A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems

Cinzia Bernardeschi, Andrea Domenici, and Paolo Masci

Abstract—This paper presents a methodology, with supporting tool, for formal modeling and analysis of software components in cyber-physical systems. Using our approach, developers can integrate a simulation of logic-based specifications of software components and Simulink models of continuous processes. The integrated simulation is useful to validate the characteristics of discrete system components early in the development process. The same logic-based specifications can also be formally verified using the Prototype Verification System (PVS), to gain additional confidence that the software design complies with specific safety requirements. Modeling patterns are defined for generating the logic-based specifications from the more familiar automata-based formalism. The ultimate aim of this work is to facilitate the introduction of formal verification technologies in the software development process of cyber-physical systems, which typically requires the integrated use of different formalisms and tools. A case study from the medical domain is used to illustrate the approach. A PVS model of a pacemaker is interfaced with a Simulink model of the human heart. The overall cyber-physical system is co-simulated to validate design requirements through exploration of relevant test scenarios. Formal verification with the PVS theorem prover is demonstrated for the pacemaker model for specific safety aspects of the pacemaker design.

Index Terms—C.3.d Real-time and embedded systems, C.4.d Modeling techniques, D.2.1.g Specification, D.2.4.d Formal methods.

1 Introduction

Cyber-physical systems connect and integrate heterogeneous components to create control loops necessary for processing, sensing, and actuation [1]. When developing cyber-physical systems, the need typically arises to simulate, and therefore model, components of different kinds. This happens, for example, when a physiological process (e.g., the heart's electrical conduction system) is monitored and controlled by an embedded digital system (e.g., a pacemaker device). To develop the system effectively, each of the two sub-systems, controlled process and embedded device, should be modeled in the appropriate formalism, also because embedded device experts may not be familiar with physiological process modeling, and conversely.

Many different languages and environments have been introduced to support modeling and simulation. This variety and abundance of tools reflects the fact that different components are best modeled with domain-specific languages and concepts. For example, a tool devised to simulate embedded systems is probably not the best solution to simulate physiological processes. An important distinction among modeling and simulation formalisms is between those oriented to *discrete* systems and those oriented to *continuous* processes. The former can be described as evolving through a discrete set of states, whereas the latter can be described by a set of variables whose values change continuously according to some law, usually defined by differential equations.

In critical application domains such as health care and avionics, developers need to prove that the system design complies with safety requirements. Therefore, besides analyzing the system through simulations, it is also highly desirable (and often mandatory) that the system is assessed using formal verification technologies.

- *C. Bernardeschi and A. Domenici are with the Department of Information Engineering, University of Pisa, Italy. E-mail: cinzia.bernardeschi@ing.unipi.it,Andrea.Domenici@iet.unipi.it*
- *P. Masci is with HASLab/INESC TEC and Universidade do Minho, Portugal. E-mail: paolo.masci@inesctec.pt*

For discrete components of a cyber-physical systems, one class of formal modeling languages that has gained wide acceptance among software designers is the one based on various kinds of automata. Their acceptance rests both on the immediateness of their basic concepts (states and transitions) and their graphical representation. Logic-based formalisms, on the other hand, are also widely used in verification technologies. They may offer more expressiveness power in the specification of system properties beyond the strictly behavioral ones, thanks to the more abstract nature of the concepts they can represent. For example, they have been applied to problems not fitting the automata paradigm, such as aircraft trajectory modeling [2], or used to derive system constraints from a mathematical description [3].

In this work, the *Prototype Verification System* (PVS) [4] is adopted for formal modeling and analysis of discrete system components. PVS is a theorem-proving environment for higher-order logic, provided with an extension package, called *PVSio* [5], that makes it possible to simulate a system specified in the purely logical language of PVS. With this choice, developers may use the same PVS model both for simulation and formal verification.

The continuous part of a cyber-physical system, on the other hand, is usually modeled and simulated with a block-based graphical tool, such as Simulink [6] or ScicosLab [7]. With these tools, a system is specified through a set of interconnected functional blocks, each representing an operation on signals, such as integration, amplification or attenuation, sampling, and so on.

Software engineers would benefit from using all the above tools in combination — each part of the system could be modeled using the most appropriate tool. However, those tools are usually *not interoperable*. To simulate the whole system, models would need to be translated into a common formalism, as each model can be executed only in its native simulation environment. This is not always feasible or convenient, e.g., because environments like Simulink use proprietary languages [8], or because a single environment does not fit all modeling and analysis needs. An approach capable of bridging this gap is *co-simulation* [9], i.e., integrated simulation of different sub-models, each described and

simulated in the respective language and environment.

The main contribution of this paper is a methodology, with supporting tools, whereby a logical model of a discrete system is interfaced to a block-based model of a continuous process. The overall system can be co-simulated, and the discrete part can be formally verified.

This methodology aims at integrating a logic formalism and tool, the *Prototype Verification System* (PVS), with other better known specification languages, and also at integrating formal verification with validation by simulation. The core of the presented approach is a modeling pattern for translating networks of communicating timed automata into a set of logic theories, one for each automaton plus a coordinating theory for the whole network. Tool support is implemented within the *PVSio-web* prototyping toolkit. The theories for the single automaton are generated automatically with the PVSio-web graphical environment. This environment is also used to simulate the modeled system, possibly in conjunction with other models specified with different formalisms, communicating through a messaging protocol. The messaging protocol makes it possible to abstract from the external interfaces of the tools implementing the models, and to run simulations in a distributed environment. This approach is illustrated through a case study in the domain of electronic medical devices, namely, the simulation of a heart-pacemaker system. The approach demonstrated for this case study is easily generalized to other systems modeled with networks of deterministic timed automata.

We may observe that a logic-based formalism such as PVS can also be used to model and verify continuous systems [10], and likewise a block-based tool such as Simulink can be used to simulate discrete systems, for example modeling automata with the Stateflow [11] component. As a matter of fact, the co-simulation framework presented in this paper can connect model of various kind in different combinations. However, the present work is focused on modeling a discrete controller in a cyber-physical system.

Organization. The rest of this paper is organized as follows. Section 2 surveys related work. The paper's contribution is outlined in Section 3. Section 4 introduces the PVSio-web framework, providing essential background on the PVS language and verification technology. Section 5 introduces the case study. Section 6 recalls basic definitions and concepts necessary to model and analyze timed automata and networks of timed automata. Section 7 presents the patterns developed in this work to represent networks of timed automata in the form of logic theories. The method developed to co-simulate a logic model together with a block-based Simulink model is presented in Section 8. Section 9 shows how the logic model used for co-simulation can be used also for verification by theorem proving. Finally, Section 10 concludes the paper.

2 Related Work

The problem of simulating heterogeneous systems has been addressed by many researchers. In this section, five classes of related work are referenced: (i) co-simulation of cyber-physical systems; (ii) formal verification of Simulink models; (iii) tools for the analysis of timed automata in PVS; (iv) work addressing the specific case study presented in the present paper; and (v) work on automata-based specification patterns.

2.1 Co-Simulation of Cyber-Physical Systems

The *Functional Mockup Interface* (FMI) [12], [13] is a standard for model exchange and co-simulation. Model exchange is the ability of a modeling tool to generate a model implementation that can be used as a component (*functional mockup unit*, FMU) in a model executed by another tool, or conversely to use a model generated by another tool as a component. Co-simulation is performed through the execution of different submodels of an overall model by different tools in a distributed environment. The submodels are orchestrated by a master that communicates with them through proxy modules (*FMI wrappers*) whose interfaces are FMI-compliant. The FMI standard is supported by many tools, e.g., HybridSim [14], but all of them are based on the block-based representation of hybrid systems, whereas the tools presented in this work enable co-simulation of models based on different conceptual frameworks, including logic theories, which in turn enable formal verification.

Jalali *et al*. [15] discuss the issues of *multisimulation*, i.e., assembling one complex simulator out of a number of pre-existing heterogeneous simulators. Their work is focused on a framework for the synchronization of the simulators, drawing on the theory of database transactions.

In [16], an approach to co-modeling and co-simulation is presented, based on expressing a discrete-event model and a continuous-time one in the Vienna Development Method (VDM) [17], [18] and in the Bond-Graph notation [19], respectively. The Crescendo tool [20] is used to integrate two simulation environments, Overture [21] for VDM and 20-sim [22] for Bond-Graphs. A key concept is a *contract* listing the information needed to co-ordinate the two models, including shared parameters, variables, and events. A *co-simulation engine* manages synchronization and data exchange.

Attarzadeh, Niaki, and Sander [23] extend the *ForSyDe* modeling framework [24] to support heterogeneous co-simulation. A ForSyDe model is a hierarchical set of processes, where each process may belong to a subset corresponding to a given *Model of Computation* (MoC) [25]. A MoC is, roughly, the underlying time, synchronization, and communication model assumed by a process, and the ForSyDe framework enables processes with different MoCs to co-execute. This framework has been extended with wrapper processes interacting with external simulators or even hardware. In a case study, three parts of a system are modeled by compiled code, VHDL code, and a Simulink model, each executed in the respective execution environment.

A comprehensive survey of other similar co-simulation technologies for cyber-physical systems is in [9].

Whilst these approaches to co-simulation share some similarities with our method, they lack an integrated vision of simulation and formal verification. This integrated perspective is especially important in application domains such as avionics, transportation, and health care, where systems require high level of assurance and system design needs to be proved compliant to given requirements. As has been noted several times by different authors, simulation and testing can be used to prove the presence of anomalies in a system design, not their absence. Formal verification provides a different, complementary perspective, allowing developers to analyze systematically all possible behaviors specified by the system design.

2.2 Formal Verification of Simulink Models

The Simulink Design Verifier [26] is a component of the Simulink environment including a property prover for discrete models. The prover is a model checker based on Stålmarck's proof procedure [27]. Verification is carried out by instrumenting the model with Simulink blocks or Matlab functions specifying *proof objectives* to verify and *proof assumptions* to use in the proof. A library of blocks or functions provides various logical conditions and property templates, including templates for temporal properties. Whilst Simulink Design Verifier can be used to analyze behavioral aspects of the models, the analysis process necessary to perform the analysis of user-defined properties can be quite complex and time consuming, as the tool is highly specialized for the identification of specific types of design anomalies, such as division by zero and buffer overflows. Additionally, the use of model checking poses limits to the size and complexity of the models that can be analyzed within certain time bounds with reasonable resources.

Many alternative approaches to formal verification of a subset of MATLAB/Simulink models have been proposed that are based on model translations. In [28] a translation of discrete-time Simulink models into the synchronous data flow language Lustre is presented; the SCADE design verifier [29] or other model checkers can be applied on the generated models. Similarly, in [30], models in a subset of Simulink are translated into the Boogie [31] verification language annotated with first-order logic formulae and verified using the Z3 theorem prover [32]. The Simulink subset consists of the blocks in the discrete library and stateless blocks.

In [33], Simulink models are annotated with contracts specifying a model's submodels, and represented as *synchronous data flow graphs* [34], from which functionally equivalent sequential programs are generated. These programs are then analyzed by refinement-based techniques [35], and verified with the Z3 theorem prover.

The *CheckMate* environment [36] is a Simulink toolbox (i.e., block library) used to simulate and verify a class of hybrid automata. Verification is done by model-checking ACTL [37] formulae.

Chen *et al*. [38] present a formal representation of Simulink models in *Timed Interval Calculus* [39], which is then translated and verified in PVS.

With these approaches based on model translation, multiple models need to be maintained. In particular, Simulink models need to comply to certain structural properties otherwise the model translators will fail to translate the model. This can be highly inefficient when dealing with complex systems, and in some cases may also prevent re-use of legacy models. Our co-simulation method alleviates these shortcomings, as it does not require model translation.

2.3 Tools for the Analysis of Timed Automata in PVS

TAME (Timed Automata Modeling Environment) [40] supports verification of timed automata by providing two main features: A set of patterns, called *templates*, to represent various classes of timed automata in PVS, and a set of theories and proof strategies built on top of the PVS proof system.

The TAME templates share various common aspects with our modeling patterns for individual automaton models: the state of the automaton is represented as a PVS record type; a dedicated state attribute represents time; transitions between states are represented as PVS functions over states; and predicates over actions check which actions are enabled at each time.

The main differences between our framework and TAME are as follows:

- TAME is more focused on automatic verification, and develops proof tactics based on the templates. Our framework is more focused on validation of automata models and properties, providing an infrastructure for supporting co-simulation of cyber-physical systems.
- Our framework provides a graphical front-end for modeling timed automata. These graphical models are automatically translated into PVS theories, allowing developers without expertise in formal modeling to specify and simulate a system in PVS. In TAME, on the other hand, a front-end for instantiating the templates has not been created, and developers need to manually edit the PVS theories.
- Our framework defines templates for building a network of timed automata out of a given set of automaton models. TAME provides templates only for single automaton models. If developers need to model a network of timed automata in TAME, they need to convert the network into an equivalent single automaton.
- Our framework supports co-simulation of PVS models and Simulink models. In TAME, on the other hand, co-simulation with Simulink is out of scope of the framework. It is also important to remark that the TAME environment cannot be embedded in Simulink using, e.g., an S-Function.

The commonalities between TAME and our framework open new interesting opportunities for development. In fact, our PVS modeling patterns can be converted into a form that is compliant with the TAME templates. Developers using our framework can therefore use TAME tactics during verification attempts. Conversely, TAME templates can also be converted into a form compatible with our modeling patterns. This allows TAME users to exploit our co-simulation engine, as well as to use the Emucharts environment for visualizing automata models.

2.4 Modeling and Analysis of the Heart-Pacemaker System

Formal verification and validation of the whole pacemaker-heart system has been explored in several papers using multiple analysis tools and modeling formalisms.

For example, in [41] and [42], a pacemaker-heart system is verified and validated using Simulink and UPPAAL [43]. The former is used for realistic simulations, the latter is used to verify safety requirements of the pacemaker-heart system using formal methods technologies. *Ad hoc* models are developed in UPPAAL to translate core parts of the Simulink models needed for the verification.

Similarly, in [44], Simulink is used in conjunction with Prism [45]. Ad hoc Prism models are developed to represent the behavior of the pacemaker-heart system and verify pacemaker properties related to energy consumption.

These approaches focus solely on verification, and lack an integrated view verification-simulation. In this case, simulation gives developers a means to validate models and demonstrate analysis results to domain experts. Translating and analyzing Simulink models in tools like UPPAAL or Prism requires manipulating the model, as abstractions need to be used to express the model in a different formalism and to make the analysis

tractable. Establishing a dialogue with medical domain experts is important whenever a model translation is performed, as the model needs to be validated – verification provides strong guarantees about a system if an accurate model of the system is used. This dialogue with domain experts is also important for validating the translation of informal requirements into properties of the model, as well as to check the relevance of counter-examples obtained from failed verification attempts. Tools like UPPAAL and Prism are not designed to support this dialogue with domain experts. Their user interfaces are highly specialized for formal methods experts. Our framework alleviates these shortcomings: model translation is not necessary; validation of formal models of the pacemaker can be carried out through co-simulation with Simulink heart models; full formal analysis of the pacemaker model can be carried out using the PVS theorem prover, using an assume-guarantee style of reasoning; finally, formal analysis results can again be demonstrated using co-simulation.

It is worth noting that formal verification tools for hybrid systems can alleviate challenges related to model translation, as their specification language is expressive enough to capture the dynamics of Simulink models. For example, KeYmaera [46], [47], is a theorem prover for differential dynamic logic. The modeling language offered by the tool is based on first order logic, and includes constructs for expressing conditions, non-determinism, loops, composition, and continuous dynamics. Modal operators are used for expressing state reachability properties of the model. Whilst KeYmaera and other similar tools provide expressive modeling languages and incorporate powerful verification technologies, these tools lack the integrated view verification-simulation necessary to support model validation and demonstration of analysis results.

2.5 Automata-Based Specification Patterns

The modeling patterns introduced in this paper (see Section 7) define a set of logic theories suitable to represent deterministic timed automata and timed automata networks. These patterns were inspired from our previous work on modeling interactive (human-machine) devices [48] and formal analysis of user interface software code [49].

Others have explored the definition of automata-based specification patterns for different purposes. In [50], for example, patterns were introduced for modeling web services based on publish-subscribe as timed automata. Another example is [51], where a library of timed automata models has been developed for representing the characteristics and functionalities of real-time systems. The library in this case was meant to be used by developers as a workbench, to assess the correctness and accuracy of different modeling and analysis tools for real-time systems.

The use of automata-based specification patterns has also been used as a means for translating natural language requirements into temporal logic formulae in a more intuitive manner. The argument is that there is usually a large semantic gap between the formulation of a property in natural language and its corresponding formalization in temporal logic. This gap makes the translation process complex and error prone (see also [52] and [53] for a discussion on the topic). An example of such an approach is [54], where modeling patters based on Büchi automata are introduced for expressing temporal properties over system executions.

3 Contribution

The main contributions of this paper are: (i) the definition of a set of PVS patterns to represent timed automata networks, (ii) a procedure to apply the patterns (Sec. 7), and (iii) a software framework (Sec. 4) where an embedded system described by timed automata can be co-simulated together with a plant modeled as a hybrid system with a block-based language.

The main features of the framework are summarized below:

- a network of timed automata is created by composing deterministic timed automata developed with the graphical editor of the PVSio-web framework;
- the network is automatically translated into logic theories according to the proposed patterns;
- the resulting logic specification is amenable both to verification and simulation with the PVS theorem prover and its PVSio extension;
- the logical specification can be co-simulated with a block-based model using the PVSio-web framework;
- two interface subsystems connect the respective models to a WebSocket [55] communication framework. Their purpose is to intercept simulation events generated by the two models, and forward them from one simulation environment to the other.

The logical patterns and the co-simulation framework are shown through a running a example. An implantable cardiac pacemaker (ICP) model, originally described using timed automata, is specified in PVS and is executed in a PVSio process, and a heart model is specified in the Simulink language and executed by the Simulink tool.

The two modules communicate through the interface subsystems and operate as follows:

- The ICP model interface is part of the PVSio simulation environment. It uses a WebSocket connection to receive events (atrial and ventricular signals) from the heart model. The same WebSocket connection is used to send pacing events generated by the ICP simulation to the heart simulation.
- The heart model interface is part of the Simulink model. It receives pacing events over a WebSocket connection from the ICP model interface. The events are injected into the heart simulation. Atrial and ventricular events from the heart simulation are sent to the ICP simulation using the same WebSocket connection.

The above approach extends the framework presented in [56] for integrated simulation of PVS models and models described in Simulink. In that work, the WebSocket framework was used to generate infusion pump simulations, where the user interface component is developed in PVS, and the pump controller is developed in Simulink. Preliminary work on the simulation of the heart-pacemaker system appeared in [57].

4 The PVSio-web Framework

The PVSio-web framework [58], [59] has been originally designed to validate the user interface of medical devices by interactively animating a formal specification of the user interaction with the device. This specification can be written directly in the PVS language, or entered graphically as a state machine diagram that is automatically translated into the PVS language.

PVSio-web is implemented in JavaScript by a software platform composed of several scripts, invoked and coordinated

through a web interface. The main components of the framework are the model builder, the Emucharts editor, and the simulator. The model builder is used to create a realistic graphical interface of the device to be studied. Typically, a picture of the device's front panel is displayed in the model builder, and the user may associate regions of the picture (e.g., buttons, knobs, or displays) to PVS functions that simulate user input or device output. A PVS model generator for the Emucharts editor has been implemented, which enables automatic generation of PVS executable functions from state machine diagrams drawn with the Emucharts editor. The modeling patterns used in the model generator are presented in Section 7. The simulator is a PVSio process that is invoked to interactively exercise the device model (see [60] for a detailed description of the functionalities and workflows supported by PVSio-web).

PVS is the main verification technology used by PVSio-web. It is an interactive theorem prover, enabling users to define theories and prove theorems within them. Theories are written in a typed higher-order language, where the user can define complex types and express properties of higher-order concepts, such as functions and sets. The theorem prover provides an extensive number of inference rules based on the sequent calculus [61], which the user can select and apply in different proof steps. The proof is not fully automatic but computer-assisted; the inference rules, however, are very powerful and experienced users may find proofs in a short time.

This paper is not mainly concerned with the theorem-proving applications of PVS, but with its use as a prototyping tool. This use is made possible by the PVSio extension. PVSio is a ground evaluator that computes the value of ground (variable-free) expressions. The evaluator can also compute functions with side effects, such as producing outputs. It should be noted that functions with side effects are logically equivalent to normal (i.e., purely logical) functions of the PVS language, so that they do not interfere with theorem proving. The PVS theorem prover can be started in PVSio mode, where it accepts as inputs ground function expressions to evaluate. In this mode, the PVSio evaluator functions as an interpreter for a logic programming language.

4.1 Background on the PVS language and the Sequent Calculus

The PVS specification language provides the usual base types, such as Booleans, naturals, integers, reals, and others. More complex types can be defined, including *function types* denoted by type expressions of the form $[domain \to codomain]$, where *domain* and *codomain* can be any type, including function types. Functions with the Boolean codomain type are called *predicates*.

PVS specifications are included in *theories*. Formulas and definitions of a theory may refer to and be proved with formulae and definitions from other theories made accessible by *IMPORTING* declarations. A set of fundamental theories, called the *prelude*, is imported implicitly, and additional libraries provide a large number of theories containing standard definitions and proved facts, e.g., about sets, sequences, and graphs.

The syntax of the PVS language is quite complex, and its basic constructs will be shown in examples throughout the paper. A few constructs and conventions, however, should be known beforehand:

- Comments extend from a '`%`' character to the end of line.
- If p is a predicate over a set S, (p) denotes the subset of S whose elements satisfy p.
- A *record* is a tuple whose elements are referred to by their respective *field* name. For example, given the declarations:

```
complex: TYPE = [#          % record type
    r: real,
    i: real
    #]
c: complex = (# r := 1.0, i := 2.0 #)
                            % record literal
```

  the expressions $r(c)$ and $i(c)$ denote the real and the imaginary part of c. Equivalent notations are $c`r$ and $c`i$.
- The *overriding* operator `:=` in a *WITH* expression redefines record fields. With the declarations above, the expression

```
c WITH [ r := -1.0 ]
```

  denotes the complex value $(-1.0, 2.0)$. Note that c is left unchanged.
- A *LET* ... *IN* construct introduces definitions in the following expression, as in

```
LET a = c WITH [ r := -1.0 ],
    b = (# r = 1.0, i = 0.5 #)
IN x = a + b
```

- Function declarations are in the form

```
foo(x: T1): T2
```

  where *foo* is the function name, x is a function argument, of type *T1*, and *T2* is the function return type.

The sequent calculus works on expressions, called *sequents*, of this form: $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$, where the A_i's are the *antecedents* and the B_i's are the *consequents*. The '$\vdash$' symbol is called a *turnstile* and may be read as "*entails*" or "*yields*." Each antecedent or consequent is a formula of any form (in the underlying logic language), except another sequent.

The inference rules of the sequent calculus are used to transform sequents. Some rules transform one sequent into two or more new sequents, so that a proof can be represented as a tree whose nodes are sequents and whose arcs are applications of inference rules. A proof terminates successfully when all branches terminate with a proved sequent, i.e., one where either any formula occurs both as an antecedent and as a consequent, or any antecedent is false, or any consequent is true.

5 Case Study Overview

This section introduces the case study chosen to illustrate the presented tools and methods, first sketching the structure of the already available model of the heart, and then describing the expected behavior of an ICP.

5.1 Simulink Model of the Heart

Simulink is a well known and widely used component of the Matlab tool. A Simulink model is built of functional blocks chosen from a vast block library. A fundamental block is the integrator, used to represent a differential equation. The core of the simulation engine is an Ordinary Differential Equation (ODE) solver, which can be selected from a supplied set according to the mathematical characteristics of the simulated system.

The ODE solver integrates functions with respect to time, advancing time by a given (possibly variable) increment at each integration step. At user-specified intervals, the output of each block is sampled and fed to the next block(s) downstream.

The available Simulink library can be extended with user-defined custom blocks. The behavior of a custom block is defined

by a user-supplied function called an S-function and written either in the Matlab programming language or in C or C++.

Models of the heart have been built [62], [63] with *hybrid automata* (HA) [64].

HA are characterized by a finite set of locations, representing distinct modes of operations, and a finite set of variables, representing time-varying quantities, such as speed or temperature, or stock market quotations, or morbidity rates. These quantities vary continuously with time according to some mathematical law, e.g., differential equations, and in different locations they may follow different laws.

In this paper, we use the Simulink model developed by Chen *et al.* [62]. In their model, the heart's electrical conduction system is specified as a network of HA implemented in Simulink. The HA representing ventricular cells have four modes: *resting and final repolarization*, *stimulation*, *upstroke*, and *plateau and early repolarization*. In each mode, the membrane voltage follows a specific differential equation. The complete Simulink model consists of over 200 functional blocks. A detailed illustration of the model is in [62]. Here, we illustrate the overall architecture and the input and output parameters of the model, as this is sufficient for the scope of this work.

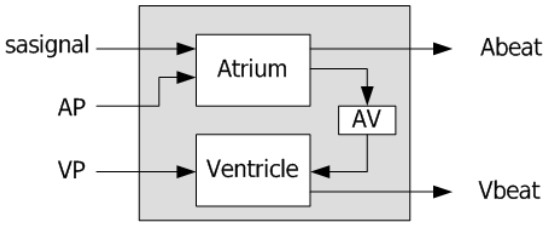

Fig. 1. Architecture of the heart model.

The heart model has two main functional modules, `Atrium` and `Ventricle`, representing the electrical behavior of the atrium and ventricle (see Fig. 1). The two modules communicate through an `AV` module, which represents the atrioventricular node of the heart. Two input parameters allow designers to inject pacemaker signals in the heart: *AP* (Atrial Pacing), is used to inject the pacing stimulus generated by the pacemaker in the atrium; *VP* (Ventricular Pacing), is used to inject into the ventricle the pacing stimulus generated by the pacemaker. Another input, *sasignal* (Sinoatrial node signal), represents the firing frequency of the impulse-generating tissue of the heart. This input can be used to change the heart behavior and explore different scenarios (e.g., normal sinus rhythm, bradycardia, tachycardia). Two output parameters, *Abeat* and *Vbeat*, can be used to check whether the electric signal from the atrium and the ventricle has reached given thresholds.

5.2 Pacemaker Operation

Cardiac rhythm results from a complex electrochemical and biological process, where electrical signals are carried by variations of ionic concentrations across cell membranes. Schematically, signals are originated in the tissues of the *sinoatrial* (SA) node and propagated to the muscle tissues of the atria and then, after a propagation delay, to the muscle tissues of the ventricles. The SA signals determine the heart rate, and the propagation delays determine the interval between atrial and ventricular contractions.

A cardiac pacemaker maintains the cardiac rhythm within its physiological range by detecting pathological deviations of the heart rate (*tachycardia* and *bradycardia*) and reacting by issuing electrical pulses. A pacemaker must ensure that cardiac cycles occur at the correct rate, and that contractions of the atria and ventricles are appropriately separated.

A dual chamber pacemaker detects *atrial* and *ventricular sense events* (AS and VS, respectively) and issues *atrial* and *ventricular pulse events* (AP and VP), whose relative timing is constrained by a set of parameters. A sense event occurs when either chamber contracts, and a pulse event occurs when the pacemaker stimulates either chamber. We observe that the AP and VP events, besides being generated by the pacemaker, are also used within its internal logic, as will be shown in Section 7.

Following the schematization presented in [65], the timing parameters are:

LRI *Lower Rate Interval*, is the maximum allowable separation between ventricular events;

URI *Upper Rate Interval*, is the minimum allowable separation between atrial events;

AVI *Atrioventricular Interval*, is the maximum allowable separation between an atrial and a ventricular event;

PVAB *Postventricular Atrial Blanking*, is the period after a ventricular event during which atrial events are ignored;

PVARP *Postventricular Atrial Refractory Period*, is the period following the PVAB during which the pacemaker responds to atrial events by issuing an atrial event rejection (AR) action;

VRP *Ventricular Refractory Period* is the period after a ventricular event during which further ventricular events are ignored.

The timing enforced by the pacemaker can then be described as follows, using timed automata (background notions on timed automata are in Section 6):

1) After a ventricular event, the next atrial event must occur within an interval equal to the difference between the LRI and the AVI. If no AS event occurs within that interval, the pacemaker issues an AP when the interval expires, as specified by the LRI automaton in Fig. 2.
2) After an atrial event, a ventricular pacing pulse is issued if no ventricular event has occurred within the AVI, unless the time since the last ventricular is less than the URI. In this case, the ventricular pacing pulse is issued when the URI expires, as specified by the AVI automaton in Fig. 2.
3) After a ventricular event, atrial events are ignored during the PVAB and the PVARP, as specified by the PVARP automaton in Fig. 2. The AR event is not used in the pacemaker model considered in this work.
4) After a ventricular event, further ventricular events are ignored during the VRP, as specified by the PVARP automaton in Fig. 2.

6 Background on Timed Automata

This section introduces background notions on timed automata as used in [62] to formally specify the cardiac pacemaker.

A *timed automaton* (TA) [66] is a graph characterized by a finite set of *locations* with one *initial* location, a finite set of variables over the non-negative reals, called *clocks*, a finite set of *actions*, a finite set of predicates on clock values, called *constraints*, and a finite set of *edges* connecting locations. Each edge is labeled with one action, one constraint, and a set of zero or more clocks. The *state* of a TA is given by the location and the values of the clocks at a given time.

The identically *true* constraint, i.e., the constraint that is always satisfied, is omitted from the graphical representation of a TA.

Intuitively, a TA models a system operating in a number of distinct modes represented by the locations, switching between them when actions labeling an edge occur, provided the corresponding constraint is satisfied. If the edge is labeled with a set of clocks, those clocks are reset. While the system remains in a given location, the progress of time is reflected by the values of the clocks, whose values increase all at the same rate. In the case study discussed in this paper, timing constraints in the pacemaker are specified by TA.

The original concept of TA has been extended over the years by different researchers, to allow the composition of TA into *networks* of TA. In the present work, as in [65], we use a subset of the theory for networks of TA implemented in the UPPAAL model checker [67]. In UPPAAL, synchronization between two automata is modeled by the existence in the network of two edges, one for each of the two automata, labeled with *complementary* actions. The set of synchronized actions in the network is partitioned into two subsets of equal cardinality, such that each action in one set has a complementary action in the other one. The two subsets are called the sets of *input* and *output* actions, respectively. One automaton executing an output action synchronizes with one or more automata, each executing the complementary action. Two edges labeled with complementary actions can be taken only if the constraints on each of them, if any, are satisfied. Actions not participating in synchronizations, i.e., *internal* actions, are all equivalent with respect to the network behavior, and are represented by the τ action.

Graphically, input actions are denoted by a question mark ($?$) and output actions by an exclamation mark ($!$). The τ label is usually omitted.

Another important extension to TA as defined by Alur and Dill [66], used in UPPAAL, are *location invariants* [68], i.e., constraints on clock values that must hold while an automaton is in a given location.

The above concepts are defined in the literature with slightly different terminology and notation. The following definitions derive from those in [69], [70], [71], [72].

Definition 1 (Clocks and time constraints)**.** *Let $\mathcal{C}$ be a finite set of variables (called* clocks*) ranging over $\mathbb{R}_{\geqslant 0}$. A* simple time constraint *is an expression of the form $x \bowtie c$ or $x - y \bowtie c$, with $x, y \in \mathcal{C}$, $c \in \mathbb{N}$, and $\bowtie \in \{<, \leqslant, >, \geqslant\}$. A* time constraint *is a simple time constraint, or a conjunction, disjunction, or negation of time constraints.*

A time constraint can be a *guard* or an *invariant* (see below). Please note that, as in [72], we consider invariants of the form $x < c$ or $x \leqslant c$.

Definition 2 (Resets)**.** *$R = 2^{\mathcal{C}}$ is the set of* resets*.*

A reset is a clock to be set to zero.

Definition 3 (Invariants)**.** *Let $\mathcal{B}(\mathcal{C})$ be a set of time constraints over $\mathcal{C}$, and $\mathcal{L}$ a finite set of* locations*. $I(\mathcal{L})$ is the set of* invariants*, with $I : \mathcal{L} \to \mathcal{B}(\mathcal{C})$.*

Definition 4 (Actions)**.** *$\Sigma = \Sigma_{\text{in}} \cup \Sigma_{\text{out}} \cup \{\tau\}$, with $\Sigma_{\text{in}} \cap \Sigma_{\text{out}} = \emptyset$, is a finite set of actions, partitioned into* input *actions (Σ_{in}),* outputs *actions (Σ_{out}), and the* internal *action (τ).*

Action τ represents any internal action not involving communication with other automata.

Definition 5 (Timed I/O Automaton)**.** *A* timed input/output automaton *is a tuple $\mathbf{A} = (\mathcal{L}, l_0, \mathcal{C}, E, \Sigma, I)$, where*

- *$\mathcal{L}$ is a finite set of* locations*;*
- *$l_0 \in \mathcal{L}$ is the* initial *location;*
- *$\mathcal{C}$ is a finite set of* clocks*;*
- *$E \subseteq \mathcal{L} \times \Sigma \times \mathcal{B}(\mathcal{C}) \times R \times \mathcal{L}$ is the set of* edges*;*
- *$I : \mathcal{L} \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations.*

In an edge (l, a, g, r, l'), $g \in \mathcal{B}(\mathcal{C})$ is called a *guard*.

In the present work, we consider *deterministic* timed I/O automata, i.e., such that any edges labeled with the same action have mutually exclusive guards.

Definition 6 (Clock valuation)**.** *A* clock valuation *is a function $v : \mathcal{C} \to \mathbb{R}_{\geqslant 0}$. We let $v+t$ denote the valuation that maps each clock x to $v(x)+t$, and we let $[r \mapsto 0]v$, with $r \in R$, denote a valuation that maps each clock in r to zero and agrees with v elsewhere. A valuation* satisfies *a set of constraints if the conjunction of all constraints in the set holds for the clock values assigned by the valuation. We let $\mathcal{V}_{\mathcal{C}}$ denote the set of all valuations over $\mathcal{C}$.*

For readability, in the following we let $l \xrightarrow{a,g,r} l'$ stand for $(l, a, g, r, l') \in E$, and we let $l \xrightarrow{\alpha} l'$ stand for $(s, \alpha, s') \in \to$, with $\alpha \in \Sigma \cup \mathbb{R}_{\geqslant 0}$.

Definition 7 (Semantics of a Timed I/O Automaton)**.** *The semantics of a timed I/O automaton is captured by a Timed I/O Transition System (TIOTS), and is defined as a quadruple $\mathcal{S} = (S, s_0, \Sigma, \to)$, where:*

- *$S \subseteq L \times \mathcal{V}_{\mathcal{C}}$ is a finite set of* states*;*
- *$s_0 \in S$ is the* initial state *$(l_0, 0)$;*
- *Σ is the set of actions of the automaton;*
- *$\to : S \times (\Sigma \cup \mathbb{R}_{\geqslant 0}) \times S$ is a* transition relation*.*

The transition relation is defined by the following rules, where $l, l' \in \mathcal{L}$, $g \in \mathcal{B}(\mathcal{C})$, $r \in R$, and $v, v' \in \mathcal{V}_{\mathcal{C}}$:

- *$(l, v) \xrightarrow{d} (l, v+d)$* if *$d \in \mathbb{R}_{\geqslant 0}$* and *$\forall t : t \in [0, d] \Rightarrow v+t$* satisfies *$I(l)$.*
- *$(l, v) \xrightarrow{a} (l', v')$* if *$a \in \Sigma$* and *$l \xrightarrow{a,g,r} l'$* and *$v$* satisfies *$g$* and *$v' = [r \mapsto 0]v$* and *$v'$* satisfies *$I(l')$.*

Single timed automata models can be composed to create networks of Timed I/O Automata (defined below). This is useful to build complex models using simpler models as building blocks. In this work, this approach is used to develop the full pacemaker model.

Definition 8 (Network of Timed I/O Automata)**.** *A* network *of timed automata is a finite set $\{\mathbf{A}_1, \ldots, \mathbf{A}_n\}$ of timed automata. A* location vector *is a set $\boldsymbol{l} = \{l_1, \ldots, l_n\}$, where $l_i \in \mathcal{L}_i$, $i = 1, \ldots, n$. The* initial location vector *$\boldsymbol{l}_0$ is the set $\{l_{01}, \ldots, l_{0n}\}$ of initial locations. We let $\boldsymbol{l}[l_i'/l_i]$ denote the location vector where the i-th element l_i of $\boldsymbol{l}$ has been substituted with l_i'. The* network actions *are the set $A = \bigcup_{i=1}^{n} \Sigma_i$. The* network clocks *are the set $\boldsymbol{C} = \bigcup_{i=1}^{n} \mathcal{C}_i$. The* network invariants *are the set $I(\boldsymbol{l}) = \bigcup_{i=1}^{n} I_i(\mathcal{L}_i)$. We let $\mathcal{V}_{\boldsymbol{C}}$ denote the set of all valuations over $\boldsymbol{C}$.*

Definition 9 (Complementary actions)**.** *For any $i, j \in [1, n], i \neq j$, two actions of the same name $a? \in \Sigma_{\text{in}}^{i}$ and $a! \in \Sigma_{\text{out}}^{j}$ are said to be* complementary*, or* matching*.*

In a network of timed I/O automata, the complementary actions become the *network internal* action, denoted by τ.

Definition 10 (External actions)**.** *For any $i \in [1,n]$, an action $a \in \Sigma^i_{\text{in}}$ ($a \in \Sigma^i_{\text{out}}$) without a complementary action belongs to the set $X \subseteq \Sigma$ of* external *actions. Set X is partitioned into the disjoint sets of input (X_{in}) and output (X_{out}) external actions.*

In this paper, external actions are used to model the sensing activity (signals Abeat and Vbeat) and the actuation actions (signals AP and VP) performed by the pacemaker.

Definition 11 (Semantics of a network of timed I/O automata)**.** *The semantics of a network of timed automata $N = \{\mathbf{A}_1, \ldots, \mathbf{A}_n\}$ is a transition system $(\mathbf{S}, \boldsymbol{s}_0, A', \rightarrow)$, where:*

- $\mathbf{S} = \mathcal{L}_1 \times \cdots \mathcal{L}_n \times \mathcal{V}_{\mathbf{C}}$ *is the set of* network states*;*
- $\boldsymbol{s}_0 = (\boldsymbol{l}_0, \boldsymbol{v}_0)$ *is the* network initial state*, where $\boldsymbol{v}_0 \in \mathcal{V}_{\mathbf{C}}$ is the* network initial valuation*; and*
- $A' = A \cup \tau$*;*
- $\rightarrow \subseteq \mathbf{S} \times (A' \cup \mathbb{R}_{\geqslant 0}) \times \mathbf{S}$ *is the* network transition relation*.*

The network transition relation is defined by the following rules:

1) $(\boldsymbol{l}, v) \xrightarrow{d} (\boldsymbol{l}, v+d)$ if $d \in \mathbb{R}_{\geqslant 0}$, v *satisfies* $I(\boldsymbol{l})$, and $(v+d)$ *satisfies* $I(\boldsymbol{l})$.
2) $(\boldsymbol{l}, v) \xrightarrow{\tau} (\boldsymbol{l}[l'_i/l_i], v')$ if $l_i \xrightarrow{\tau,g,r} l'_i$, v satisfies g, $v' = [r \mapsto 0]v$, and v' satisfies $I(\boldsymbol{l}[l'_i/l_i])$.
3) $(\boldsymbol{l}, v) \xrightarrow{\tau} (\boldsymbol{l}[l'_i/l_i][l'_j/l_j], v')$ if $\exists\, i \neq j$ such that
   - $l_i \xrightarrow{a?,g_i,r_i} l'_i$, $l_j \xrightarrow{a!,g_j,r_j} l'_j$,
   - v satisfies $g_i \wedge g_j$,
   - $v' = [r_i \cup r_j \mapsto 0]v$ and v' satisfies $I(\boldsymbol{l}[l'_i/l_i][l'_j/l_j])$.
4) $(\boldsymbol{l}, v) \xrightarrow{a?} (\boldsymbol{l}[l'_i/l_i], v')$ if $a? \in X_{\text{in}}$, $l_i \xrightarrow{a?,g,r} l'_i$, v satisfies g, $v' = [r \mapsto 0]v$, and v' satisfies $I(\boldsymbol{l}[l'_i/l_i])$.
5) $(\boldsymbol{l}, v) \xrightarrow{a!} (\boldsymbol{l}[l'_i/l_i], v')$ if $a! \in X_{\text{out}}$, $l_i \xrightarrow{a!,g,r} l'_i$, v satisfies g, $v' = [r \mapsto 0]v$, and v' satisfies $I(\boldsymbol{l}[l'_i/l_i])$.

In the above definition, rule 1 describes state changes due to the passing of time, and rules 2 and 3 describe changes due to internal actions of single automata or, respectively, input/output synchronizations within pairs of automata. Rules 4 and 5 describe synchronization between an automaton and the external environment, i.e., inputs (outputs) to (from) the network.

Additional definitions will be explained in the rest of this paper when necessary.

7 PVS Patterns for Networks of Timed Automata

Following previous work by other authors [41], [42], [44], [62], in this paper a dual-chamber pacemaker is modeled as a network of timed automata. This section introduces the basic notions on networks of timed automata and the modeling patterns we adopt to represent the timed automata in the PVS language.

7.1 The Pacemaker Network

Fig. 2 (adapted from [65]) shows the network for the pacemaker. Each automaton enforces some constraint on the intervals between pairs of events, using clocks named t (this name is local to each automaton) or *clk* (this clock is accessed both by the URI and the AVI automata). For example, the LRI automaton keeps the heart rate above a minimum value, defined by the *Lower Rate Interval* (TLRI). The automaton starts in the LRI location. Upon a ventricular event (VP? or VS?), clock t is reset. Upon an atrial sense event (AS?), the automaton waits in location ASed until a ventricular event occurs, which causes the automaton to return to the LRI location, resetting t. Since an atrial event must occur within $\text{TLRI} - \text{TAVI}$ seconds after the last ventricular event, an atrial pulse (AP!) must occur if the automaton remains in LRI for a longer interval. This is specified by the invariant $\{t \leqslant \text{TLRI} - \text{TAVI}\}$ on location LRI and by the edge labeled with the guard $[t \geqslant \text{TLRI} - \text{TAVI}]$, the action AP!, and the reset of t. This models the issue of an atrial pacing pulse by the pacemaker.

7.2 PVS Specification of a Timed Automaton

The concepts of TA can be expressed in the higher-order logic language of the PVS. In particular, the pacemaker model discussed above has been translated into a set of PVS theories. These theories are divided in two layers: the set of theories defining each single automaton, and a coordinating theory defining their interaction in a network.

Let us consider first how the single automata are defined:

- The state of an automaton is defined by a record type with one field representing the current location, plus one real-valued variable for each clock.
- For each action a defined for the automaton, an *enabling* predicate checks the enabling conditions for a. These conditions depend on the current location and on guards and invariants.
- For each action a defined for the automaton, a *transition function* returns the next state as specified by the edges labeled with a.
- A *time-checking* predicate checks the enabling conditions for the internal actions τ.
- A *timing* function returns the next state as specified by the edges labeled with τ.

The following PVS fragment shows part of the PVS model for the LRI timed automaton of Fig. 2. In the PVS code, suffixes `in` and `out` represent the *?* and *!* annotations of TA actions.

```
LRI: THEORY BEGIN
IMPORTING constants

Mode: TYPE = { LRI, ASed }
state: TYPE = [#
    time: real,
    loc: Mode
#]

init_LRI: state = (# time := 0, loc := LRI #)

% enabling predicates for AP!, AS?, VP?, VS?
en_APout(st: state): boolean =
    loc(st) = LRI AND time(st) >= TLRI-TAVI
en_ASin(st: state): boolean = loc(st) = LRI
en_VPin(st: state): boolean = true
en_VSin(st: state): boolean = true

% transition functions for AP!, AS?, VP?, VS?
APout(st:(en_APout)):state = (# time := 0, loc := LRI #)
ASin(st: (en_ASin)): state = st WITH [ loc := ASed ]
VPin(st: (en_VPin)): state = (# time := 0, loc := LRI #)
VSin(st: (en_VSin)): state = (# time := 0, loc := LRI #)

% time-checking predicate
en_tau(st: state): boolean = false

% timing function
tau(st: state): state = st
END LRI
```

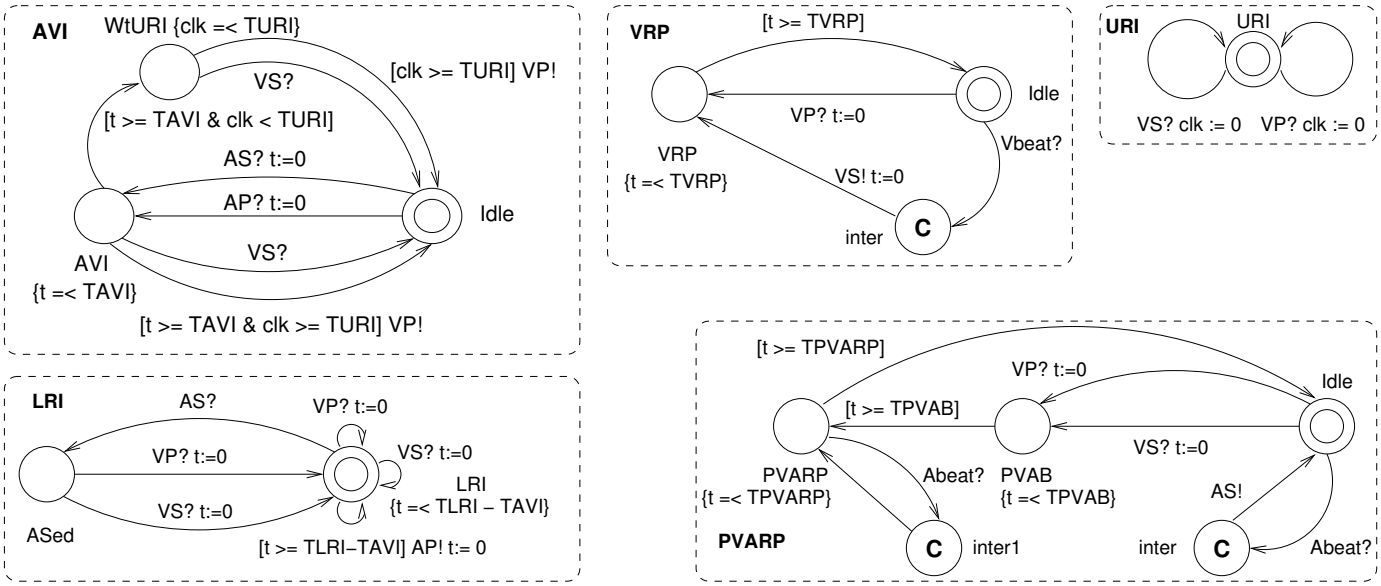

Fig. 2. TA network for the pacemaker. Each automaton implements the constraints defined by the respective timing parameter (e.g., TAVI in the AVI automaton). Square brackets enclose guards and curly brackets enclose state invariants. Two concentric circles denote the initial location. Letter C denotes committed locations (Sec. 7.3).

The first two declarations define the set of locations and the structure of the state record, containing the *loc* and *t* fields.

Functions *en_APout* and *APout* are the enabling predicate and the transition function, respectively, for action AP!. Similar pairs of enabling predicate and transition functions are declared for the other actions affecting the automaton.

Function *APout* is the output action of the automaton, and functions *ASin*, *VPin*, and *VSin* are input actions. Note that each function definition uses subtyping to restrict the function domain to the set of states where the action is enabled. This is denoted by specifying the type of the function argument as a predicate adorned with round brackets. For example, action *APout* is enabled only when predicate *en_APout* is true, and the type of the function argument is therefore *(en_APout)*.

Predicate *en_tau* is the time-checking predicate and *tau* is the timing function, which in this case is a no-operation since there are no internal actions. This pair of predicate and function, however, is declared in all automata in order to achieve a more uniform treatment of automata synchronization, as discussed in Section 7.3 below.

The full PVS model of the ICP is in the Appendix.

7.3 Specification of the Network of Timed Automata

A *network* of timed automata is a set of timed automata synchronized through *communication events* (or simply *events*), i.e., coincident executions of one output action and one or more complementary input actions. Consider, for example, the VRP automaton in Fig. 2. The automaton has input actions VP? and Vbeat?, and the output action VS!. All the other automata have an input action VS?. When transitions in the network labeled with VS! or VS? are enabled, they are executed simultaneously. Similarly, event VP occurs when action VP! is enabled in the AVI automaton and action VP? is enabled in other automata.

The Vbeat? action has a special role. We may notice that no automaton in the pacemaker model has a Vbeat! or an Abeat! output action. These output actions are assumed to be executed by an external system, namely the heart model. The VRP automaton, besides modeling the ventricular events affected by constraints on the refractory period, interfaces the pacemaker model to the heart model by immediately executing the VS! action upon the occurrence of the Vbeat event. The location symbol adorned with a C is a *committed* location, meant to model an immediate and atomic response. The Abeat event is handled similarly by the PVARP automaton (see Fig. 2).

The following PVS fragment shows part of the PVS model for the VRP automaton.

```
VRP: THEORY BEGIN
IMPORTING constants

Mode: TYPE = { Idle, inter, VRP }
state: TYPE = [#
    time: real,
    loc: Mode
#]

init_VRP: state = (# time := 0, loc := Idle #)

en_Vbeatin(st: state): boolean = loc(st) = Idle
en_VSout(st: state): boolean = loc(st) = inter

Vbeatin(st: (en_Vbeatin)): state =
    st WITH [ loc := inter ]
VSout(st: (en_VSout)): state =
    st WITH [ time := 0, loc := VRP ]
...
END VRP
```

The network behavior is modeled by a coordinating theory, *pacemaker*. In the considered pacemaker model, each output action occurs in only one TA, called the *originating* automaton in the following. The theory is structured as follows:

- The network state is defined by a record type composed of the states of all TA (grouped in the *dev* field) plus four fields representing signals exchanged with the heart model.
- For each output action *a*!, an *event enabling* predicate checks

the enabling conditions for the synchronized occurrence of *a*! and its complementary actions. These conditions, in turn, depend on the enabling conditions for the actions in the single TA.

- For each output action *a*!, an *event transition* function returns the next network state, resulting from the next states computed by the transition functions in the single TA.
- For each automaton *A*, a *network time-checking* predicate checks the enabling conditions for the internal actions of the originating automaton.
- For each automaton *A*, a *network timing* function returns the next network state, resulting from the next state computed by the timing function in the originating automaton.
- An *advance-time* function increases the values of all clocks by one time unit.
- A *network transition* function computes the next value of the *dev* field of the network state by computing the next states of the TA.
- An *interface function* provides an external interface to the network. Its first parameter encodes the Abeat! and Vbeat! actions. With this information the function can compute the successor of a network state *st*. The interface function drives the discrete-time simulation, synchronizing the ground evaluator on the pacemaker side with the fixed-step solver on the heart side.

The following fragment shows the definition of the network state record, *State*:

```
pacemaker: THEORY
BEGIN
IMPORTING LRI, AVI, PVARP, URI, VRP

devState: TYPE = [#
    avi: AVI.state,
    lri: LRI.state,
    pvarp: PVARP.state,
    uri: URI.state,
    vrp: VRP.state
#]

State: TYPE = [#
    dev: devState,
    %-- pacemaker inputs
    Abeat: nat,
    Vbeat: nat,
    %-- pacemaker outputs
    AP: nat,
    VP: nat
#]
```

The event enabling and event transition functions take on different forms depending on the type of event: An event can be an *input boundary* event, an *output boundary* event, or a *non-boundary* event. An input boundary event models the reception of a signal from an external system and is characterized by an input action, such as Abeat? and Vbeat? in our example, without its complementary action. An output boundary event models the issue of a signal towards an external system and its output action, such as AP! and VP!, may have its complementary action or not. A non-boundary event models an exchange of signals within the network with matched pairs of complementary actions.

Let us consider an example of non-boundary event first:

```
en_VSevent(st: State): bool =
    en_VSout(vrp(dev(st))) AND
    (
    en_VSin(lri(dev(st))) OR
    en_VSin(avi(dev(st))) OR
    en_VSin(pvarp(dev(st))) OR
    en_VSin(uri(dev(st)))
    )

VSevent(st: (en_VSevent)): State =
    st WITH [
        dev := dev(st) WITH [
             vrp := VSout(vrp(dev(st))),
             lri := IF (en_VSin(lri(dev(st))))
                    THEN VSin(lri(dev(st)))
                    ELSE lri(dev(st))
                    ENDIF,
             pvarp := IF (en_VSin(pvarp(dev(st))))
                    THEN VSin(pvarp(dev(st)))
                    ELSE pvarp(dev(st))
                    ENDIF,
             uri := IF (en_VSin(uri(dev(st))))
                    THEN VSin(uri(dev(st)))
                    ELSE uri(dev(st))
                    ENDIF,
             avi := IF (en_VSin(avi(dev(st))))
                    THEN VSin(avi(dev(st)))
                    ELSE avi(dev(st))
                    ENDIF
        ]
     ]
```

The event enabling predicate checks if the VS! action is enabled in the VRP automaton and the VS? action is enabled in the other automata.

The event transition function returns the new network state resulting from the composition of the new states of the automata involved in the event.

An example of input boundary event is the Vbeat event:

```
en_Vbeatevent(st: State): bool =
    Vbeat(st) = 1 AND
    en_Vbeatin(vrp(dev(st))) AND
    (
        en_VSin(lri(dev(st))) OR
        en_VSin(avi(dev(st))) OR
        en_VSin(pvarp(dev(st))) OR
        en_VSin(uri(dev(st)))
    )

Vbeatevent(st: State): State =
    st WITH [
        dev := dev(st) WITH [
            vrp := VSout(Vbeatin(vrp(dev(st)))),
            lri := IF (en_VSin(lri(dev(st))))
                       THEN VSin(lri(dev(st)))
                       ELSE lri(dev(st))
                       ENDIF,
            avi := IF (en_VSin(avi(dev(st))))
                       THEN VSin(avi(dev(st)))
                       ELSE avi(dev(st))
                       ENDIF,
            pvarp := IF (en_VSin(pvarp(dev(st))))
                       THEN VSin(pvarp(dev(st)))
                       ELSE pvarp(dev(st))
                       ENDIF,
            uri := IF (en_VSin(uri(dev(st))))
                       THEN VSin(uri(dev(st)))
                       ELSE uri(dev(st))
                       ENDIF
            ]
        ]
```

The event enabling predicate checks if (i) the Vbeat field in the state record is asserted, representing a Vbeat! output action executed by the heart model; (ii) the Vbeat? action is enabled in the VRP automaton; and (iii) the VS? action is enabled in the other automata. As shown in Fig. 2, the Vbeat? action triggers a transition to a committed state, from where a mandatory transition executes the VS! action.

The transition function for output boundary events is more complex because the heart model requires the AP! and VP! signals to be rectangular pulses. In order to simulate these pulses, fields to control their duration have been added to the network state record. The following fragment shows the enabling predicate and transition function for the AP event:

```
en_APevent(st: State): bool =
    en_APout(lri(dev(st))) AND en_APin(avi(dev(st)))

APevent(st: (en_APevent)): State =
    LET
        st =
            st WITH [
                dev := dev(st) WITH [
                    lri := APout(lri(dev(st))),
                    avi := APin(avi(dev(st)))
                ]
            ],
        st =
            st WITH [
                apon :=
                    IF (aptime(st) < APWIDTH)
                    THEN true ELSE false
                    ENDIF
            ]
    IN st
```

The event transition function computes the new LRI and AVI states and the new value of the *apon* variable. This variable is true if and only if the AP signal has been on for a time *aptime* less than the pulse duration *APWIDTH*. The *apon* variable is used in the network transition function to set the value of AP and in the time-advance function to increase or reset the value of *aptime*.

The network time-check predicate and timing function of each automaton use the automaton's time-check predicate and timing function.

The time-advance function simulates the passing of time, by increasing the values of all clocks, including those that control the generation of rectangular pulses:

```
advance_time(st: State): State =
    st WITH [
        dev :=
            dev(st) WITH [
            lri := lri(dev(st))
                WITH [time := time(lri(dev(st))) + 1],
            avi := avi(dev(st))
                WITH [time := time(avi(dev(st))) + 1],
            pvarp := pvarp(dev(st))
                WITH [time := time(pvarp(dev(st))) + 1],
            vrp := vrp(dev(st))
                WITH [time := time(vrp(dev(st))) + 1],
            uri := uri(dev(st))
                WITH [clk := clk(uri(dev(st))) + 1]
            ],
         aptime :=
            IF (apon(st) AND aptime(st) < APWIDTH)
            THEN aptime(st) + 1 ELSE 0
            ENDIF,
         apon := apon(st) AND aptime(st) < APWIDTH,
         vptime :=
            IF (vpon(st) AND vptime(st) < VPWIDTH)
            THEN vptime(st) + 1 ELSE 0
            ENDIF,
         vpon := vpon(st) AND vptime(st) < VPWIDTH
        ]
```

7.4 Simulation of Timed Automata

The network transition function *exec* returns the next network state by computing the output variables, handling shared clocks, and evaluating the transition function for an enabled event. Since the pacemaker model is deterministic, at most one event may be enabled.

```
exec(st: State): State =
  LET
    st =
        st WITH [
            AP := IF (apon(st)) THEN 1 ELSE 0 ENDIF,
            VP := IF (vpon(st)) THEN 1 ELSE 0 ENDIF
        ],
    st =
        st WITH [
            dev := dev(st) WITH [
                avi := avi(dev(st))
                    WITH [clk := clk(uri(dev(st)))]
            ]
        ]
  IN
      IF en_Abeatevent(st) THEN Abeatevent(st)
      ELSIF en_Vbeatevent(st) THEN Vbeatevent(st)
      ELSIF en_APevent(st) THEN APevent(st)
      ELSIF en_VPevent(st) THEN VPevent(st)
      ELSIF en_LRItau(st) THEN LRItau(st)
      ELSIF en_AVItau(st) THEN AVItau(st)
      ELSIF en_PVARPtau(st) THEN PVARPtau(st)
      ELSIF en_VRPtau(st) THEN VRPtau(st)
      ELSE st
      ENDIF
```

The interface function, *pacemaker_tick*, provides the external interface to the network. Its first parameter, *ia*, encodes the external Abeat! and Vbeat! actions as two real numbers, different from zero if the corresponding action occurs. With this information the function computes the successor of the network state *st*, and in particular it checks if the AP and VP events (atrial and ventricular pacing) occur.

```
InputActions: TYPE = [#
    Abeat: nat,
    Vbeat: nat
#]

pacemaker_tick(ia: InputActions)(st: State): State =
    LET
        st = st
            WITH [ Abeat := ia`Abeat,
                   Vbeat := ia`Vbeat ],
        st = exec(st),
        st = advance_time(st)
    IN st
```

At the PVSio interactive prompt, a user could type an application of *pacemaker_tick* providing ground arguments, and the ground evaluator would print out the resulting next state, as shown in the following example, where *init_input* and *init* are constants defined in the *pacemaker* theory.

```
<PVSio> pacemaker_tick(init_input)(init);
```

```
==>
(# Abeat := 0,
   AP := 0,
   device :=
      (# avi := (# clk := 0, loc := Idle, time := 1 #),
         lri := (# loc := LRI, time := 1 #),
         pvarp := (# loc := Idle, time := 1 #),
         uri := (# clk := 1 #),
         vrp := (# loc := Idle, time := 1 #) #),
   Vbeat := 0,
   VP := 0 #)
```

Section 8 explains how the pacemaker model interacts with the heart model.

8 PVS-Simulink Model Co-Simulation

Connecting two or more models based on different formalisms and implemented on different development tools poses three main issues: (i) matching the semantics of the data used in the models; (ii) matching the syntax of the same data; and (iii) synchronizing the simulation of the models. The rest of this section reports how these issues have been dealt with in the specific case discussed in this paper.

8.1 Matching the Semantics

In principle, the semantics of the two models are quite different, one being based on a network of timed automata and one on a large and complex hybrid automaton. However, in this case the interface between the two models is very simple: the heart model produces a *one* value on its Abeat or Vbeat signals when an atrial or ventricular contraction is simulated, and the pacemaker model produces a short rectangular waveform on AP or VP when a pulse must be delivered to either chamber. Therefore, the problem of matching the two data semantics reduces to a simple exchange of events, encoded as single values for the Abeat and Vbeat signals, and sequences of constant values for the AP and VP signals.

Whilst in our specific case PVS and Simulink exchange binary values that can be easily encoded in text and converted between the two tools, it is important to discuss the validity of this solution in the general case, where integer or real numbers need to be exchanged between the tools. With this respect, it is important to notice that communication between PVS and Simulink is carried out only during co-simulation runs. Therefore, the PVS component involved in the communication is PVSio, and *not* the PVS theorem prover.

In PVSio, an implementation of mathematical integers and reals is used. By default, numerical reals are evaluated in double precision (when semantic attachments are used) and then transformed into exact rationals and printed in the form n/d, where n is the numerator, and d is the denominator of the rational. PVSio uses a Lisp execution environment, and in Lisp there are no limits to the precision of rational numbers (other than computer memory). If a guaranteed decimal approximation is needed, a NASA library (`fast_approx`) can be used by the developers. The library provides a function `set_precision` for setting the desired decimal precision, and a function `rat2decstr` for printing real values in decimal format and according to the set decimal precision.

In Simulink, the precision and range of values exchanged with PVSio is defined at compile time, based on the type definitions and the compilation parameters used in the S-Function block responsible for managing communication of events and data with PVSio.

To ensure correct treatment and encoding of values, therefore, developers need to correctly set up the two simulation environments and the models, so as to make sure that the arithmetic precision used for integers and reals in the PVS model is congruent to the range of values exchanged with the Simulink model.

The other important aspect of model semantics, i.e., time, is discussed in the section on synchronization (Sec. 8.3).

8.2 Matching the Syntax

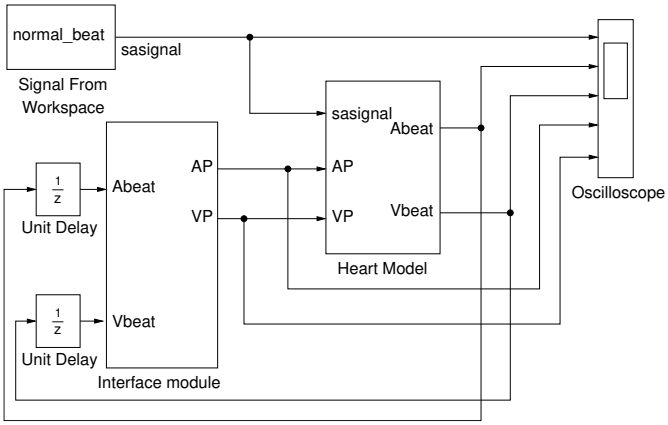

Fig. 3. Simulink diagram of the extended heart model.

The syntax matching issue arises from the different user interaction styles of the applications implementing the two models. PVSio has a command-line interface, where the user types in a function expression and the PVSio solver prints out the result. Simulink computes the evolution of the system's variables, samples their values at user-specified simulated intervals, then displays the sampled values of selected variables graphically on simulated oscilloscope screens or numerically on simulated displays, or stores them in a file. It is then necessary, at each communication event, to extract information from the originating model's user interface, serialize it into a form suitable for transmission, and convert it in to a form acceptable for the destination model's interface.

Since the inputs and outputs of the pacemaker model are text strings formatted according to the PVS syntax, it is straightforward to encode and decode the data from or towards the other model. The only problem is that PVSio expects inputs from the operating system's standard input (normally the keyboard) and writes outputs to the standard output (normally an Emacs editor window or a command window). The operating system's facilities for input and output redirection must then be used to connect the PVSio process to the other model.

The heart model, in Simulink, has a graphical interface, and during simulation it does not accept user interaction, aside from pausing and restarting simulation, and possibly changing model parameters during pauses. In order to let the model communicate with other processes, a custom interface block has been added to the heart model (see Fig. 3), using a Simulink S-Function block. This block extracts the AP and VP signals from text data received from the pacemaker model and feeds them to the heart block. It also reads the Abeat and Vbeat signals from the heart block, converts them into text form, and sends them to the

pacemaker model. At each simulation step, the heart block reads the current value of the sinoatrial signal from a sequence of values (*normal_beat* in the figure) stored in the Matlab workspace, and the AP and VP values from the interface block.

8.3 Synchronization

In the heart model, the time increment at each simulation step (which is actually an ODE integration step) depends on the ODE solver's algorithm. Simulink provides both variable- and fixed-step solvers. The latter are used for the heart model, with a solver step of 4 ms. This step duration is the common time unit used as reference in the co-simulation.

In the pacemaker model, the progress of time is modeled by incrementing the clock variables by one time unit at each simulation step, i.e., each time the interface function *pacemaker_tick* is evaluated.

In order to synchronize the two models, the simulated time of the heart model is taken as the global reference. At each sampling time, the interface block is executed and its S-function sends the pacemaker model a request to evaluate the interface function, so that the two models execute in lockstep with a blocking communications pattern.

It is important to remark that the simulation speed depends on the processing capabilities of the machine running the simulation. Therefore, 1 ms in the simulated execution run can last longer than 1 ms in the real world. This is however not an issue in our case, as the PVSio and Simulink execution environments do not have real-time constraints, and they correctly handle simulation runs where simulated time deviates from real-world wall-clock time.

8.4 Co-Simulation in the PVSio-web Framework

The main components of the co-simulation environment are:

1) The PVSio-web platform.
2) The PVSio solver.
3) The co-simulation engine.
4) The heart model interface block.
5) The heart model.

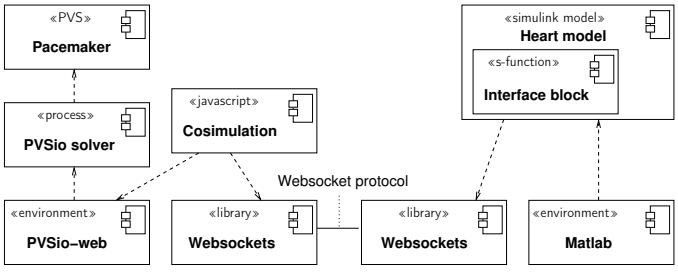

Fig. 4. Components of the co-simulation architecture.

PVSio-web offers two co-simulation engines: an internal engine based on a WebSocket protocol and an external engine [73] based on the communication and co-ordination middleware SAPERE [74]. In the present work, the internal co-simulation engine is used, as it is sufficient to support co-simulation of the considered pacemaker-heart example. In this section, the details of the internal engine and of the heart model interface block are presented.

The internal co-simulation engine uses the APIs provided by PVSio-web to create a web server that encapsulates an instance of PVSio animating the PVS model. The component is implemented in JavaScript, and includes three main functions:

- *Connect*: this function creates a WebSocket connection channel with the co-simulation component managing Simulink simulation runs. The function is invoked at the beginning of the co-simulation run.
- *OnMessageReceived*: this function listens for simulation events and data sent by the heart model interface block over a WebSocket connection on a pre-defined port. Events and data are encoded using a tool-neutral JSON format. The function translates simulation events and data into PVS expressions that can be evaluated in PVSio. The format of the PVS expressions is based on the structure of the PVS record type used to encode the state of the network of timed automata.
- *SendMessage*: this function listens for evaluation results returned by PVSio, and translates these results into JSON objects that can be sent to the Simulink heart model's interface block.

The heart model interface block is implemented as a Simulink System Function (S-Function) block. The block implements a communication bridge for exchanging commands and data with the pacemaker model using JSON objects. A standard communication library, libwebsockets[1], is used as a basis for the implementation. The block has two output buses for injecting simulation events (AP, VP) received from the pacemaker into the heart model. Similarly, two input buses are used for intercepting relevant state variables of the heart model (Abeat and Vbeat) that need to be transmitted to the pacemaker.

Fig. 4 shows the components of the co-simulation architectures with their dependencies, using the UML component notation, with stereotypes to show different kinds of components, such as, e.g., PVS or JavaScript code.

The Emucharts editor of the PVSio-web platform is used to represent graphically the single TA. Fig. 5 shows the PVSio-web interface, where the Emucharts editor canvas is in the lower part. The *PVS Theory* item in the *Code Generators* menu produces the corresponding PVS theory. The LRI automaton in Fig. 5 is reproduced for clarity in Fig. 6. Notably, the latest release of the Emucharts editor can import UPPAAL models.

The coordinating theory *pacemaker* is currently not generated automatically by PVSio-web, and developers need to write the theory directly in the PVS language using the patterns defined in Section 7.3.

8.5 Interaction Details

When the PVSio-web platform is started, it creates a WebSocket server and opens a web browser on the PVSio-web start page. The user types the URL of the directory containing the pacemaker model files, and the co-simulation HTML page opens. This page loads the co-simulation engine, a JavaScript file that sends the WebSocket server a command to start the PVSio solver, loading the pacemaker model theory. The solver is instantiated through another script of the PVSio-web platform, which redirects the solver's input and output to the web sockets server. The user can then start the heart simulation in the Simulink environment. The simulation engine executes the interface block for the first time, the block's S-function establishes a connection with the local WebSocket server, and the port for that connection is shown in a

1. https://libwebsockets.org/

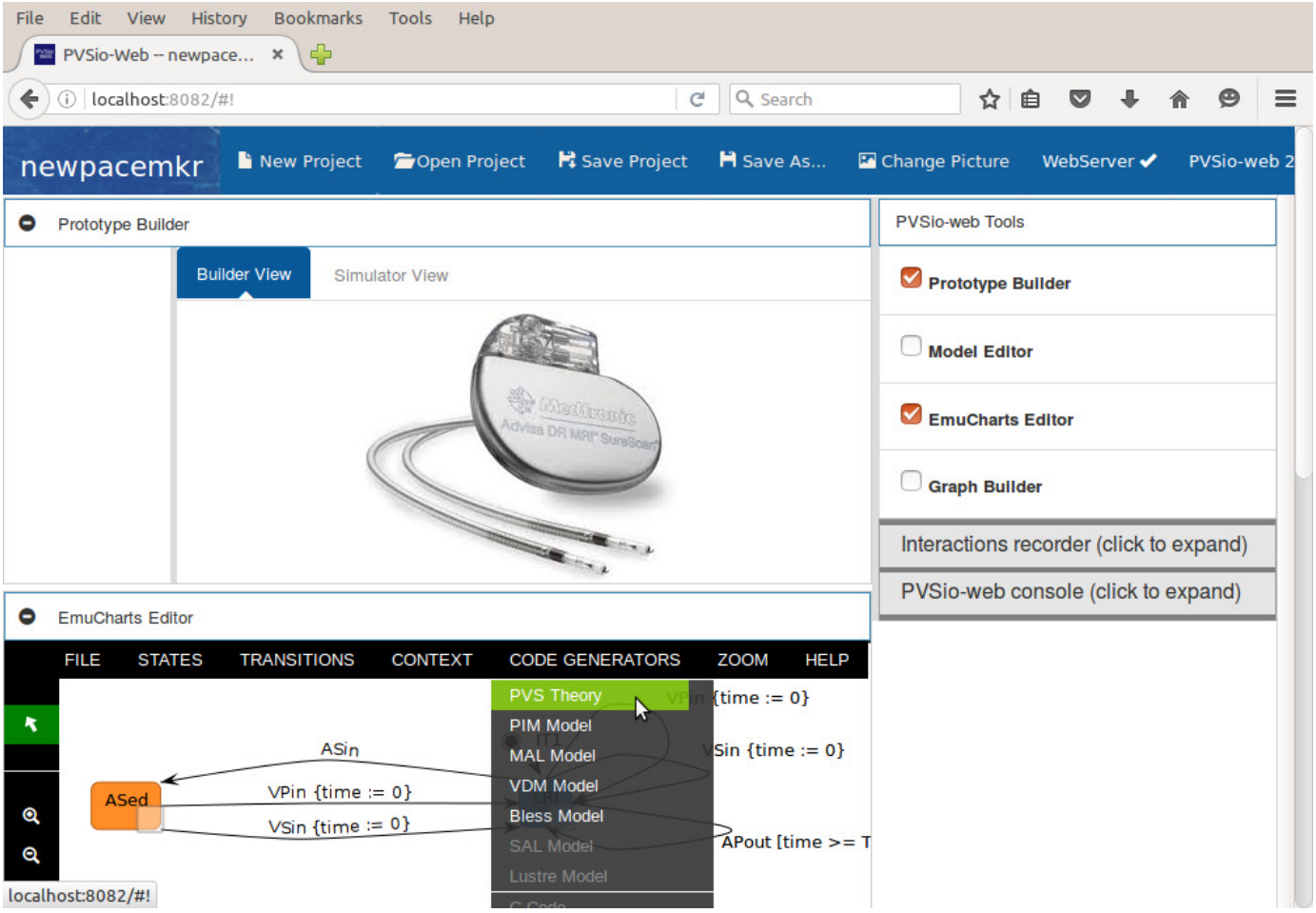

Fig. 5. User interface of the PVSio-web environment. This web page shows the Prototype Builder interface in the top left frame and the Emucharts Editor in the bottom left frame. In the latter, the user has selected the PVS generator.

numerical display block of the heart model's graphical interface. The user can now turn to the co-simulation web page (see Fig. 7), insert the hostname of the machine where Simulink is running, and click on a "connect" button, thus establishing communication between the two models and the co-simulation engine. Two display windows show debugging information.

Fig. 8 shows the exchange of messages after the connections have been established. In the figure, the PVSio-web lifeline represents the PVSio-web platform including the PVSio solver, and the Heart model represents the Simulink model including the heart model block and the interface block.

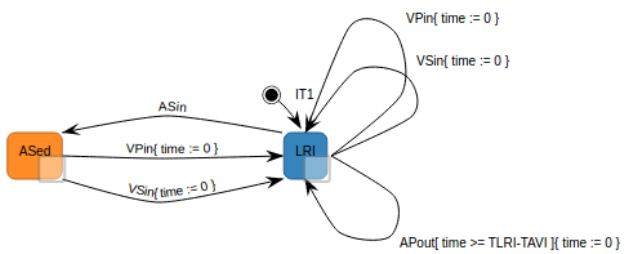

Fig. 6. Emucharts diagram of the LRI automaton.

More in detail, the co-simulation engine, upon being loaded in the web browser, creates a WebSocket and registers a few listener functions that will be called on the occurrence of such events as connection opening or closing, and connects to PVSio-web. The listener for the *connection opened* event sends messages to PVSio-web causing it to start the PVSio solver, then it associates the *connect* button on the web page with a function that establishes a connection to Simulink and starts the process orchestrating the exchange of messages. First, a timer is programmed to send Simulink a *sense* command at regular intervals. When Simulink replies, the values of Abeat and Vbeat are inserted in a string representing a *pacemaker_tick* invocation, together with the current pacemaker state. The string is sent to PVSio-web and the co-simulation engine waits for the control signals AP and VP from the PVSio solver, that are then sent to Simulink.

On the Simulink side, the interface block is implemented as an S-function with the values of Abeat and Vbeat as inputs and AP and VP as output arguments. The simulation engine executes the interface block every sampling time. The first time the interface block is executed, its S-function tries to open a WebSocket on a predefined port number. If the attempt fails, other ports are tried. The selected port number is then shown on a display block (not shown in Fig. 3). At each subsequent invocation, the S-function waits for an incoming message. If the message is a *sense* command, the values of Abeat and Vbeat are converted to strings and inserted in the response, which is sent to the co-simulation

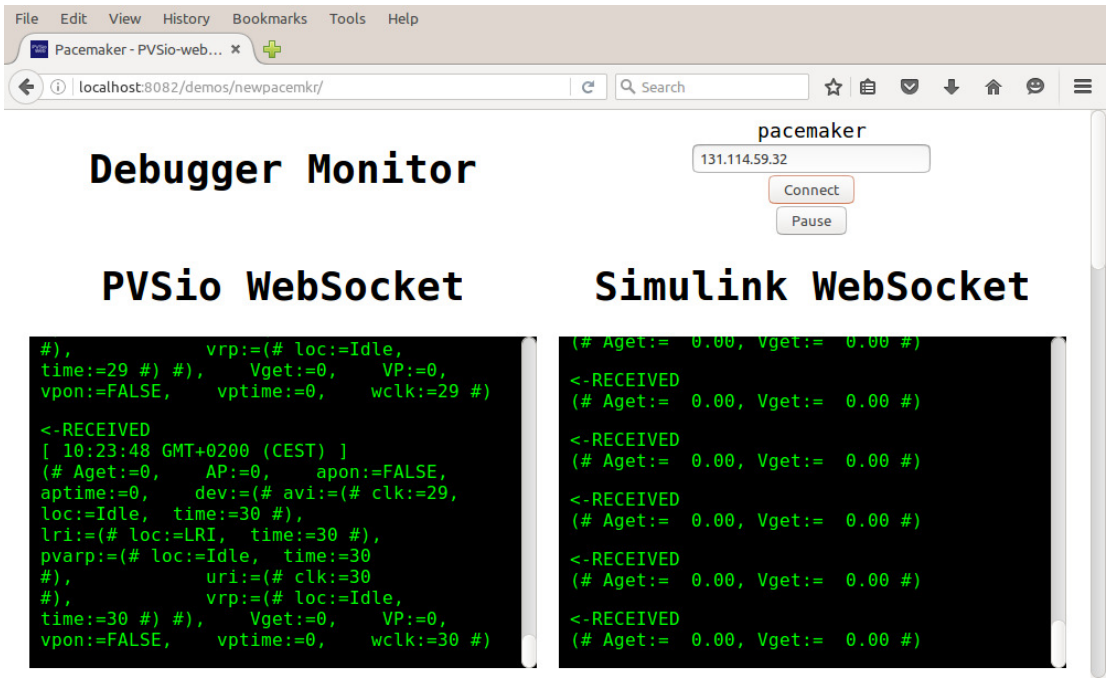

Fig. 7. Co-simulation web page. The *pacemaker* field shows the IP number of the remote machine hosting the Simulink environment. The Connect button connects the co-simulation module to the Heart Model, and the other two fields trace the exchanged messages.

engine. If the message is a pacing command, the values of AP and VP are extracted from the command and returned to the simulation engine. The simulation proceeds then to the next sampling time.

The co-simulation engine sends another *sense* command to the heart model, and the cycle repeats until the heart model simulation terminates.

8.6 Validation of the Pacemaker-Heart Model

The co-simulation structure described above can be used to validate the pacemaker specification under different pathological conditions. As mentioned in Sec. 5.1, the signal from the SA node determines the heartbeat rate. In the model shown in Fig. 3, the SA signal is generated by a *signal from workspace* block, representing a sequence of values stored in the Matlab workspace. Different patient conditions are simulated by changing the SA signal.

Fig. 9 shows the results of simulating a bradycardic episode, lasting about sixteen seconds. The four traces show, from top to bottom, a bradycardic SA signal, the resulting bradycardic Vbeat signal in absence of external pacing, the VP signal from the pacemaker, and the resulting paced Vbeat signal from the heart. It may be observed how the pacemaker correctly issues pacing signals at the rate of approximately one beat per second, doubling the rate of the Vbeat signal.

9 Verification

Whereas this work is focused on simulation, it should be remembered that the PVS is first and foremost an interactive environment to prove logical statements by manipulating them with commands implementing inference rules. For example, the following statement can be proved with the single PVS command *grind*, which iteratively applies instantiations and simplifications:

```
lri_ap: LEMMA
  FORALL (s0, s1: State):
    en_APout(lri(s0)) AND s1 = APout(s0)
    IMPLIES
    loc(lri(s1)) = LRI AND time(lri(s1)) = 0
```

The above lemma means: "*It is always the case that module lri is in mode* LRI *and its clock is reset when transition AP is executed.*" Lemmas like this allow developers to perform essential sanity checks for the model, and verify that the model definition correctly incorporates hypotheses about the behavior of the system. In our case, an attempt to prove this lemma on an early version of the model failed, leading us to the discovery of an error in another part of the specification.

More interesting properties require more complex proofs. To prove, for example, that the pacemaker network is deterministic, one may prove that all events are mutually exclusive, i.e, when one event is enabled, all others are disabled. The following is a fragment of a theory for the verification of the pacemaker network:

```
pacemaker_verif: THEORY
BEGIN
IMPORTING pacemaker

pvarp1: AXIOM
    forall (st: State):
        loc(pvarp(dev(st))) = inter => Abeat(st) = 1

vrp1: AXIOM
    forall (st: State):
        loc(vrp(dev(st))) = inter => Vbeat(st) = 1

vrp2: AXIOM
    forall (st: State):
        not en_Vbeatevent(st)
        => not en_VSout(vrp(dev(st)))

ap_en_pvarptau: LEMMA
    forall (st: State):
        en_APevent(st)
```

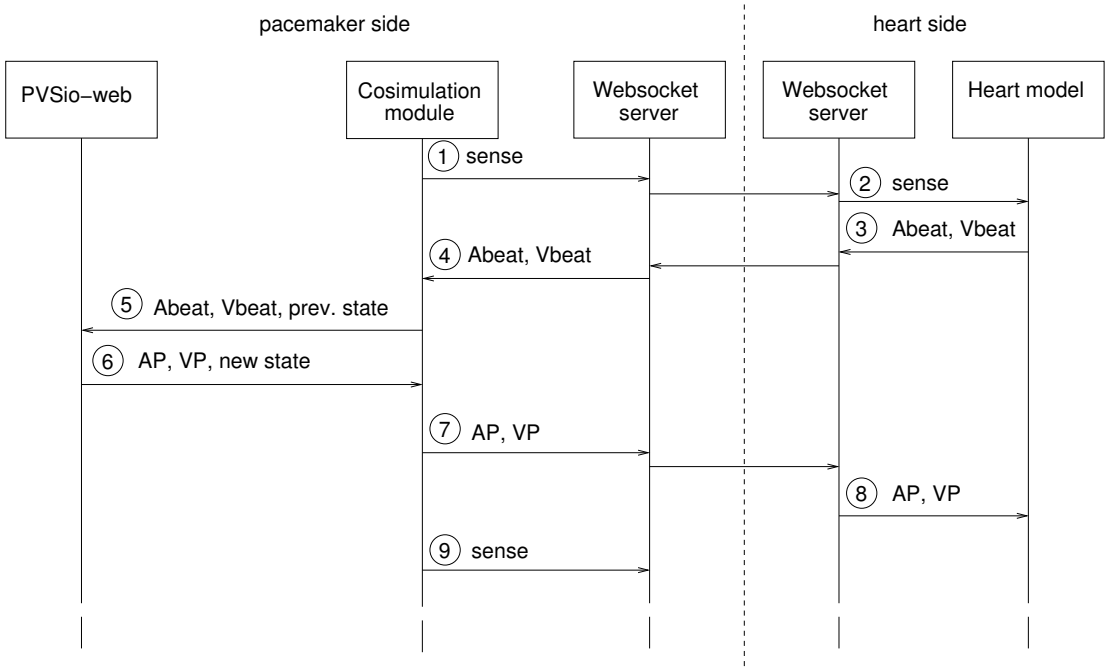

Fig. 8. Message flow during co-simulation.

```
        IMPLIES
        not en_PVARPtau(st)

...
END pacemaker_verif
```

The axioms shown above express the semantics of committed locations in automata PVARP and VRP. Other axioms, not shown, represent location invariants and other properties immediately deducible from the theory of timed automata. Lemma *ap_en_pvarptau* states that the internal actions of PVARP are not enabled when event AP is enabled. The proof tree of this lemma is shown in Fig. 10. The diagram, generated by the PVS, shows the initial sequent at the root, the commands issued by the user, and the resulting sequents, represented by the entailment symbols. The sequents can be made visible in the PVS theorem prover by clicking on the respective symbols, besides being typed out in the prover interface.

The first command, *skosimp∗*, eliminates the universal quantifier by instantiating the sequent with fresh variables, *expand* replaces function names by their definitions, and *lemma* introduces axioms or already proved lemmas. The *inst* command eliminates a quantifier using ground expressions (constants in this case), *split* and *flatten* restructure a sequent, and *assert* applies various inference rules, which in this case lead to proving the sequents.

A few steps of the proof are shown in the following fragment, where the dashed line represents the entailment symbol separating antecedents (above) and consequents (below). The lemma to be proved is the first sequent, and it has no antecedents:

```
ap_en_pvarptau :

  |-------
{1}   FORALL (st: State): en_APevent(st)
          IMPLIES NOT en_PVARPtau(st)

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
ap_en_pvarptau :

{-1}  en_APevent(st!1)
{-2}  en_PVARPtau(st!1)
  |-------

Rule? (expand "en_PVARPtau")
Expanding the definition of en_PVARPtau,
this simplifies to:
ap_en_pvarptau :

[-1]  en_APevent(st!1)
{-2}  en_tau(pvarp(dev(st!1)))
  |-------

...

Rule? (lemma "invar_pvab")
Applying invar_pvab
this simplifies to:
ap_en_pvarptau.1 :

{-1}  FORALL (st: State):
        loc(pvarp(dev(st))) = PVAB
            => time(pvarp(dev(st))) < TPVAB
[-2]  loc(pvarp(dev(st!1))) = PVAB
[-3]  time(pvarp(dev(st!1))) >= TPVAB
[-4]  en_APevent(st!1)
  |-------

Rule? (inst -1 "st!1")
Instantiating the top quantifier in -1 with the terms:
 st!1,
this simplifies to:
ap_en_pvarptau.1 :

{-1}  loc(pvarp(dev(st!1))) = PVAB
          => time(pvarp(dev(st!1))) < TPVAB
[-2]  loc(pvarp(dev(st!1))) = PVAB
```

```
[-3]  time(pvarp(dev(st!1))) >= TPVAB
[-4]  en_APevent(st!1)
  |-------

...

Rule? (assert)
Simplifying, rewriting, and recording
with decision procedures,

This completes the proof of ap_en_pvarptau.1.1.

...

Q.E.D.
```

After proving similar lemmas, it is immediate to prove that no other event is enabled when AP is enabled:

```
deterministic_AP: LEMMA
    forall (st: State):
        en_APevent(st)
        IMPLIES
        not en_LRItau(st)
        and not en_VPevent(st)
        and not en_AVItau(st)
        and not en_ASevent(st)
        and not en_PVARPtau(st)
        and not en_VSevent(st)
        and not en_VRPtau(st)
        and not en_Abeatevent(st)
        and not en_Vbeatevent(st)
```

It may be observed that a PVS proof includes the automatic verification performed by PVS to check the well-formedness of the specification: type correctness, coverage of conditions, disjointness of conditions. All these checks, called *typecheck conditions* (TCC), are automatically and quickly verified by the prover.

10 Discussion and Conclusion

The construction of a formal model of a medical device and the use of formal verification technologies can help system designers to gain confidence that the device performs the required functions under all the stated conditions, thus enhancing patient safety.

In the present work we have described a framework that facilitates modeling and simulation of cyber-physical systems. Software functionalities are modeled in the PVS theorem proving system. The characteristics of the plant are modeled in Simulink. Simulation of the overall cyber-physical system is obtained through co-simulation of the two models. The integrated simulation facilitates validation of the models, lightweight formal analysis based on simulation and testing, and full formal analysis of the software based on assume-guarantee reasoning [75], [70] techniques.

The capabilities of the framework were demonstrated with an example based on a pacemaker-heart system. A timed automata specification of the pacemaker software is developed in PVS according to identified modeling patterns. The framework enables integrated simulation of the PVS pacemaker model with a Simulink heart model built on medical domain-specific knowledge. Integrated simulation allows software engineers to demonstrate the functionalities of the pacemaker software, and discuss hypotheses about its behavior for different physiological parameters of the patient. On the other hand, the correctness of the pacemaker design can be analyzed using PVS. This includes: well-formedness of the design (coverage of conditions, disjointness of conditions, type correctness), and analysis of design requirements.

An important aspect related to the generality of the framework is how easy is to extend the framework to support modeling and analysis tools other than PVS and Simulink. The question boils down to discussing the generality of the PVSio-web architecture used as a basis to build the co-simulation platform. The backbone of the architecture uses a standard WebSocket protocol for communication of simulation events and data. In fact, the PVSio environment of PVS is wrapped within a web-server responsible for converting the native read-eval-print loop of PVSio into a tool-neutral event-based service. To do this conversion, the PVSio wrapper implements four main functions: *start*, which spawns a process running a PVSio instance; *sendCommand*, which translates simulation events encoded in JSON format into PVS expressions that can be evaluated in PVSio; *processData*, which translates the results returned by PVSio into tool-neutral JSON objects; and *kill*, which terminates the created process. If developers want to connect another tool providing an interactive read-eval-print loop to the PVSio-web infrastructure, they need only to re-implement these four interface functions of the wrapper.

Two important lessons were learned from this work, one related to the utility of co-simulation, and another related to challenges in the implementation of co-simulation engines.

- Co-simulation proved an effective technology for the analysis of extensive models. The heart model considered in our case study was developed and validated by others in [62]. It included over 200 functional blocks. Translating such a large model in PVS or any another formal methods language would have been impossible in a reasonable amount of time with the available resources. Using co-simulation, we were able to use the heart model as-is, to validate the PVS model of the pacemaker and perform lightweight formal verification of system-level properties of the overall pacemaker-heart system.
- The co-simulation engine implemented in our framework uses ad-hoc APIs, and each co-simulated model needs to be instrumented with those APIs. This is not an ideal situation, as developers need to re-instrument the models if a different co-simulation engine is to be used. Moving towards a standard API such as that defined in the Functional Mock-up Interface (FMI) would bring clear benefits under this perspective. We are currently extending the PVSio-web co-simulation engine in this direction.

Being focused on co-simulation, we could dedicate only limited space to demonstrating how to carry out formal verification of safety requirements for the pacemaker model in PVS. Given that the heart model is not developed in PVS but in Simulink, an assume-guarantee reasoning is necessary to perform verification of system-level properties. An example such property is: *"For any bradycardic episode, the pacemaker ensures that cardiac cycles occur at the correct rate."* With assume-guarantee, the verification effort allows developers to prove that the pacemaker model *guarantees* the desired behavior of the pacemaker-heart system under suitable *assumptions* on the heart model. Formalizing these assumptions will be the object of further work.

An interesting research strand that is worth exploring as future work relates to the use of co-simulation and formal verification technologies for the analysis of cyber-threats and security-related properties. The specific case study considered in this paper is particularly relevant to such research strand, as pacemakers can be configured and re-programmed using low-range wireless tech-

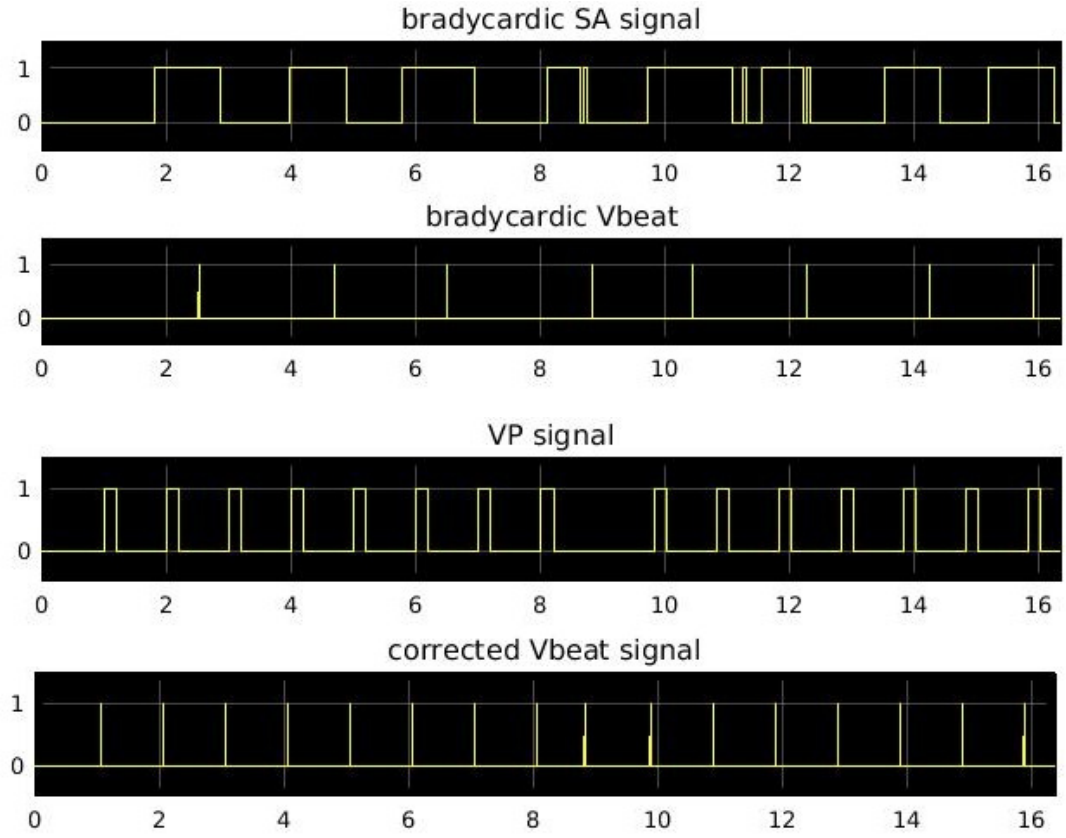

Fig. 9. Simulation output for bradycardia. The bradycardic Vbeat signal shows eight pulses in approximately sixteen seconds. The pacemaker issues VP pulses to stimulate the ventricle to contract sixteen times (corrected Vbeat) in the same timespan.

nologies, and various pacemaker manufacturers have discovered latent vulnerabilities in their pacemakers after the pacemaker was placed in the market. Device recalls have been carried out by manufacturers because these vulnerabilities, if exploited, could lead to permanent impairment or life-threatening injury for the patients (see [76]).

Acknowledgment

We would like to thank Alexandru Mereacre (University of Oxford), who helped us with the Simulink model of the heart, and César Muñoz (NASA Langley Research Center), that provided support for technical aspects of the PVSio animation environment. Piergiuseppe Mallozzi contributed to this paper with his Master's thesis. We also thank the anonymous reviewers for their valuable comments. Paolo Masci is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF) within Project "NORTE-01-0145-FEDER-000016".

References

[1] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 731–736.

[2] V. Carreño and C. Muñoz, "Aircraft trajectory modeling and alerting algorithm verification," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, M. Aagaard and J. Harrison, Eds. Springer Berlin Heidelberg, 2000, vol. 1869, pp. 90–105. [Online]. Available: http://dx.doi.org/10.1007/3-540-44659-1_6

[3] C. Bernardeschi and A. Domenici, "Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System," *Information Processing Letters*, vol. 116, no. 6, pp. 409–415, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019016300072

[4] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, "PVS: combining specification, proof checking, and model checking," in *Computer-Aided Verification, CAV '96*, ser. LNCS, R. Alur and T. Henzinger, Eds. Springer-Verlag, 1996, no. 1102, pp. 411–414.

[5] C. Muñoz, "Rapid prototyping in PVS," National Institute of Aerospace, Hampton, VA, USA, Tech. Rep. NIA 2003-03, NASA/CR-2003-212418, 2003.

[6] "Simulink® web site." [Online]. Available: http://www.mathworks.com/products/simulink

[7] "Scicoslab web site." [Online]. Available: http://www.scicoslab.org

[8] G. Hamon and J. Rushby, "An operational semantics for Stateflow," in *Fundamental Approaches to Software Engineering (FASE)*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, vol. 2984, pp. 229–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24721-0_17

[9] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: State of the art," *ACM Computing Surveys*, 2017, to appear.

[10] C. Muñoz, A. Narkawicz, G. Hagen, J. Upchurch, A. Dutle, and M. Consiglio, "DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems," in *Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015)*, Prague, Czech Republic, September 2015.

[11] *Stateflow Reference*, The MathWorks, Inc. [Online]. Available: https://www.mathworks.com/help/stateflow

[12] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf, "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," in *Proc. of the 8th International Modelica Conference*. Linköping University Electronic Press, 2011, pp. 105–114. [Online]. Available: http://dx.doi.org/10.3384/ecp11063105

[13] T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and

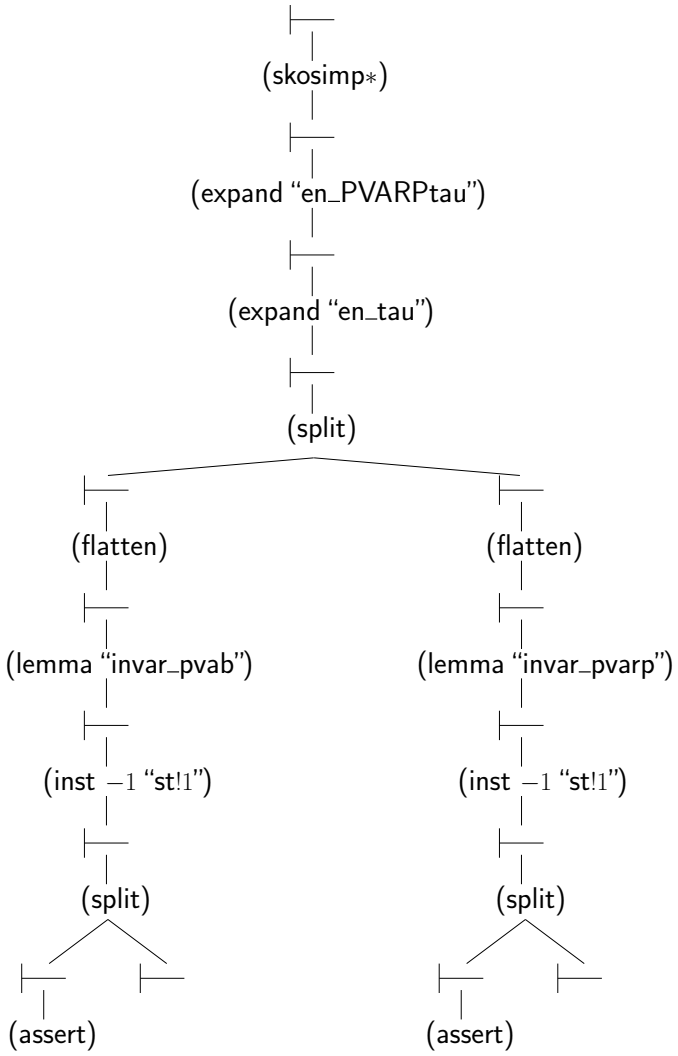

Fig. 10. Proof tree of Lemma *ap_en_pvarptau*. The $\vdash$ symbols represent (sub)goals, each followed by a PVS rule.

A. Viel, “Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models,” in *Proceedings of the 9th International Modelica Conference*. The Modelica Association, 2012, pp. 173–184. [Online]. Available: http://dx.doi.org/10.3384/ecp12076173

[14] B. Wang and J. S. Baras, “Hybridsim: A modeling and co-simulation toolchain for cyber-physical systems,” in *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE/ACM 17th International Symposium on*, Oct 2013, pp. 33–40.

[15] L. Jalali, S. Mehrotra, and N. Venkatasubramanian, “Simulation integration: Using multidatabase systems concepts,” *Simulation*, vol. 90, no. 11, pp. 1268–1289, Nov. 2014. [Online]. Available: http://dx.doi.org/10.1177/0037549714553151

[16] J. Fitzgerald, P. G. Larsen, K. Pierce, M. Verhoef, and S. Wolff, *Integrated Formal Methods: 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ch. Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems, pp. 12–26. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16265-7_2

[17] D. Bjørner and C. B. Jones, Eds., *The Vienna Development Method: The Meta-Language*. London, UK: Springer-Verlag, 1978.

[18] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef, *Vienna Development Method*. John Wiley & Sons, Inc., 2007. [Online]. Available: http://dx.doi.org/10.1002/9780470050118.ecse447

[19] D. Karnopp and R. Rosenberg, *Analysis and simulation of multiport systems; the bond graph approach to physical system dynamics*. Cambridge, MA, USA: M.I.T. Press, 1968.

[20] P. Larsen, C. Gamble, K. Pierce, A. Ribeiro, and K. Lausdahl, *Support for Co-modelling and Co-simulation: The Crescendo Tool*. Springer, 2014, pp. 97–114.

[21] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, “The Overture Initiative Integrating Tools for VDM,” *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 1, pp. 1–6, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1668862.1668864

[22] J. F. Broenink, “Modelling, simulation and analysis with 20-sim,” *Journal A*, vol. 38, no. 3, pp. 22 –25, Sep. 1997.

[23] S. H. A. Niaki and I. Sander, “Co-simulation of embedded systems in a heterogeneous MoC-based modeling framework,” in *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, June 2011, pp. 238–247.

[24] I. Sander and A. Jantsch, “System modeling and transformational design refinement in ForSyDe,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, Jan 2004.

[25] E. A. Lee and A. Sangiovanni-Vincentelli, “A framework for comparing models of computation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, Dec 1998.

[26] “Simulink Design Verifier® web site.” [Online]. Available: http://www.mathworks.com/products/designverifier

[27] M. Sheeran and G. Stålmarck, *Formal Methods in Computer-Aided Design: Second International Conference, FMCAD’ 98 Palo Alto, CA, USA, November 4–6, 1998 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, ch. A Tutorial on Stålmarck’s Proof Procedure for Propositional Logic, pp. 82–99. [Online]. Available: http://dx.doi.org/10.1007/3-540-49519-3_7

[28] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, “Translating Discrete-time Simulink to Lustre,” *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 779–818, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1113830.1113834

[29] “SCADE suite® web site.” [Online]. Available: http://www.esterel-technologies.com/products/scade-suite

[30] R. Reicherdt and S. Glesner, *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*. Cham: Springer International Publishing, 2014, ch. Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie, pp. 190–204. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10431-7_14

[31] K. R. M. Leino, “This is Boogie 2,” Microsoft Research, Tech. Rep. MSR-TR-2008-194, June 2008. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=147643

[32] L. De Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’08/ETAPS’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[33] P. Boström, *Formal Methods and Software Engineering: 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. Contract-Based Verification of Simulink Models, pp. 291–306. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24559-6_21

[34] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.

[35] R.-J. J. Back, A. Akademi, and J. V. Wright, *Refinement Calculus: A Systematic Introduction*, 1st ed., F. B. Schneider and D. Gries, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1998.

[36] B. Silva, K. Richeson, B. Krogh, and A. Chutinan, “Modeling and verifying hybrid dynamic systems using CheckMate,” in *Proc. Conf. on Automation of Mixed Processes: Hybrid Dynamic Systems*, 2000, pp. 323–328.

[37] R. De Nicola and F. Vaandrager, *Semantics of Systems of Concurrent Processes: LITP Spring School on Theoretical Computer Science La Roche Posay, France, April 23–27, 1990 Proceedings*. Berlin, Heidelberg: Springer, 1990, ch. Action versus state based logics for transition systems, pp. 407–419. [Online]. Available: http://dx.doi.org/10.1007/3-540-53479-2_17

[38] C. Chen, J. S. Dong, and J. Sun, *Formal Methods and Software Engineering: 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, FL, USA, November 14-15, 2007. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ch. Machine-Assisted Proof Support for Validation Beyond Simulink, pp. 96–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76650-6_7

[39] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. Wabenhorst, "A set-theoretic model for real-time specification and reasoning," in *Proceedings of the Mathematics of Program Construction*, ser. MPC '98. London, UK, UK: Springer-Verlag, 1998, pp. 188–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=648084.747169

[40] M. Archer, "TAME: Using PVS strategies for special-purpose theorem proving," *Annals of Mathematics and Artificial Intelligence*, vol. 29, no. 1, pp. 139–181, 2000. [Online]. Available: http://dx.doi.org/10.1023/A:1018913028597

[41] Z. Jiang, M. Pajic, A. Connolly, S. Dixit, and R. Mangharam, "Real-time heart model for implantable cardiac device validation and verification," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*. IEEE, 2010, pp. 239–248.

[42] Z. Jiang, M. Pajic, and R. Mangharam, "Cyber–physical modeling of implantable cardiac medical devices," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 122–137, 2012.

[43] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Third International Conference on Quantitative Evaluation of Systems (QEST 2006)*, Sept 2006, pp. 125–126.

[44] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, "Quantitative verification of implantable cardiac pacemakers," in *33rd IEEE Real-Time Systems Symposium (RTSS), 2012*, Dec 2012, pp. 263–272.

[45] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Computer aided verification*. Springer, 2011, pp. 585–591.

[46] A. Platzer and J.-D. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems (system description)," in *International Joint Conference on Automated Reasoning*. Springer, 2008, pp. 171–178.

[47] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer, "KeYmaera X: An axiomatic tactical theorem prover for hybrid systems," in *International Conference on Automated Deduction*. Springer, 2015, pp. 527–538.

[48] P. Masci, A. Ayoub, P. Curzon, M. D. Harrison, I. Lee, and H. Thimbleby, "Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example," in *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2013, pp. 81–90.

[49] P. Masci, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby, "Formal verification of medical device user interfaces using PVS," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2014, pp. 200–214.

[50] V. Valero, G. Díaz, and M.-E. Cambronero, "Timed automata modeling and verification for publish-subscribe structures using distributed resources," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 76–99, 2017.

[51] L. Shan, S. Graf, and S. Quinton, "RTLib: A Library of Timed Automata for Modeling Real-Time Systems," Ph.D. dissertation, Grenoble 1 UGA-Université Grenoble Alpe; INRIA Grenoble-Rhone-Alpes, 2016.

[52] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proceedings of the Second Workshop on Formal Methods in Software Practice*, ser. FMSP '98. New York, NY, USA: ACM, 1998, pp. 7–15. [Online]. Available: http://doi.acm.org/10.1145/298595.298598

[53] K. Y. Rozier, "Specification: The biggest bottleneck in formal methods and autonomy," in *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*, 2016, pp. 8–26. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-48869-1_2

[54] K. C. Castillos, F. Dadeau, J. Julliand, B. Kanso, and S. Taha, *A Compositional Automata-Based Semantics for Property Patterns*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 316–330. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38613-8_22

[55] I. Fette and A. Melnikov, "The WebSocket Protocol," Internet Requests for Comments, RFC Editor, RFC 6455, December 2011, http://www.rfc-editor.org/rfc/rfc6455.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6455.txt

[56] P. Masci, Y. Zhang, P. Jones, P. Oladimeji, E. D'Urso, C. Bernardeschi, P. Curzon, and H. Thimbleby, "Combining PVSio with stateflow," in *Proceedings of the 6th NASA Formal Methods Symposium (NFM2014)*. Berlin, Heidelberg: Springer-Verlag, April-May 2014.

[57] C. Bernardeschi, A. Domenici, and P. Masci, "Integrated simulation of implantable cardiac pacemaker software and heart models," in *CARDIOTECHNIX 2014, 2d International Congress on Cardiovascular Technology*. SCITEPRESS, 2014, pp. 55–59.

[58] P. Oladimeji, P. Masci, P. Curzon, and H. Thimbleby, "PVSio-web: a tool for rapid prototyping device user interfaces in PVS," in *FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems, London, UK, June 24, 2013*, 2013.

[59] P. Masci, P. Oladimeji, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby, "PVSio-web 2.0: Joining PVS to HCI," in *Computer Aided Verification: 27th International Conference, CAV 2015, Proceedings, Part I*, D. Kroening and S. C. Păsăreanu, Eds. Springer International Publishing, 2015, pp. 470–478, tool available at http://www.pvsioweb.org.

[60] C. Fayollas, C. Martinie, P. Palanque, P. Masci, M. D. Harrison, J. C. Campos, and S. R. e Silva, "Evaluation of formal IDEs for human-machine interface design and analysis: the case of CIRCUS and PVSio-web," in *3rd Workshop on Formal Integrated Development Environment (F-IDE), satellite workshop of Formal Methods 2016*. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2016.

[61] R. M. Smullyan, *First-order logic*. New York: Dover publications, 1995.

[62] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, "Quantitative verification of implantable cardiac pacemakers over hybrid heart models," *Information and Computation*, vol. 236, no. 0, pp. 87–101, 2014, special Issue on Hybrid Systems and Biology. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540114000157

[63] P. Ye, E. Entcheva, and S. A. Smolka, "Efficient modeling of excitable cells using hybrid automata," in *Computational Methods in System Biology*, 2005, pp. 216–227.

[64] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 278–292. [Online]. Available: http://dl.acm.org/citation.cfm?id=788018.788803

[65] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam, "Modeling and verification of a dual chamber implantable pacemaker," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Flanagan and B. König, Eds. Springer Berlin Heidelberg, 2012, vol. 7214, pp. 188–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28756-5_14

[66] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397594900108

[67] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on UPPAAL," in *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer Verlag, 2004, pp. 200–237. [Online]. Available: http://doc.utwente.nl/51010/

[68] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and computation*, vol. 111, no. 2, pp. 193–244, 1994.

[69] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, pp. 219–246, 1989.

[70] A. David, K. G. Larsen, A. Legay, M. H. Møller, U. Nyman, A. P. Ravn, A. Skou, and A. Wąsowski, "Compositional verification of real-time systems using ECDAR," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 6, pp. 703–720, 2012.

[71] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed I/O automata: a complete specification theory for real-time systems," in *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*. ACM, 2010, pp. 91–100.

[72] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on concurrency and Petri nets*. Springer, 2004, pp. 87–124.

[73] P. Masci, P. Mallozzi, F. De Angelis, G. D. M. Serugendo, and P. Curzon, "Using PVSio-web and SAPERE for rapid prototyping of user interfaces in Integrated Clinical Environments," in *in Verisure2015, Workshop on Verification and Assurance, co-located with CAV*, 2015.

[74] F. Zambonelli, A. Omicini, B. Anzengruber, G. Castelli, F. L. De Angelis, G. D. M. Serugendo, S. Dobson, J. L. Fernandez-Marquez, A. Ferscha, M. Mamei *et al.*, "Developing pervasive multi-agent systems with nature-inspired coordination," *Pervasive and Mobile Computing*, vol. 17, pp. 236–252, 2015.

[75] T. A. Henzinger, M. Minea, and V. S. Prabhu, "Assume-guarantee reasoning for hierarchical hybrid systems." in *HSCC*, ser. Lecture Notes in Computer Science, M. D. D. Benedetto and A. L. Sangiovanni-Vincentelli, Eds., vol. 2034. Springer, 2001, pp. 275–290.

[76] J. Sametinger, J. Rozenblit, R. Lysecky, and P. Ott, "Security challenges for medical devices," *Communications of the ACM*, vol. 58, no. 4, pp. 74–82, 2015.

APPENDIX

PVS THEORIES

```
pacemaker: THEORY
BEGIN
IMPORTING LRI, AVI, PVARP, URI, VRP

devState: TYPE = [#
  avi: AVI.state,
  lri: LRI.state,
  pvarp: PVARP.state,
  uri: URI.state,
  vrp: VRP.state
#]

pmode: TYPE = { ON, OFF }

State: TYPE = [#
  dev: devState,
  aptime: real,        % pulse for AP
  apon: bool,
  vptime: real,        % pulse for VP
  vpon: bool,
  %-- pacemaker inputs
  Abeat: nat,
  Vbeat: nat,
  %-- pacemaker outputs
  AP: nat,
  VP: nat
#]

init_pacemaker: State = (#
  dev := (#
           avi := init_AVI,
           lri := init_LRI,
           pvarp := init_PVARP,
           uri := init_URI,
           vrp := init_VRP
         #),
  aptime := 0,
  apon := false,
  vptime := 0,
  vpon := false,
  AP := 0,
  VP := 0,
  Abeat := 0,
  Vbeat := 0
#)


%-- events generated by LRI
en_APevent(st: State): bool =
  en_APout(lri(dev(st)))
  AND en_APin(avi(dev(st)))

APevent(st: (en_APevent)): State =
  st WITH [
      dev := dev(st) WITH [
          lri := APout(lri(dev(st))),
          avi := APin(avi(dev(st)))
          ],
      apon := IF (aptime(st) < APWIDTH)
              THEN true ELSE false ENDIF
     ]

en_LRItau(st: State): bool =
  en_tau(lri(dev(st)))

LRItau(st: (en_LRItau)): State =
  st WITH [
      dev := dev(st)
          WITH [ lri := tau(lri(dev(st))) ] ]


%-- events generated by AVI
en_VPevent(st: State): bool =
  en_VPout(avi(dev(st))) AND
  (
  en_VPin(lri(dev(st))) OR
  en_VPin(pvarp(dev(st))) OR
  en_VPin(vrp(dev(st))) OR
  en_VPin(uri(dev(st)))
  )

VPevent(st: (en_VPevent)): State =
  st WITH [
      dev := dev(st) WITH [
        avi := VPout(avi(dev(st))),
        lri := IF (en_VPin(lri(dev(st))))
               THEN VPin(lri(dev(st)))
               ELSE lri(dev(st))
               ENDIF,
        pvarp := IF (en_VPin(pvarp(dev(st))))
                 THEN VPin(pvarp(dev(st)))
                 ELSE pvarp(dev(st))
                 ENDIF,
        uri := IF (en_VPin(uri(dev(st))))
               THEN VPin(uri(dev(st)))
               ELSE uri(dev(st))
               ENDIF,
        vrp := IF (en_VPin(vrp(dev(st))))
               THEN VPin(vrp(dev(st)))
               ELSE vrp(dev(st))
               ENDIF
      ],
      vpon := IF (vptime(st) < VPWIDTH)
                  THEN true ELSE false ENDIF
  ]

en_AVItau(st: State): bool =
  en_tau(avi(dev(st)))

AVItau(st: (en_AVItau)): State =
  st WITH [ dev := dev(st) WITH
                 [ avi := tau(avi(dev(st))) ]]


%-- events generated by PVARP
en_ASevent(st: State): bool =
  en_ASout(pvarp(dev(st))) AND
  (
  en_ASin(lri(dev(st))) OR
  en_ASin(avi(dev(st)))
  )
ASevent(st: (en_ASevent)): State =
  st WITH [
      dev := dev(st) WITH [
          pvarp := ASout(pvarp(dev(st))),
          avi := IF (en_ASin(avi(dev(st))))
                 THEN ASin(avi(dev(st)))
                 ELSE avi(dev(st))
                 ENDIF,
          lri := IF (en_ASin(lri(dev(st))))
                 THEN ASin(lri(dev(st)))
                 ELSE lri(dev(st))
                 ENDIF
      ]]

en_PVARPtau(st: State): bool =
  en_tau(pvarp(dev(st)))
```

```
PVARPtau(st: (en_PVARPtau)): State =
  st WITH [ dev := dev(st) WITH
      [ pvarp := tau(pvarp(dev(st))) ]]


%-- events generated by VRP
en_VSevent(st: State): bool =
  en_VSout(vrp(dev(st))) AND
  (
  en_VSin(lri(dev(st))) OR
  en_VSin(avi(dev(st))) OR
  en_VSin(pvarp(dev(st))) OR
  en_VSin(uri(dev(st)))
  )

VSevent(st: (en_VSevent)): State =
  st WITH [
     dev := dev(st) WITH [
        vrp := VSout(vrp(dev(st))),
        lri := IF (en_VSin(lri(dev(st))))
               THEN VSin(lri(dev(st)))
               ELSE lri(dev(st))
               ENDIF,
        pvarp := IF (en_VSin(pvarp(dev(st))))
               THEN VSin(pvarp(dev(st)))
               ELSE pvarp(dev(st))
               ENDIF,
        uri := IF (en_VSin(uri(dev(st))))
               THEN VSin(uri(dev(st)))
               ELSE uri(dev(st))
               ENDIF,
        avi := IF (en_VSin(avi(dev(st))))
               THEN VSin(avi(dev(st)))
               ELSE avi(dev(st))
               ENDIF
     ]
   ]

en_VRPtau(st: State): bool =
  en_tau(vrp(dev(st)))

VRPtau(st: (en_VRPtau)): State =
  st WITH [ dev := dev(st) WITH
               [ vrp := tau(vrp(dev(st))) ]]


%-- external events
en_Abeatevent(st: State): bool =
  Abeat(st) = 1 AND
  en_Abeatin(pvarp(dev(st))) AND
  (
  en_ASin(lri(dev(st))) OR
  en_ASin(avi(dev(st)))
  )

Abeatevent(st: State): State =
  st WITH [
    dev :=
      dev(st) WITH [
        pvarp :=
            ASout(Abeatin(pvarp(dev(st)))),
        lri := IF (en_ASin(lri(dev(st))))
               THEN ASin(lri(dev(st)))
               ELSE lri(dev(st))
               ENDIF,
        avi := IF (en_ASin(avi(dev(st))))
               THEN ASin(avi(dev(st)))
               ELSE avi(dev(st))
               ENDIF
      ]]

en_Vbeatevent(st: State): bool =
  Vbeat(st) = 1 AND
  en_Vbeatin(vrp(dev(st))) AND
  (
  en_VSin(lri(dev(st))) OR
  en_VSin(avi(dev(st))) OR
  en_VSin(pvarp(dev(st))) OR
  en_VSin(uri(dev(st)))
  )

Vbeatevent(st: State): State =
  st WITH [
      dev := dev(st) WITH [
        vrp := VSout(Vbeatin(vrp(dev(st)))),
        lri := IF (en_VSin(lri(dev(st))))
               THEN VSin(lri(dev(st)))
               ELSE lri(dev(st))
               ENDIF,
        avi := IF (en_VSin(avi(dev(st))))
               THEN VSin(avi(dev(st)))
               ELSE avi(dev(st))
               ENDIF,
        pvarp := IF (en_VSin(pvarp(dev(st))))
                 THEN VSin(pvarp(dev(st)))
                 ELSE pvarp(dev(st))
                 ENDIF,
        uri := IF (en_VSin(uri(dev(st))))
               THEN VSin(uri(dev(st)))
               ELSE uri(dev(st))
               ENDIF
        ]]

exec(st: State): State =
  LET
    st =
        st WITH [
            AP := IF (apon(st)) THEN 1
                  ELSE 0 ENDIF,
            VP := IF (vpon(st)) THEN 1
                  ELSE 0 ENDIF
        ],
    st =
        st WITH [
            dev := dev(st) WITH [
              avi := avi(dev(st)) WITH [
                  clk := clk(uri(dev(st))) ]
            ]]
  IN
  IF en_APevent(st) THEN APevent(st)
  ELSIF en_VPevent(st) THEN VPevent(st)
  ELSIF en_Abeatevent(st) THEN Abeatevent(st)
  ELSIF en_Vbeatevent(st) THEN Vbeatevent(st)
  ELSIF en_LRItau(st) THEN LRItau(st)
  ELSIF en_AVItau(st) THEN AVItau(st)
  ELSIF en_PVARPtau(st) THEN PVARPtau(st)
  ELSIF en_VRPtau(st) THEN VRPtau(st)
  ELSE st ENDIF


InputActions: TYPE = [#
  Abeat: nat,
  Vbeat: nat
#]

init_input: InputActions =
  (# Abeat := 0, Vbeat := 0 #)

advance_time(st: State): State =
```

```
  st WITH [
  dev :=
    dev(st) WITH [
    lri := lri(dev(st)) WITH
        [ time := time(lri(dev(st))) + 1 ],
    avi := avi(dev(st)) WITH
        [ time := time(avi(dev(st))) + 1 ],
    pvarp := pvarp(dev(st)) WITH
        [ time := time(pvarp(dev(st))) + 1 ],
    vrp := vrp(dev(st)) WITH
        [ time := time(vrp(dev(st))) + 1 ],
    uri := uri(dev(st)) WITH
        [ clk := clk(uri(dev(st))) + 1 ]
    ],
  aptime :=
    IF (apon(st) AND aptime(st) < APWIDTH)
    THEN aptime(st) + 1
    ELSE 0 ENDIF,
  apon := apon(st) AND aptime(st) < APWIDTH,
  vptime :=
      IF (vpon(st) AND vptime(st) < VPWIDTH)
      THEN vptime(st) + 1
      ELSE 0 ENDIF,
  vpon := vpon(st) AND vptime(st) < VPWIDTH
  ]

pacemaker_tick(ia: InputActions)(st: State)
        : State =
  LET
  st = st WITH
    [ Abeat := ia`Abeat, Vbeat := ia`Vbeat ],
  st = exec(st),
  st = advance_time(st)
  IN st

init(x: real): State = init_pacemaker

END pacemaker

AVI: THEORY
BEGIN

IMPORTING constants

Mode: TYPE = { Idle, AVI, WaitURI }

state: TYPE = [#
  time: real,      % for AVI cycle
  clk : real,      % for URI cycle
  loc: Mode
#]

init_AVI: state =
      (# time := 0, clk := 0, loc := Idle #)

en_ASin(st: state): bool = loc(st) = Idle
ASin(st: (en_ASin)): state =
  st WITH [ time := 0, loc := AVI ]

en_APin(st: state): bool =
  oc(st) = Idle
APin(st: (en_APin)): state =
  st WITH [ time := 0, loc := AVI ]

en_VSin(st: state): bool =
  loc(st) = AVI OR loc(st) = WaitURI
VSin(st: (en_VSin)): state =
  st WITH [ loc := Idle ]

en_VPout(st: state): boolean =
  (loc(st) = AVI
    AND time(st) >= TAVI AND clk(st) >= TURI)
  OR (loc(st) = WaitURI AND clk(st) >= TURI)
VPout(st: (en_VPout)): state =
  st WITH [ loc := Idle ]

en_tau(st: state): boolean =
  loc(st) = AVI AND time(st) >= TAVI
      AND clk(st) < TURI
tau(st: (en_tau)): state =
  st WITH [ loc := WaitURI ]

END AVI

LRI: THEORY
BEGIN

IMPORTING constants

Mode: TYPE = { LRI, ASed }

state: TYPE = [#
  time: real,
  loc: Mode
#]

init_LRI: state = (# time := 0, loc := LRI #)

en_APout(st: state): boolean =
  loc(st) = LRI AND time(st) >= TLRI-TAVI
APout(st: (en_APout)): state =
  (# time := 0, loc := LRI #)

en_VSin(st: state): boolean = true
VSin(st: (en_VSin)): state =
  (# time := 0, loc := LRI #)

en_VPin(st: state): boolean = true
VPin(st: (en_VPin)): state =
  (# time := 0, loc := LRI #)

en_ASin(st: state): bool = loc(st) = LRI
ASin(st: (en_ASin)): state =
  st WITH [ loc := ASed ]

en_tau(st: state): bool = false
tau(st: state): state = st

END LRI


PVARP: THEORY
BEGIN

IMPORTING constants

Mode: TYPE =
  { Idle, inter, inter1, PVAB, PVARP }

state: TYPE = [#
  time: real,
  loc: Mode
#]

init_PVARP: state =
  (# time := 0, loc := Idle #)

en_VPin(st: state): bool = loc(st) = Idle
VPin(st: (en_VPin)): state =
  (# time := 0, loc := PVAB #)
```

```
en_VSin(st: state): bool = loc(st) = Idle
VSin(st: (en_VSin)): state = VPin(st)

en_Abeatin(st: state): bool =
  loc(st) = Idle OR loc(st) = PVARP
Abeatin(st: (en_Abeatin)): state =
  COND
  loc(st) = Idle -> st WITH [ loc := inter ],
  loc(st) = PVARP ->
        st WITH [ loc := inter1 ]
  ENDCOND

en_ASout(st: state): bool = loc(st) = inter
ASout(st: (en_ASout)): state =
  st WITH [ loc := Idle ]

en_tau(st: state): boolean =
  (loc(st) = PVAB AND time(st) >= TPVAB) OR
  (loc(st) = PVARP AND time(st) >= TPVARP)
tau(st: (en_tau)): state =
  COND
  loc(st) = PVAB -> st WITH [ loc := PVARP ],
  loc(st) = PVARP -> st WITH [ loc := Idle ]
  ENDCOND

END PVARP


URI: THEORY
BEGIN

IMPORTING constants

state: TYPE = [#
  clk : real
#]

init_URI: state = (# clk := 0 #)

en_VPin(st: state): bool = true
VPin(st: (en_VPin)): state =
   st WITH [ clk := 0 ]

en_VSin(st: state): bool = true
VSin(st: (en_VSin)): state =
   st WITH [ clk := 0 ]

END URI
```

```
VRP: THEORY
BEGIN

IMPORTING constants

Mode: TYPE = { Idle, inter, VRP }

state: TYPE = [#
  time: real,
  loc: Mode
#]

init_VRP: state =
  (# time := 0, loc := Idle #)

en_Vbeatin(st: state): boolean =
  loc(st) = Idle
Vbeatin(st: (en_Vbeatin)): state =
  st WITH [ loc := inter ]

en_VSout(st: state): boolean =
  loc(st) = inter
VSout(st: (en_VSout)): state =
  st WITH [ time := 0, loc := VRP ]

en_VPin(st: state): boolean = loc(st) = Idle
VPin(st: (en_VPin)): state =
  st WITH [ time := 0, loc := VRP ]

en_tau(st: state): boolean =
  loc(st) = VRP AND time(st) >= TVRP
tau(st: (en_tau)): state =
  st WITH [ loc := Idle ]

END VRP

constants: THEORY
BEGIN

TURI: real = 250;
TLRI: real = 250;
TAVI: real = 75;
TPVARP: real = 125;
TPVAB: real = 20;
TVRP: real = 160;
APWIDTH: real = 50;
VPWIDTH: real = 50;

END constants
```