easier formulated than currently is the case with the modal μ-calculus.

For the nearby future, it is recommended that Vitatron starts developing some experience with model checking. Furthermore it is recommended to investigate whether applying model checking during early design is indeed valuable and feasible for Vitatron.

Initially these two goals can for instance be combined in some coming firmware design project by trying to apply model checking to design components of limited size. Especially design components that contain many collaborating state machines are good candidates to

be formally verified. It is preferred that formal models of such design components are created by the firmware designers and during the design specification stage, because through formal modeling much insight can be obtained about the consequences of made design decisions. Furthermore it is recommended to state the requirements to the modeled design components also already during the design specification stage, so that these requirements can be verified 'immediately' on the concurrently developed formal models. Note that obeying our recommendation of explicitly specifying the purpose and interface of each component implies that the main requirements to the design components are available.

Even if model checking is not applied immediately, it is anyhow recommended to obey our 'model checking aimed'-specification recommendations. Specifying future firmware designs according to these recommendations increases the feasibility of the application of model checking later on.

Bibliography

[1] Hugo/RT, UML model translator for model checking, theorem proving, and code generation. Webpage: `http://www.pst.informatik.uni-muenchen.de/projekte/hugo/`.

[2] mCRL2 toolset homepage. `http://www.mcrl2.org/`.

[3] UPPAAL homepage. `http://www.uppaal.com/`.

[4] G. Behrmann, A. David, and K.G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, 2004.

[5] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, number 3098 in LNCS. Springer–Verlag, 2004.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla. *Model checking*. MIT Press, 2000.

[7] F. Feith. Electrophysiology. Education Modules Vitatron: EMD 104-144, 1996.

[8] F. Feith. Pathology. Education Modules Vitatron: EMD 105-145, 1996.

[9] J.F. Groote, A.H.J. Mathijssen, S.C.W. Ploeger, M.A. Reniers, M.J. van Weerdenburg, and J. van der Wulp. *Process Algebra and mCRL2*. IPA Basic Course on Formal Methods, 2006.

[10] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. The formal specification language mcrl2. In *Methods for Modelling Software Systems*, number 06351 in Dagstuhl Seminar Proceedings, 2007.

[11] J.F. Groote and M.A. Reniers. Designing and understanding the behaviour of systems. Lecture notes for the TU/e course "Requirement Analysis, Design and Verification" (2IW25), given by J.F. Groote. These notes are available at `http://www.win.tue.nl/~jfg/educ/educ.html`, 2007.

[12] D.J. Hatley and I.A. Pirbhai. *Strategies for real-time system specification*. Dorset House Publishing, New York, 1987.

[13] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag.

[14] M.R.A. Huth and M.D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, second edition, 2004.

[15] A. Lindgren and S. Jansson. *Heart physiology and stimulation: an introduction*. Siemens-Elema, 1992.

[16] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[17] H.J.J. Wellens, K.I. Lie, and M.J. Janse. *The conduction system of the heart: structure, function and clinical implications*. Stenfert Kroese, 1976.