Assurance Cases in Model-Driven Development of the Pacemaker Software*

Eunkyoung Jee, Insup Lee, and Oleg Sokolsky

PRECISE Center
Department of Computer and Information Science
University of Pennsylvania, Philadelphia PA 19104, USA
eunkjee@seas.upenn.edu, {lee, sokolsky}@cis.upenn.edu

Abstract. We discuss the construction of an assurance case for the pacemaker software. The software is developed following a model-based technique that combined formal modeling of the system, systematic code generation from the formal model, and measurement of timing behavior of the implementation. We show how the structure of the assurance case reflects our development approach.

Keywords: assurance case, pacemaker challenge, model-driven development, real-time software

1 Introduction

We consider the problem of developing an assurance case for the real-time cardiac pacemaker software, representative of life-critical systems in which many complex timing constraints are imposed. This work was motivated by the Pacemaker Grand Challenge, the first certification challenge problem issued by the Software Certification Consortium (SCC) [1]. Boston Scientific has released into the public domain the system specification for a previous-generation pacemaker to have it serve as the basis for a challenge to the formal methods community. In [2], we proposed a safety-assured approach for the development of pacemaker software. In this paper, we consider how the features of our development process are reflected in the structure of the assurance case.

When we develop a real-time system, guaranteeing timing properties on its implementation is an important but non-trivial issue. It becomes essential if the real-time system is a safety-critical one in which violation of timing properties can result in loss of life. We focus on how to systematically implement time-guaranteed real-time software from a given model and how to convincingly demonstrate the safety of the software.

Several concepts and approaches can be effectively integrated to contribute to the development of safety-assured real-time software. The model-driven development (MDD) approach is steadily gaining popularity in the development of embedded software. According to the MDD concept, we create a formal model

* This research was supported in part by NSF CNS-0834524 and NSF CNS-0930647.

of the real-time system, verify the model, and generate an implementation code from it. In order to validate the result and check the timing constraints on the implementation, we perform measurement-based timing analysis on the implementation and revise the implementation and the model according to the timing analysis result, repeating the verification process if necessary.

Many safety critical systems, such as avionics systems and medical devices, are subject to regulatory approval. Once the system is implemented, it is necessary to present development documentation to the regulators for review. Currently, this process is lengthy and expensive. Certification costs constitute a significant fraction of the development costs for regulated systems.

Assurance cases are currently seen to be holding a promise of both reducing certification costs *and* improving the quality of certification by tying it to the evidence. An assurance case is a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a system's properties are adequately justified for a given application in a given environment [3]. Yet, there are few commonly accepted ways of constructing assurance cases. There is evidence that a poorly structured assurance case can hamper the evaluation process, rather than help it [4]. Clearly, there is no "one size fits all" structure, and software developed through different processes is likely to require different arguments about its safety. The case study put forth in this paper aims to discover appropriate structures for one development approach, namely model-driven development.

The contribution of this paper is the construction of an assurance case for real-time software developed using a model-driven safety-assured process based on formal modeling, rigorous code generation from the verified model, and subsequent validation of the timing characteristics of the developed code. We believe that other model-driven development frameworks will be amenable to similarly structured assurance cases. Our ultimate goal is to arrive at an accepted assurance case template that can be applied to a variety of safety-critical software-based systems. Having such a template will simplify regulatory approval of these systems, by making the argument easier for the evaluators to follow. While this goal still lies ahead of us, this work can be seen as the first step in the right direction.

The remainder of the paper is organized as follows: Section 2 explains the background of the case study. Section 3 presents the overview of our development process and demonstrates its application to the development of the pacemaker software. Section 4 presents the assurance case for the pacemaker software in its relation to the evidence generated during the development. We discuss related issues in Section 5 and present a review of previous work related to topics addressed in this paper in Section 6. Section 7 concludes the paper.

2 Pacemaker Operation

2.1 Heart

A human heart has four chambers: right and left atria, and right and left ventricles. De-oxygenated blood from the body is collected in the right atrium and then pumped into the lungs via the right ventricle. In the lungs, carbon dioxide in the blood is replaced with oxygen. This oxygenated blood then passes through the left atrium and enters the left ventricle, which pumps it out to the rest of the body.

From an electrical point of view, the heart is a pump made up of muscle tissue, controlled by an intrinsic electrical system. An electrical stimulus generated periodically (normally about 60-100 times per minute) by the sinus node, located in the right atrium, travels through the conduction pathways and causes the heart's chambers to contract and pump out blood. The atria are stimulated and contract shortly before the ventricles are stimulated and contract.

Under some conditions, this intrinsic cardiac system does not work properly and the heart rate becomes overly fast or slow, or irregular. In these situations, the body may not receive enough blood, which causes several symptoms such as low blood pressure, weakness, and fatigue. To avoid these symptoms, a pacemaker can be used to regulate the heartbeat [5].

2.2 Pacemaker

A cardiac pacemaker is an electronic device implanted into the body to regulate the heart beat by delivering electrical stimuli over leads with electrodes that are in contact with the heart. These stimuli are called *paces*. The pacemaker may also detect natural cardiac stimulations, called *senses*. We refer to cardiac paces and senses collectively as *events*.

A pacemaker must satisfy three fundamental medical requirements: the rate at which the cardiac chambers contract must not be too high; the rate at which the cardiac chambers contract must not be too low; the ventricles must contract at a particular interval after the atria contract. These general requirements are concretized by setting specific values or ranges to configurable parameters for the pacemaker.

The pacemaker can operate in a number of modes, distinguished by which chambers of the heart are sensed and paced, how sensed events will affect pacing, and whether the pacing rate is adapted to the patient state. In this paper, we concentrate on the VVI mode, in which the pacemaker senses only ventricular contractions and performs only ventricular pacing. In this mode, pacing is inhibited if ventricular contractions are sensed.

A pacemaker in the VVI mode operates in a timing cycle that begins with a paced or sensed ventricular event. The basis of the timing cycle is the *lower rate interval* (LRI), which is the maximum amount of time between two consecutive events in one chamber. If the LRI elapses and no sensed event occurs since the beginning of the cycle, a pace is delivered and the cycle is reset. At the beginning

of each cycle, there is a *ventricular refractory period* (VRP), usually 200-350 ms. Chaotic electrical activity in the heart immediately following a pace may lead to spurious detection of sensed events that can interfere with future pacing. For this reason, sensing is disabled during the VRP period. Once the VRP period is over, a sensed ventricular event inhibits the pacing and resets the LRI, starting the new timing cycle. Hysteresis pacing can be enabled in the VVI mode, when the pacemaker will delay pacing beyond the LRI to give the heart a chance of resuming normal operation. In that case, the timing cycle is to a larger value, namely the *hysteresis rate interval* (HRI). In our implementation, hysteresis pacing is applied after a ventricular sense is received, and disabled after sending a pacing signal.

3 Model-Driven Development of Pacemaker Software

3.1 Overall Process

We propose a safety-assured development process for real-time software. The proposed process follows a model-driven development approach with the emphasis on ensuring that the implementation satisfies timing properties that are satisfied in the model. Fig. 1 shows the overall process.

During the requirements and design phases of the software life cycle, developers first start from formal modeling with timed automata of the real-time software. Second, model checking is performed on the timed automata model with respect to desired properties using a real-time model checker such as UPPAAL. We focus on safety properties, especially timing properties which require that a certain event should happen no later than a specific delay. Given a verified formal model, an implementation code is synthesized in the third step.

In the fourth step, we check to see if the same properties checked on the model are still satisfied by the code running on a target platform. If some timing properties are not satisfied by the code, we measure how much actual time deviates from the expected. During the fourth step, we find a *timing tolerance* value, Δ, through the measurement-based timing analysis. Guards in the code are modified with this Δ to make the code satisfy timing properties. Once it is confirmed that the code satisfies the desired timing properties with the Δ, changes of the code, i.e., modified guards with the Δ, are reflected to the model in the fifth step. If the modified model still satisfies all the properties, the overall process ends. Otherwise, the process is repeated by revising the problematic model and the code. We describe each step with the pacemaker example in the following subsections.

3.2 Formal Modeling

We used the Boston Scientific's system specification for a pacemaker [6]. Because timing constraints are so prevalent in the specification of the pacemaker, it is intuitive and straightforward to use timed automata [7] as our modeling

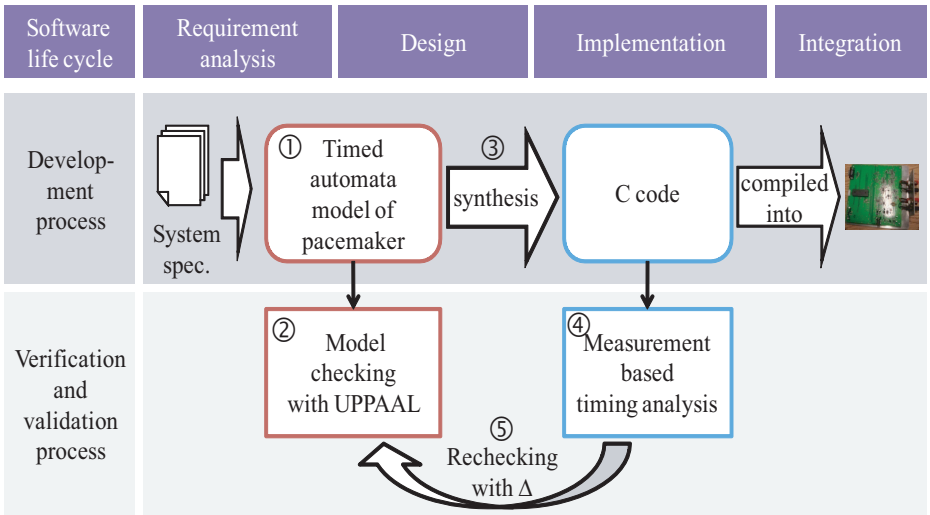


Fig. 1. Overall process of a safety-assured development for a real-time pacemaker software

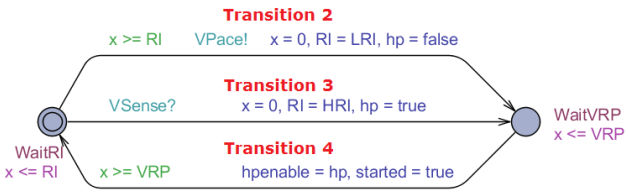
language. Here we use the UPPAAL tool [8] to specify a timed automata model of the pacemaker in VVI mode.

We extracted properties to be satisfied by the VVI mode pacemaker from the system specification. LRI, HRI, and VRP are considered the most important timing periods which should be guaranteed by the VVI mode pacemaker. Fig. 2 shows two automata for Ventricle and Heart, representing the controller for ventricular pacing in the VVI mode and a heart model as the environment for model verification, respectively. Our heart model is the most permissive environment that is ready to accept a pacing signal whenever it is sent and can choose to deliver a sensing signal at any time.

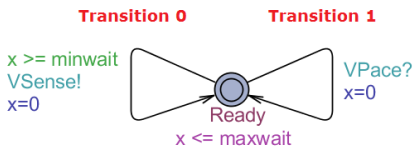
The Ventricle automaton shown in Fig. 2(a) represents sensing signals from the ventricle and emission of ventricular pacing signals to the heart, according to the LRI, HRI, and VRP timing periods. Values of these intervals are captured as parameters of the automaton.

Event channels are used to communicate between pacemaker and its environment. VPace and VSense are channels for sending pacing signals and for receiving sensed events, respectively. A question mark after the channel name represents input from the channel, while an exclamation mark denotes output to the channel. The automaton has two states, WaitRI and WaitVRP, described below.

– WaitRI: The pacemaker starts from this state (denoted by the double circle) and waits for a ventricular sensing or pacing event. If sensing does not occur before the RI period ends, the ventricle controller sends a pacing signal to the heart (Transition 2) and the timer x is reset. The RI value is reset to LRI and hp is set to false, indicating that hysteresis pacing is not used in



(a) Ventricle controller model



(b) Heart model

Fig. 2. Uppaal model for a pacemaker in VVI mode

this case. When a ventricular sense occurs, Transition 3 is taken, where the timer x is reset, hp is set to true and HRI is assigned to RI, which allows a longer period to elapse before pacing . Once the ventricle is paced or sensed, current state is changed to WaitVRP.

– WaitVRP: In this state the pacemaker waits for a VRP period to elapse. It returns to the WaitRI state after a VRP period by setting hpenable to hp and started to true. hpenable and started are auxiliary variables to be used in property description for model checking. started is initially false and holds true after the first visit of WaitVRP.

3.3 Formal Verification

We mapped the timing requirements to the following verification queries in UPPAAL. Below, A□ means that the property must hold in every state along every execution. Notation P.x denotes a variable x defined in the automaton P.

– PropDeadlock: $A\Box(\neg deadlock)$. This property expresses the deadlock freedom in the model.
– PropLRI: $A\Box(\neg \mathsf{Ventricle.hpenable} \Rightarrow \mathsf{Ventricle.x} \leq \mathsf{Ventricle.LRI})$. When hysteresis pacing is disabled, the LRI period should not be exceeded between any two pacing or sensing events.
– PropHRI: $A\Box(\mathsf{Ventricle.hpenable} \Rightarrow \mathsf{Ventricle.x} \leq \mathsf{Ventricle.HRI})$. When hysteresis pacing is enabled, the HRI period is used in place of LRI.
– PropVRP: $A\Box(\mathsf{Ventricle.WaitRI} \wedge \mathsf{Ventricle.started} \Rightarrow \mathsf{Ventricle.x} \geq \mathsf{Ventricle.VRP})$. Except the initial state, the pacemaker can be in the state WaitRI, where sense signals are accepted, only after the VRP period expires.

When we performed model checking on the model shown in Fig. 2 with the above four properties, we confirmed that the model satisfied all these properties.

3.4 Code Generation

We implemented the pacemaker software on a hardware reference platform of the Pacemaker Formal Method Challenge [1], which is based on a Microchip 8-bit PIC18F4520 MicroController Unit (PIC18 MCU) [9] running at 40 MHz clock speed. We generated a single-threaded code where the timed automata models are implemented inside a single loop. The code checks the current enabled transitions and takes one of them in each iteration.

The code generation algorithms adapts the techniques used in the Times tool [10] to produce code for the PIC18 MCU board. While the platform is substantially different from the one supported by Times, the code structure is essentially the same and we can reuse the correctness properties of the Times algorithm.

3.5 Validation of the Generated Code

We utilized MPLAB SIM, a software simulator for PIC18 MCU in the MPLAB Integrated Development Environment (IDE) [9] to execute the code and measure its timing. We tested the generated code under a variety of testing scenarios that cover all sequences of sensing and pacing events of length two that are qualitatively different with respect to the VRP and LRI periods.

Timing analysis of the observed event sequences was used to validate the code. An iteration of the validation cycle (see Fig. 1) was necessary to obtain the bounds on event processing delay, update the model to reflect these delays, repeat the verification, and re-generate the code. Testing of the re-generated code did not reveal any violations of the timing properties. Details of the validation process and timing analysis can be found in [2].

4 Assurance Cases

We created an assurance case to demonstrate that the implemented code is safe to operate, with the intention of providing a guiding example of assurance cases to be possibly used in the certification process of pacemaker software. The assurance case went through multiple review cycles within our group until we were satisfied that no unaddressed arguments result in significant risk to the pacemaker software.

Fig. 3 shows the top-level goal (G1) that the pacemaker software for the VVI mode, implemented as described in Section 3, is acceptably safe. The assurance case is implemented using the goal-structuring notation (GSN) [11]. It concentrates on the pacemaker software, assuming that the hardware platform is reliable (A1). Two context references (C1) and (C2) were added to clarify the goal statement. The assurance case is intended to be a part of the larger case

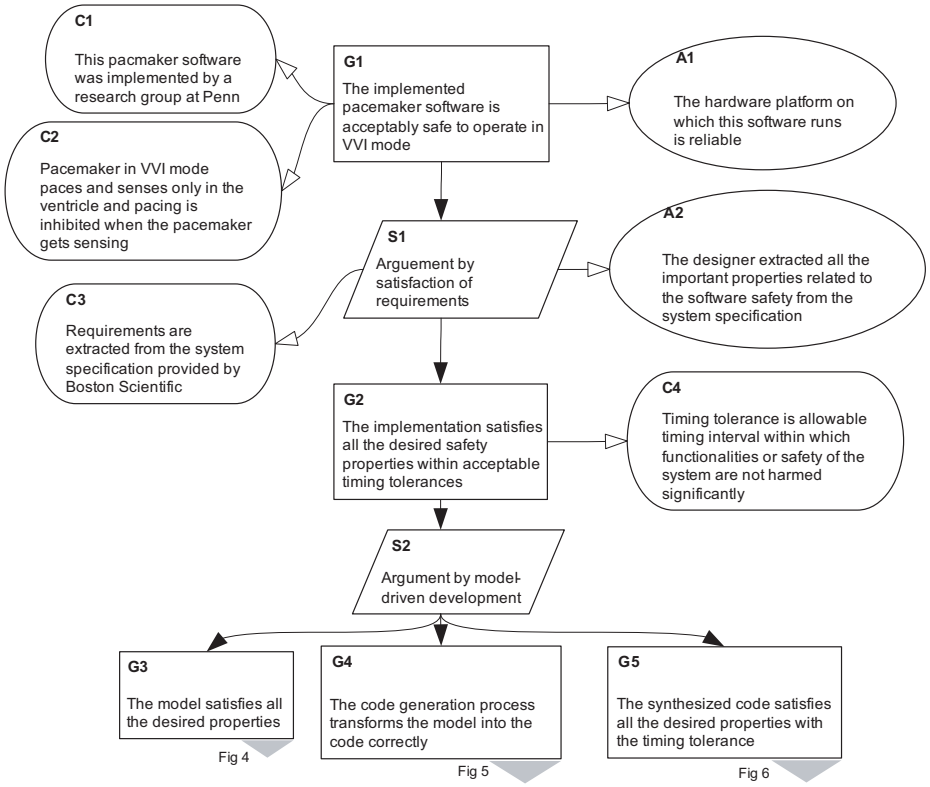


Fig. 3. The pacemaker assurance case - the pacemaker software is acceptably safe

that considers the overall system and makes claims about the assumptions made here.

The element (S1) describes the strategy we are using to argue the goal (G1): it is achieved by satisfying requirements, assuming that the designer extracted all the important properties related to the software safety from the system specification (A2). With this strategy, the goal (G1) is converted into the goal (G2), to show that the implementation satisfies all the desired safety properties within acceptable timing tolerances. Context reference (C4) clarifies the meaning of timing tolerances in this context. Arguing by following the model-driven development approach (S2), the goal (G2) is supported by three subgoals: the model satisfies all the desired properties (G3), the code generation process transforms the model into the code correctly (G4), and the synthesized code satisfies all the desired properties with timing tolerance (G5).

Fig. 4 presents the argument for goal (G3). The model (M1) is the timed automata model of the pacemaker shown in Fig. 2. Four desired properties described in (C5) are described in Section 3.3. Conformance of the model to each

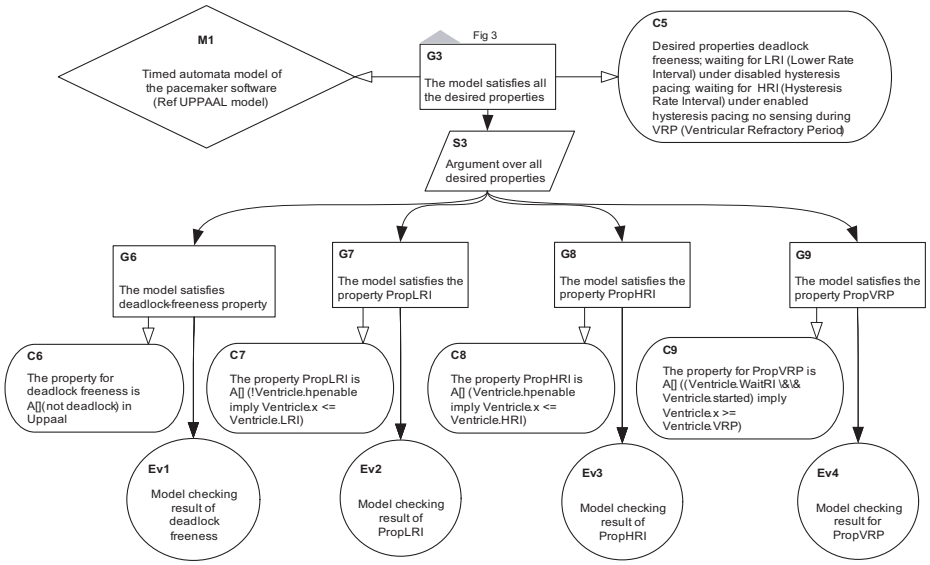


Fig. 4. The pacemaker assurance case - The model satisfies all the desired properties

property is argued by a separate subgoal ((G6)–(G9)), using model checking results as evidence.

Fig. 5 argues the goal (G4). Two strategies (S4) and (S5) were used to split the goal (G4). One of the subgoals supporting (G4) is that, in the context of using the Times tool (C10), the code synthesis of the Times tool for the verified model is correct (G10). Correctness arguments for the the code synthesis of the Times tool given in [12] are used as evidence (Ev5) to support (G10). Since we had to manually modify the code generated by the Times tool to port it on the pacemaker platform, we have to supplement this argument with the claim that manual modifications do not alter correctness of the code. Two subgoals, (G11) and (G12) identify the nature of the modifications, with code review results as evidence (Ev6), and demonstrate that they do not affect the functionality. In the latter case, results of code validation are used as evidence (Ev7).

Fig. 6 addresses the third subgoal of (G2). It argues that the synthesized code satisfies all the desired properties with the timing tolerance (G5). Again, the argument is presented as a separate subgoal for each of the properties. The deadlock freedom property (G13), which does not involve tolerances, is ensured by the guarantees provided by the Times tool, which is used as evidence (Ev8). The other three subgoals, (G14)–(G16), are established through the code level checking based on the justification (J1) that a property in a form of $\mathsf{A}\Box(\mathsf{P})$ can be checked in the code by checking if P is true at the end of every loop with a set of test cases. As shown in Fig. 6 and Fig. 7, (G14), (G15), and (G16) are argued by testing and rephrased by subgoals (G17), (G18), and (G19), respectively.

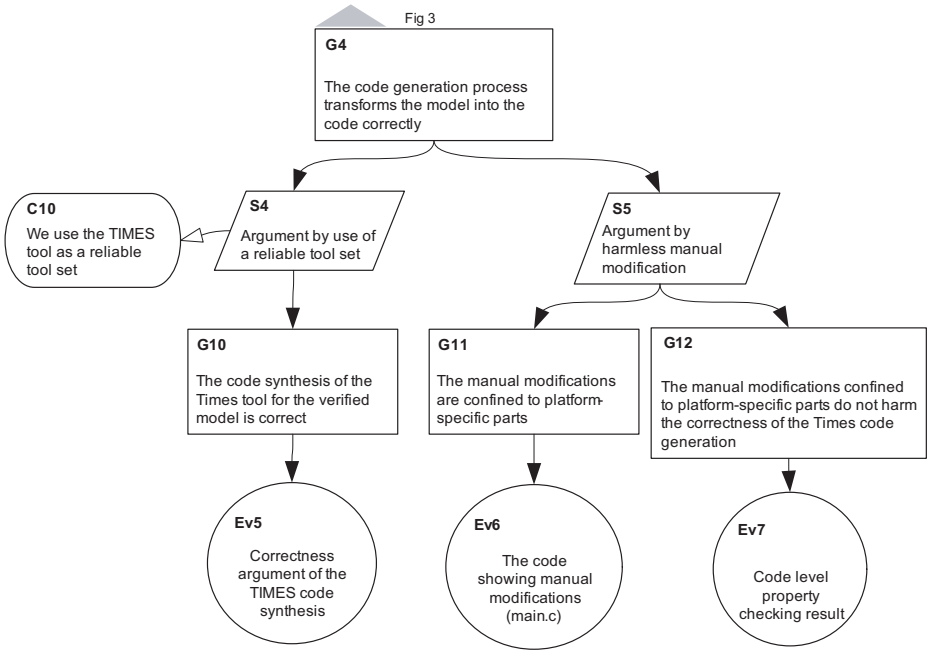


Fig. 5. The pacemaker assurance case - The code generation process is correct

Note that the argument structure for the claim (G16) is simpler than the ones for (G14) and (G15) because the property PropVRP had been satisfied in the code all the time and no alterations were made to the corresponding guards in the code and the model. On the other hand, the properties PropLRI and PropHRI were satisfied in the modified code which involves relaxation in the corresponding guards (See (G17) and (G18) in Fig. 7). The context information for the guard relaxation was described in (C11) which can be instantiated with concrete values.

5 Discussion

Limits of the case study. We begin the construction of the assurance case with the requirements phase of the development. In a real system, the safety argument would also cover hazard analysis and offer claims that hazards are appropriately mitigated by the requirements. We omitted this phase to concentrate on the model-drive aspect of the development process. This decision also matches the current setting of the Pacemaker challenge, which begins with the pacemaker requirements by Boston Scientific. It makes sense to assume that the requirements were properly engineered with respect to hazard.

Similarly, we assume the nominal behavior of the underlying platform. We thus omit the questions of fault tolerance both in the development process and

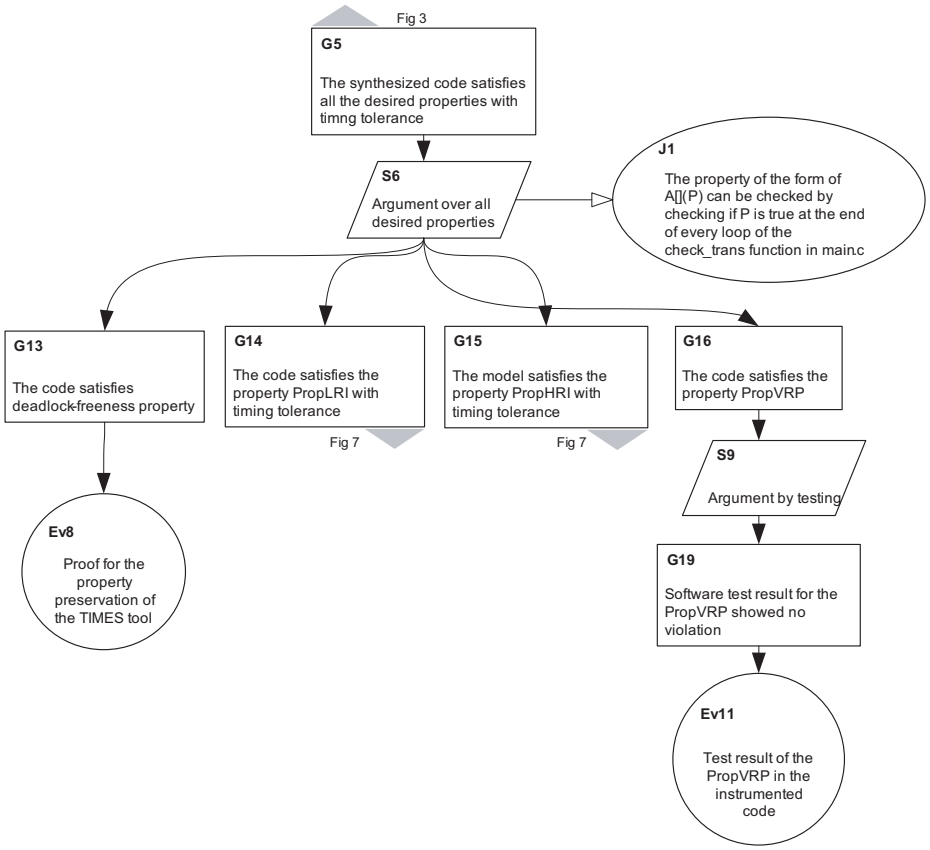


Fig. 6. The pacemaker assurance case - the code satisfies the properties

in the assurance case construction. An assurance case for a complete system will of course have to deal with these issues.

Alternative ways to organize the assurance case. There can be alternative ways to construct the assurance case. When an assurance case has the same or similar structures within it, those common structures can be possibly merged and placed in an upper level. For example, the argument structures for (G14), (G15), and (G16) are similar and they have a common strategy "Argument by testing" as found in (S7)–(S9). "Argument by testing" can be placed in an upper level of (G14)–(G16), accompanied with logically consistent modifications to other parts.

Similarly, "Argument over all desired properties" are also commonly found under (G3) in Fig. 4 and under (G5) in Fig. 6 because we used the same strategy for arguing the property satisfaction in the code as well as by the model. It is possible to change the overall structure of the assurance case by placing "Argument over all desired properties", found in (S3) and (S6), above "Argument by model-driven development" (S2) and modifying other parts consistently. Note

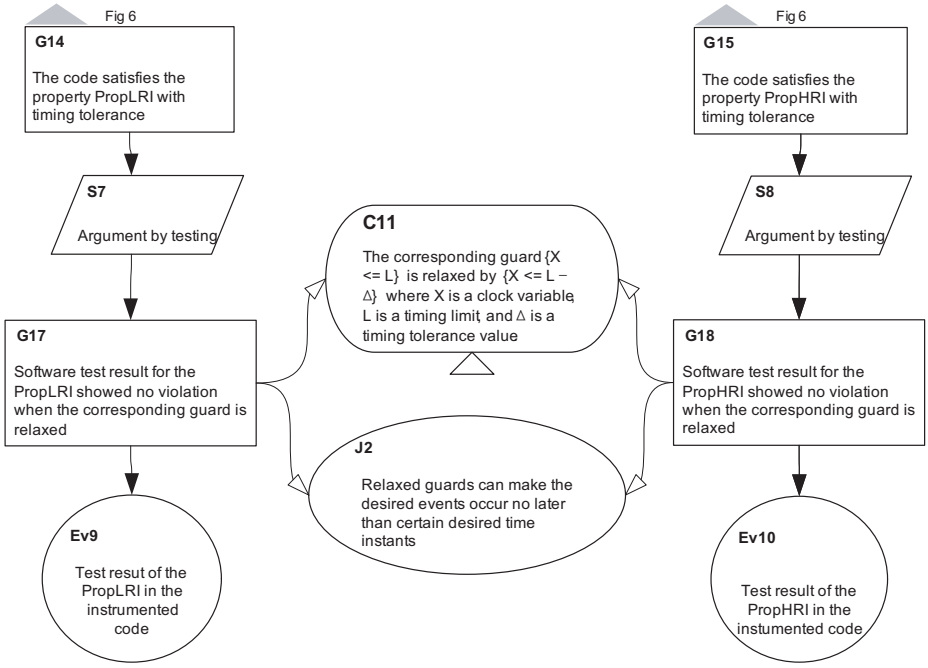


Fig. 7. The pacemaker assurance case - the code satisfies PropLRI/PropHRI

that these modifications do not change the logic of the argument, but may affect the size of the assurance case as we combine common nodes in different branches.

Alternative sources of evidence. In general, argument for a claim can vary and be supported by different kinds of evidence. For example, in our case study we relied on testing to establish timing properties of the generated code. If a higher level of safety is desired, we would resort to more rigorous worst-case execution time analysis using, for example, the aiT tool [13]. However, this change in technology affects only one claim, and the overall structure of the assurance case is not affected.

Ideally, when multiple alternatives can be used as evidence, we should aim to quantify the level of assurance each alternative brings and match it against the level of assurance required for the system. However, quantitative comparison cannot be achieved given today's state of the art. Even qualitative comparison of alternatives is difficult in many cases. This is an important direction of future work for our group.

Significance of the work. We believe that our case study is the first step towards developing assurance case templates for systems developed through model-driven processes. Model-drive development typically includes stages of modeling and model verification, code generation (manual or automatic) with respect to the

model, and validation of the generated code and the whole system. In our approach, each of these stages correspond to a separate claim (or, in general, a set of claims) in the assurance case. This structure makes it more intuitive to follow during the evaluation and provides a clear connection to the evidence obtained in each phase.

6 Related Work

In [14], the authors considered a practice of using assurance cases in the development and approval of medical devices and addressed some of the important issues surrounding the possible adoption of assurance cases by the medical device community. It was mentioned that a set of agreed argumentation patterns (templates) would be useful to manufacturers and reviewers. They suggested that creating and publishing a series of FDA-approvable archetypes for different kinds of medical devices be undertaken to ease the transition of assurance cases into the medical device community. With the same intention as theirs, we took a step forward by developing an argumentation template for another medical device, the pacemaker.

The process of assurance case construction and reuse can become more systematic through documentation of reusable safety case elements as patterns. In [15], 'Safety Case Patterns' for the reuse of common structures in safety case arguments were suggested. Assurance case patterns for security have been studied [16]. Our approach to the assurance case construction presented in this paper may lead to the development of assurance case patterns for model-driven development.

There are other case studies for assurance cases. In [17], the authors described an industrial application of assurance cases to the problem of ensuring that a transition from a legacy system of the Global Positioning System (GPS) to its replacement will not compromise mission assurance objectives. The assurance case demonstrated to the Air Force that the transition posed no major mission assurance concerns and this conclusion was validated by a successful transition.

7 Conclusion

We presented an approach for the construction of assurance cases for the model-driven development of safety-critical software. As a case study, we considered software for a cardiac pacemaker in the VVI mode. The assurance case ties together all the evidence collected during the development process. Several simplifications were applied in the process of constructing the assurance case, to keep the size of the case study under control and to concentrate on the aspects specific to model-driven development.

Future work includes the development of rigorous methods for the evaluation of assurance cases. For bigger systems, we also plan to study compositional construction of assurance cases. This will allow us to simplify certification of component-based systems based on product-line architectures.

References

1. Software Quality Research Laboratory: Pacemaker formal methods challenge `http://sqrl.mcmaster.ca/pacemaker.htm`.
2. Jee, E., Wang, S., Kim, J.K., Lee, J., Sokolsky, O., Lee, I.: A safety-assured development approach for real-time software. In: Proceedings of 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. (August 2010)
3. Adelard: ASCAD – The Adelard Safety Case Development (ASCAD) Manual. (1998)
4. Wassyng, A., Maibaum, T., Lawford, M.: Software certification: The case against safety-cases. In: Proceedings of the Workshop on Modeling, Development, and Verification of Adaptive Computer Systems. (April 2010) To appear.
5. Oregon Health and Science University: Overview of pacemakers `http://www.ohsu.edu/health/health-topics/topic.cfm?id=10395`.
6. Boston Scientific: Pacemaker system specification (January 2007) `http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf`.
7. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
8. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. (November 2004)
9. Microchip: PIC18 family microcontroller `http://www.microchip.com/`.
10. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems. (September 2003) 60–72
11. Kelly, T., Weaver, R.: The goal structuring notation – a safety argument notation. In: Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases. (2004)
12. Amnell, T., Fersman, E., Pettersson, P., Yi, W., Sun, H.: Code synthesis for timed automata. Nordic Journal of Computing **9**(4) (2002) 269–300
13. Ferdinand, C., Heckmann, R.: aiT: Worst-case execution time prediction by static programm analysis. In Jacquart, R., ed.: IFIP Congress Topical Sessions, Kluwer (2004) 377–384
14. Weinstock, C.B., Goodenough, J.B.: Towards an assurance case practice for medical device. Technical Report CMU/SEI-2009-TN-018, CMU/SEI (October 2009)
15. Kelly, T., McDermid, J.: Safety case construction and reuse using patterns. In: Proceedings of the 16th International Conference on Computer Safety, Reliability and Security, Springer-Verlag (1997) 55–69
16. Bloomfield, R.E., Guerra, S., Miller, A., Masera, M., Weinstock, C.B.: International working group on assurance cases (for security). IEEE Security and Privacy **4**(3) (2006) 66–68
17. Nguyen, E.A., Greenwell, W.S., Hecht, M.J.: Using an assurance case to support independent assessment of the transition to a new gps ground control system. In: Proceedings of the International Conference on Dependable Systems and Networks. (June 2008) Anchorage, Alaska.