Modular code generation for emulating the electrical conduction system of the human heart

Nathan Allen, Sidharta Andalam, Partha Roop, Avinash Malik, Mark Trew and Nitish Patel
University of Auckland, New Zealand
Email: nall426@aucklanduni.ac.nz

***Abstract*—We study the problem of modular code generation for emulating the electrical conduction system of the heart, which is essential for the validation of implantable devices such as pacemakers. In order to develop high fidelity models, it is essential to consider the operation of hundreds, if not millions of conduction elements, called *nodes* of the heart. Published results so far, however, have considered a maximum of 33 nodes [1], modelled as Hybrid Input Output Automata (HIOA). The behaviour of this model is captured using the well known commercial tool Simulink®. These approaches are limiting due to the lack of model fidelity of the conduction system.**

In this paper, we first develop a semantic preserving modular compilation approach for a network of HIOA, by proposing to translate them to a network of Finite State Machines (FSMs). We then demonstrate that a delayed synchronous composition of the cardiac nodes enables modular code generation that is both semantic preserving and efficient. In addition to the above example, we have developed several examples from other domains to compare Simulink® and the developed tool called *Piha*. The results show that we are able to generate code which, for the cardiac model, is 60% smaller in binary size while executing 20 times faster when compared to Simulink®.

I. Introduction

Pacemakers are safety-critical Cyber-Physical Systems (CPSs) that control the pacing of a heart for providing therapy for bradycardia – abnormally slow pacing of the heart. Between 1990-2000, close to 200,000 pacemakers were recalled due to software related failures [2]. Consequently, there is a need for the development of better processes for validation and certification of such devices. We propose the well known, engineering technique of *emulation* [3] to tackle this problem. Emulation, also known as hardware-in-the-loop simulation, is used to validate controllers (such as motor controllers) by running them in closed-loop with the actual plant (e.g., the synchronous motor). However, for emulation of pacemakers, the use of the actual plant (i.e. animal / human organs) is limiting. Hence, there is a need for the development of high-fidelity heart models that can provide the required real-time response to facilitate emulation. Bioengineering heart models [4] provide excellent model fidelity at the expense of computation time as the simulation of a single heart beat may take several hours making them unsuitable for emulation.

Recently, timed automata [5] based heart models have been developed primarily for model checking. These models abstract the continuous dynamics and hence, are unsuitable for emulation. In contrast to this, Hybrid Input Output Automata (HIOA) [6], [7] is used for modelling the forward conduction system of the heart with 33 nodes in [1]. This work is the starting point for real-time emulation by demonstrating that Simulink® can be used for closed-loop verification of pacemakers. However, this work has limited model fidelity and the limitations of the tool Simulink® are inherited by the developed approach. Simulink® has semantic limitations, as the semantics of composition is unclear. Moreover, there is no direct correspondence between the Simulink® model and the HIOA models. Finally, Simulink® generated code has scalability issues when generating large networks, as we will show.

For modular code generation, we have developed an approach based on the well known synchronous languages [8]. Using the concept of delayed synchronous composition [9], we are then able to produce modular code for each node separately. The generated code is both smaller and has faster execution times. In addition, the generated code accurately captures the specification in HIOA, similar in spirit to Ptolemy [10] and Zélus [11]. However, unlike these approaches, the presented approach does not depend upon dynamic numerical solvers, which are not ideal for the *emulation* of the heart.

An overview of the proposed modular compilation approach is presented in Figure 1. It has two steps: (1) given a network of HIOAs, step 1 generates a semantic preserving FSM representation of each HIOA in the network. (2) Given a network of such FSMs, step 2 then composes them using the synchronous parallel operator (||) enabling generation of efficient 'C'-code. Step 1 is presented in Section IV. Step 2 is presented in Section V.

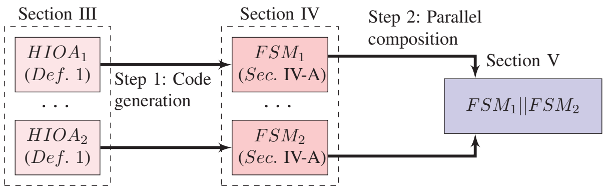
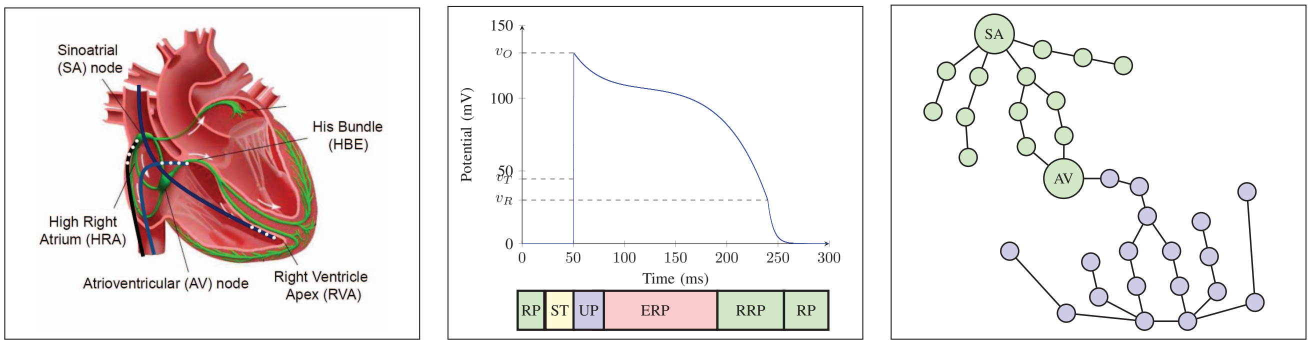


Fig. 1. Overview of the proposed modular code generation approach

Contributions of the paper are: (1) We have developed a *modular code generation* approach, for the first time for HIOA models (Section IV), which is also semantic preserving. The proposed approach performs code generation using a new synchronous semantics [8] of HIOA. More importantly, the developed semantics is not restrictive, unlike [12], [13] and does not interact with dynamic numerical solvers unlike [10], [11]. (2) We *quantitatively evaluate* the efficacy of the proposed

[1] Authors express their gratitude to the research group of Prof. Marta Kwiatkowska for sharing their Simulink® heart model reported in [1], which was foundational in developing the proposed models in this paper



(a) Diagram of the heart. Reproduced from [5] (b) The four stages of an Action Potential (AP) (c) An abstracted model of the conduction system

Fig. 2. Electrical conduction systems of the heart

approach relative Simulink® (Section VI) to demonstrate the scalability of the approach relative to the best known model in published literature [1]. Our experiments show that code generation is feasible for thousands of nodes of the heart, unlike the 33-node published model.

II. Background–the human heart

Using Figure 2, we describe the electrical conduction system of the heart. The human heart (see Figure 2(a)) is a biological pump that is essential for blood circulation using its rhythmic pumping activity. It achieves this by regularly contracting and relaxing its muscles, which is orchestrated through flow of electrical signals along a set of cellular groups called nodes and associated paths called its *conduction system*. The source of the signal is a natural pacemaker in the left ventricle, called the Sinoatrial (SA) node, which triggers automatically. First, this signal travels through the left and right atria, contracting the muscles and pushing the blood into the ventricles.

Second, to ensure both ventricles are filled, the Atrioventicular (AV) node introduces critical delay in the conduction system. Finally, the electrical signal propagates through both ventricles. This contracts the muscles and pumps the blood out of the heart.

Action Potential (AP). The propagation of electrical signals at the cellular / nodal level is described as a change in voltage across the cell membrane due to ionic flow. Over a period of time, this change is depicted as the Action Potential (AP) [1] (see Figure 2(b)). It can be described in four stages.

1) Resting Period (RP): This is the steady state, when the node is awaiting activation by an external stimulus.
2) Stimulated (ST): When the external stimulus is above a threshold voltage (V_T).
3) Upstroke (UP): After continuous stimulation, the cell's voltage reaches the threshold voltage (V_T. This causes the node to depolarises (inward flow of positive ions) and contracts the muscles. This depolarisation yields a stimulus that activates neighbouring nodes.
4) Effective Refractory Period (ERP): Once activated, the node cannot be activated again due to the recovery process of the ionic channels. Any new stimulus will be blocked during this refractory period.

Finally, the ionic channels will partially recover (Relative Refractory Period (RRP)) and then fully recover (RP). If activated again by external stimulus during RRP, the morphology of the AP will be shorter when compared to RP.

Abstract model. The human heart has over two trillion cells. For analytical purposes, a model consisting of a network of 33 nodes is presented in literature and is used for testing off-the-shelf pacemakers [5], [1]. The abstracted electrical conduction system consisting of nodes and paths is presented in Figure 2(c). The functionality of each node is described using Hybrid Input Output Automata (HIOA) in Section III. During implementation, the connections between nodes are implemented as buffers where the length of the buffer depends on the conduction delay and step size. Later in Section IV, we use this network of nodes to illustrate our modular code generation.

III. Hybrid Automata

Hybrid Input Output Automata (HIOA) is ideal for describing CPS. It is very expressive, non-deterministic and has been successfully used for capturing the behaviour of a plant in control systems. In this section, we use the HIOA in Figure 3, which captures the AP in Figure 2(b) as a running example to provide a formal definition and an informal description of the semantics of HIOA.

A. Example of Hybrid Input Output Automata (HIOA)

The heart node HIOA, shown in Figure 3, captures the behaviour of the four stages of AP shown in Figure 2(b). Each stage of the AP is captured using locations in the HIOA. The AP of every node, in the heart, evolves over time, this evolution of the AP is captured with three Ordinary Differential Equations (ODEs) evolving three individual variables: v_x, v_y, and v_z. The overall voltage of the AP is then aggregated into a single variable v as defined in [14]. The aggregated AP voltage v of each node affects its neighbouring nodes and vice-versa. This effect of the neighbouring voltages is captured in the HIOA via a function $g(\vec{v_I})$, where $\vec{v_I}$ represents the *vector* of input voltages. This function is shown in Equation 1 where k_i is a constant depending on the neighbour, and v refers to the node's current voltage. Finally, the dynamic response of

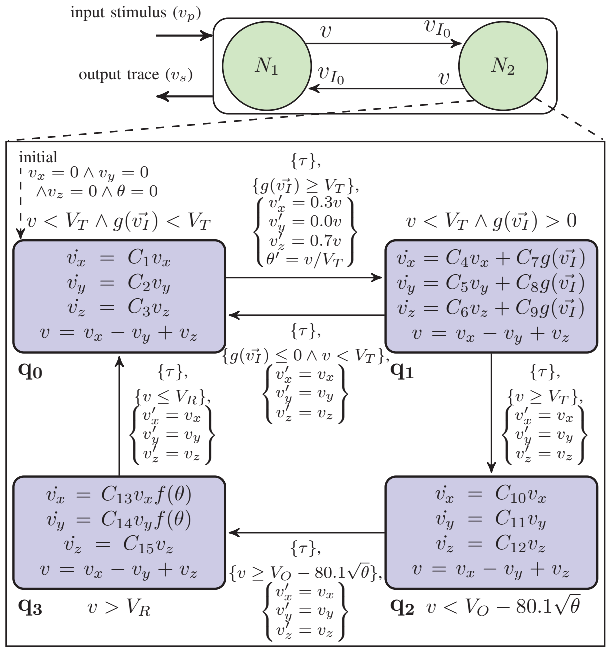

input stimulus (v_p)
output trace (v_s)
N_1
N_2
v
v_{I_0}
v_{I_0}
v
initial
$v_x = 0 \wedge v_y = 0$
$\wedge v_z = 0 \wedge \theta = 0$
$v < V_T \wedge g(\vec{v_I}) < V_T$
$\dot{v_x} = C_1 v_x$
$\dot{v_y} = C_2 v_y$
$\dot{v_z} = C_3 v_z$
$v = v_x - v_y + v_z$
$\mathbf{q_0}$
$\{\tau\}$, $\{g(\vec{v_I}) \geq V_T\}$, $\{v'_x = 0.3v, v'_y = 0.0v, v'_z = 0.7v, \theta' = v/V_T\}$
$v < V_T \wedge g(\vec{v_I}) > 0$
$\dot{v_x} = C_4 v_x + C_7 g(\vec{v_I})$
$\dot{v_y} = C_5 v_y + C_8 g(\vec{v_I})$
$\dot{v_z} = C_6 v_z + C_9 g(\vec{v_I})$
$v = v_x - v_y + v_z$
$\mathbf{q_1}$
$\{\tau\}$, $\{g(\vec{v_I}) \leq 0 \wedge v < V_T\}$, $\{v'_x = v_x, v'_y = v_y, v'_z = v_z\}$
$\{\tau\}$, $\{v \leq V_R\}$, $\{v'_x = v_x, v'_y = v_y, v'_z = v_z\}$
$\{\tau\}$, $\{v \geq V_T\}$, $\{v'_x = v_x, v'_y = v_y, v'_z = v_z\}$
$\dot{v_x} = C_{13} v_x f(\theta)$
$\dot{v_y} = C_{14} v_y f(\theta)$
$\dot{v_z} = C_{15} v_z$
$v = v_x - v_y + v_z$
$\mathbf{q_3}$ $v > V_R$
$\{\tau\}$, $\{v \geq V_O - 80.1\sqrt{\theta}\}$, $\{v'_x = v_x, v'_y = v_y, v'_z = v_z\}$
$\dot{v_x} = C_{10} v_x$
$\dot{v_y} = C_{11} v_y$
$\dot{v_z} = C_{12} v_z$
$v = v_x - v_y + v_z$
$\mathbf{q_2}$ $v < V_O - 80.1\sqrt{\theta}$

Fig. 3. Top of the figure shows the interaction between two heart nodes (N_1 and N_2). Each node is defined as a Hybrid Input Output Automata (HIOA)

the action potential to secondary excitation is captured by the variable θ.

$$g(\vec{v_I}) = \sum_i k_i(v_{I_i} - v) \tag{1}$$

For the first stage (RP), the continuous dynamics are captured in location $\mathbf{q_0}$. Ordinary Differential Equations (ODEs), for example $\dot{v_x} = C_1 v_x$, capture the continuous evolution of components of v (v_x, v_y, v_z) when control resides in location $\mathbf{q_0}$ while the location invariant $v < V_T \wedge g(\vec{v_I}) < V_T$ holds. Once the external stimulus, $g(\vec{v_I})$, crosses some threshold V_T, indicated by the predicate $g(\vec{v_I}) \geq V_T$ on the edge connecting locations $\mathbf{q_0}$ and $\mathbf{q_1}$, the HIOA transitions to location $\mathbf{q_1}$ (ST) *instantaneously*. During this transition, the variables (e.g, v'_x) are updated. These updated values of the variables are used as initial conditions from which the variables evolve further via the ODEs in the target location.

Once the voltage of the node (v) exceeds the threshold voltage V_T, it begins its UP phase (captured by location $\mathbf{q_2}$). The final location, $\mathbf{q_3}$, represents the ERP stage. The constants C_1 through to C_{15} configure the morphology of the AP. Furthermore, constants V_T, V_O, and V_R determine the height of the AP and transitions between locations of the HIOA to further capture the morphology. We formalise the HIOA using Definition 1.

Definition 1: A Hybrid Input Output Automata (HIOA) $\mathcal{H} = \langle Loc, Edge, \Sigma_I, \Sigma_{EO}, \Sigma_{EI}, X_I, X_{EO}, X_{EI}, Init, Inv, Flow, Up, Jump\rangle$ where

- $Loc = \{l_1, .., l_n\}$ representing n locations.
- $\Sigma_E = \Sigma_{EI} \cup \Sigma_{EO}$ is the set of external events. Σ_{EI} and Σ_{EO} are the set of external input and external output events, respectively. Furthermore, $\Sigma_{EI} \cap \Sigma_{EO} = \emptyset$.
- $\Sigma = \Sigma_E \cup \{\tau\}$ is the set of event names comprising of external and the $true$ event, respectively.
- $Edge \subseteq Loc \times \mathcal{B}(\Sigma_{EI} \cup \{\tau\}) \times 2^{\Sigma_{EO}} \times Loc$ are the set of edges between locations.
- There are three sets of internal continuous variables, their rate of change and their updated values are represented as follows: $X_I = \{x_1, .., x_m\}$, $\dot{X_I} = \{\dot{x_1}, .., \dot{x_m}\}$, and $X'_I = \{x'_1, .., x'_m\}$.
- The set of external continuous variables: $X_E = X_{EI} \cup X_{EO}$, where X_{EI} and X_{EO} refer to input and output variables respectively. Furthermore, $X_{EI} \cap X_{EO} = \emptyset$.
- $Init(l)$: A predicate whose free variables are from X_I. It specifies the possible valuations of these when the HIOA starts in l.
- $Inv(l)$: A predicate whose free variables are from $X_I \cup X_E$ and it constrains these when the HIOA resides in l.
- $Flow(l)$: A predicate whose free variables are from $X_I \cup \dot{X_I}$ and it specifies the rate of change of these variables when the HIOA resides in l.
- $Up(l)$: A function that assigns to each variable in set X_{EO} an algebraic expression involving the variables from $X_I \cup X_{EI}$.
- $Jump(e)$: A function that assigns to the edge 'e' a predicate whose free variables are from $X_I \cup X'_I \cup X_E$. This predicate specifies when the location switch using 'e' is possible. It also specifies the updated values of the variables when this location switch happens.

For the HIOA in Figure 3, $Loc = \{q_0, q_1, q_2, q_3\}$, $\Sigma_{EI} = \emptyset$, $\Sigma_{EO} = \emptyset$, and $\Sigma = \{\tau\}$. $X_I = \{\theta, v_x, v_y, v_z\}$, $\dot{X_I} = \{\dot{\theta}, \dot{v_x}, \dot{v_y}, \dot{v_z}\}$, $X'_I = \{\theta', v'_x, v'_y, v'_z\}$. $X_{EI} = \{v_i | \forall v_i \in \vec{v_I}\}$. $X_{EO} = \{v\}$. An example of an initial predicate is: $Init(l) : (v_x = 0 \wedge v_y = 0 \wedge v_z = 0 \wedge \theta = 0)$. An example edge connecting locations $\mathbf{q_0}$ and $\mathbf{q_1}$ is represented with the tuple: $(q_0, \tau, \emptyset, q_1)$. The example of invariant (Inv) predicate on location $\mathbf{q_0}$ is: $v < V_T \wedge g(\vec{v_I}) < V_T$. The $Jump$ condition on the edge connecting locations $\mathbf{q_0}$ and $\mathbf{q_1}$ is enabled when the predicate $g(\vec{v_I}) \geq V_T$ is true, and the updated variables are as shown in Figure 3. An example of the Up predicate is the function $v = v_x - v_y + v_z$ in all locations in Figure 3. Finally, the $Flow$ predicate in location $\mathbf{q_0}$ is the set of ODEs, shown in Figure 3.

IV. Modular code generation

In this section we present our approach for modular code generation from a network of HIOAs as it is implemented in our tool, called Piha. It is a two step procedure as presented previously in Figure 1. Firstly, we create a FSM where all ODEs from the HIOA are replaced by their so called witness functions, i.e., static numerical solutions. We describe the translation of the ODEs to their equivalent witness functions in Section IV-A followed by code generation for a single HIOA through the use of a Mealy FSM in Section IV-B.

A. Converting the ODEs into equivalent witness functions

Given the ODE for $\dot{v_x}$ in location $\mathbf{q_0}$ of the heart node, and the HIOA defined in Figure 3 (reproduced in Equation (2)), the witness function computes the updated value of v_x (denoted v_x') at discrete time instants, separated by the so called time step δ ($\delta \in \mathbb{R}^+$), using the forward Euler method.

The witness function, for the ODE evolving v_x, implementing the forward Euler method is shown in Equation (3). Equation (3) evolves v_x *iteratively* while the invariant condition on location $\mathbf{q_0}$ ($v < V_T \wedge g(\vec{v_I}) < V_T$) holds. The initial value from which v_x starts evolving, in $\mathbf{q_0}$ is either: (1) specified by the programmer in the $Init$ predicate or (2) the final value of v_x when an instantaneous transition is made from location $\mathbf{q_3}$ or $\mathbf{q_1}$ to $\mathbf{q_0}$, shown as update $v_x' = v_x$ on the edge connecting these locations in Figure 3.

$$\dot{v_x} = C_1 v_x \tag{2}$$
$$v_x' = v_x + \delta \times (C_1 v_x) \tag{3}$$

B. Backend code generation for a single HIOA

In order to facilitate code generation, each HIOA is transformed into a simple Mealy FSM. The FSM generated from the HIOA in Figure 3 is shown in Figure 4.

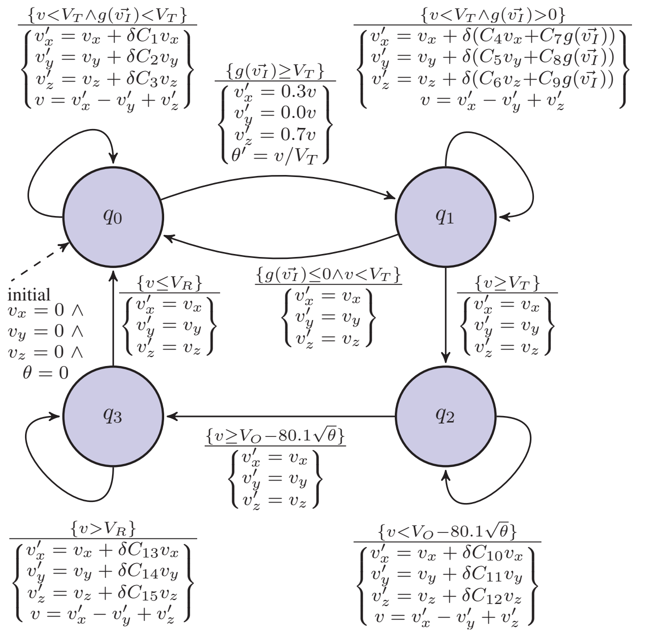
The translation from a single HIOA to its FSM representation is a two step procedure:

(1) *Translating locations into states*: The first step translates each location in the HIOA into a state in the FSM. Hence, the set of locations $\{q_0, q_1, q_2, q_3\}$ have an equivalent named set of states in the generated FSM.

(2) *Translating each edge in the HIOA to a transition in the FSM*: The second step translates each edge (e) in the HIOA into an equivalent transition (t) in the FSM. Every transition (t) in the FSM is of the form: $\frac{guard}{update}$. The $guard$ precedent is the conjunction of the set $\Sigma_{EI} \cup \{\tau\}$ and the conditions specified in the $Jump$ predicate on edge e. The $update$ antecedent is the conjunction of the set Σ_{EO} and the update extracted from the $Jump$ predicate on edge e. For example, the edge $(q_0, \tau, \emptyset, q_1)$ with the $Jump$ predicate as shown in Figure 3 translated into a transition in the generated FSM is $\frac{\tau \wedge g(\vec{v_I}) \geq V_T}{v_x'=0.3v, v_y'=0.0v, v_z'=0.7v, \theta'=v/V_T}$. Note, that in Figure 4, we have ignored the τ events, because they always evaluate to $true$.

The aforementioned steps generate an FSM that is able to capture the instantaneous transitions across locations called *inter-location* transitions, but is unable to capture the evolution of the continuous variables via the ODEs. In order to replicate the continuous evolution of variables within the locations, we create *self-transitions* on each state in the FSM. A self-transition by definition has the same source and target state, e.g., transition from state $\mathbf{q_0}$ to $\mathbf{q_0}$ as shown in Figure 4. The invariant from the initial HIOA becomes the *guard* of the self-transition, and the witness functions for each of the ODEs described in Section IV-A become the *update* of the transition.

Inter-location transitions may be enabled by external inputs (continuous variables from set X_{EI} or discrete events from set Σ_{EI}). For example, the edge connecting $\mathbf{q_0}$ to $\mathbf{q_1}$, in Figure 3,



$\{v<V_T \wedge g(\vec{v_I})<V_T\}$
$v_x' = v_x + \delta C_1 v_x$, $v_y' = v_y + \delta C_2 v_y$, $v_z' = v_z + \delta C_3 v_z$, $v = v_x' - v_y' + v_z'$
$\{g(\vec{v_I}) \geq V_T\}$
$v_x' = 0.3v$, $v_y' = 0.0v$, $v_z' = 0.7v$, $\theta' = v/V_T$
$\{v<V_T \wedge g(\vec{v_I})>0\}$
$v_x' = v_x + \delta(C_4 v_x + C_7 g(\vec{v_I}))$, $v_y' = v_y + \delta(C_5 v_y + C_8 g(\vec{v_I}))$, $v_z' = v_z + \delta(C_6 v_z + C_9 g(\vec{v_I}))$, $v = v_x' - v_y' + v_z'$
q_0
q_1
q_2
q_3
initial
$v_x = 0 \wedge$ $v_y = 0 \wedge$ $v_z = 0 \wedge$ $\theta = 0$
$\{v \leq V_R\}$
$v_x' = v_x$, $v_y' = v_y$, $v_z' = v_z$
$\{g(\vec{v_I}) \leq 0 \wedge v<V_T\}$
$v_x' = v_x$, $v_y' = v_y$, $v_z' = v_z$
$\{v \geq V_T\}$
$v_x' = v_x$, $v_y' = v_y$, $v_z' = v_z$
$\{v \geq V_O - 80.1\sqrt{\theta}\}$
$v_x' = v_x$, $v_y' = v_y$, $v_z' = v_z$
$\{v>V_R\}$
$v_x' = v_x + \delta C_{13} v_x$, $v_y' = v_y + \delta C_{14} v_y$, $v_z' = v_z + \delta C_{15} v_z$, $v = v_x' - v_y' + v_z'$
$\{v<V_O - 80.1\sqrt{\theta}\}$
$v_x' = v_x + \delta C_{10} v_x$, $v_y' = v_y + \delta C_{11} v_y$, $v_z' = v_z + \delta C_{12} v_z$, $v = v_x' - v_y' + v_z'$

Fig. 4. Finite State Machine (FSM) of a heart node . We ignore the τ event in the FSM, because it always evaluates to $true$.

is enabled as soon as the aggregate voltage of neighbouring node ($g(\vec{v_I})$) is greater than or equal to some threshold voltage V_T. In order to ensure that such inter-location transitions are enabled correctly, we need to enforce that all self-transitions in the generated FSM are the lowest priority transitions. This is accomplished by updating the guard condition of the self-transitions as follows:

- $guard \leftarrow guard \wedge \bigwedge_{\forall \sigma \in \Sigma_{EI}} \neg\sigma$

In the HIOA of Figure 3, $\Sigma_{EI} = \emptyset$, hence, the self-transition guard conditions remain the same.

Each of the generated FSMs in the network may be individually transformed into a switch case statement in 'C'-code as a `Step` function, with corresponding `Initialization` and `Input-Output` functions to initialize the FSM and communicate with its environment, respectively.

V. Parallel Composition

A. Mapping between FSMs

Given a network of FSMs, the interactions between each FSM are defined as follows:

- There exists a set of continuous variable inputs from all FSMs globally X_{GI}, and a set of continuous variable outputs from all FSMs globally X_{GO}.
- There exists a mapping function $\lambda(x_i) : x_o$ who takes as input one of X_{GI} and returns as output one of X_{GO}.

Such a mapping function indicates that the value of the continuous variable input x_i should be updated with the value from the continuous variable output x_o. Each input x_i may only be mapped to at most one output from X_{GO}, however multiple inputs from X_{GI} may be mapped to the same output x_o.

For the example from Figure 3, $X_{GI} = \{N_1.v_{I_0}, N_2.v_{I_0}\}$ and $X_{GO} = \{N_1.v, N_2.v\}$. The mapping function λ consists of $\lambda(N_1.v_{I_0})$: $N_2.v$ and $\lambda(N_2.v_{I_0})$: $N_1.v$.

B. Synchronous Parallel

In order to compose the FSMs together, we take inspiration from the synchronous language SL [9]. The concept of *ticks* and *reactions* are carried over, whereby each FSM performs only a single transition ("tick") until all other FSMs have also completed a transition ("reaction") and the process can repeat. In order to deal with data dependencies, we also implement the delayed semantics of *pre* whereby the value of all inputs to each FSM are not updated with new values until the end of each reaction. This allows for the behaviour of the system to be agnostic to the scheduling order of the individual FSMs. This concept also enables us to simplify the process of handling cyclic ODEs (an important issue not considered in related work [13]) by causing the dependencies to reference previous, rather than current, values.

Piha orchestrates FSMs through the use of a round-robin scheduler. It executes one tick of each FSM (the `Step` function described in Section IV-B) in series, followed by an *I/O Synchronisation Stage* at the end of each reaction. At the end of each reaction, all outputs are emitted and new inputs are sampled. This process continues indefinitely to emulate the dynamics of the system. In addition, when the system initially starts up it enters an *Initialisation Stage* whereby the `Initialization` function is executed. Due to the scheduling order agnosticism described earlier, Piha can be extended to support parallel computation by executing different FSMs on separate threads (whether logical or physical)

VI. Benchmarking

We present a set of experiments to evaluate the efficacy of the proposed modular code generation tool (Piha) against Simulink®. In the first experiment, we compare the *scalability* of the two tools as the number of nodes in the Network of Heart Nodes (NHN) model increases. In the second experiment, we select benchmarks that span across different application domains to illustrate the *diversity* of the proposed approach. All benchmarks are available online [15].

A. Experimental set-up

The following aspects were considered in order to achieve a fair comparison between Piha and Simulink®. (1) **Solver**: To reflect the synchronous execution model, we used a discrete numerical solver with a fixed step in Simulink®, namely `ode1` (Forward Euler). (2) **Step Size**: The fixed step size in Simulink® and in Piha is 0.01 ms and δ = 0.01 ms, respectively. (3) **Time**: All benchmarks were simulated for 10 seconds. With a step size of 0.01 milliseconds this translates to 1 million iterations in Piha. (4) **Compiler**: All code was compiled using the Microsoft Visual C++ compiler. Piha code was compiled using both no optimisation (`O0`) and `O2` optimisation. Simulink® code was compiled using the automatically generated Makefile. The experiments were evaluated using an Intel i7-4790 processor with 8 GB RAM on Windows 7.

TABLE I
Benchmark descriptions

Benchmarks	#L	Description
Thermostat Network (TSN). Physics [16]	(2^{50})	50 thermostats heating multiple rooms to keep them warm
Network of Heart Nodes (NHN). Biology [1]	(4^{33})	Captures the electrical conduction system of a heart with 33 nodes
Water Heating System (WH). Physics [7]	(4^{50})	Models the heating of water in a series of 50 tank
Multiple Train Gate (MTG). Industrial automation [17]	(2×3^{30})	Models the behaviour of a gate and 30 trains at a rail road crossing
Nuclear Plant control (NP). Industrial automation [6]	$(3^{30} \times 3^{30})$	Switches between two fuel rods in 30 reactors to avoid a meltdown

B. Scalability

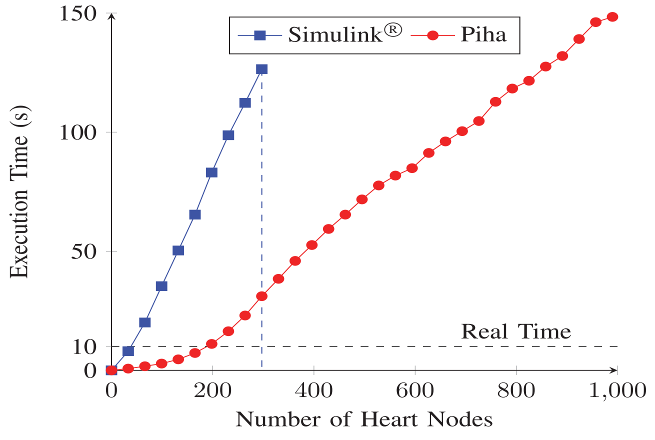


Fig. 5. Scalability in execution time of Simulink® and Piha

In this experiment, we validate the scalability of Piha through the running example of the NHN whilst comparing it to Simulink®. Code was generated for varying network sizes (33, 66, 99 nodes, etc.) and the execution time recorded. When simulating the 33 node network, a correlation co-efficient of 0.9994 was calculated between the output of Piha and Simulink®, illustrating its correctness relative to Simulink®.

The results are shown in Figure 5, no data is recorded for Simulink® for complexities greater than 297 nodes. Simulink® imposes an inbuilt requirement that the generated code use less than 2 GB of memory. This discontinuity represents the point after which the memory usage exceeds this limit (1.8 GB). Piha, on the other hand, is able to continue past this point.

These results also illustrate that Piha has a smaller increase in Execution Time as network size increases. For the real time constraint of 10 seconds for the benchmark, this means Piha is able to emulate a model roughly 5 times larger (200 nodes vs 40 nodes) than Simulink®. This can be seen through the line at 10 seconds - the point at which the simulated time is equal to the execution time. It is also of note that the change in gradient of Piha around the 200 node mark is due to the memory usage exceeding the L3 cache size (8 MB in our CPU).

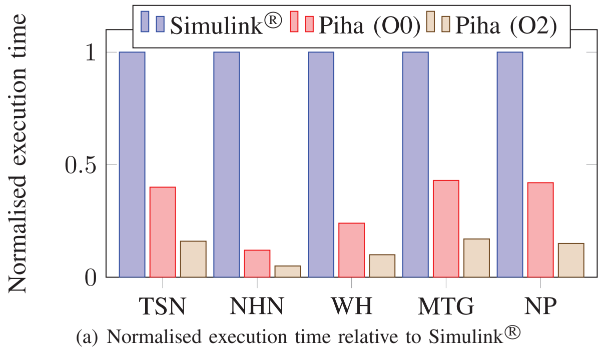
C. Diversity

For the second experiment, we use the benchmarks presented in Table I where #L represents the number of locations in each hybrid automata. For example, (2^{50}) denotes that the Thermostat Network (TSN) benchmark is described by 50 instances of an HIOA with two locations.

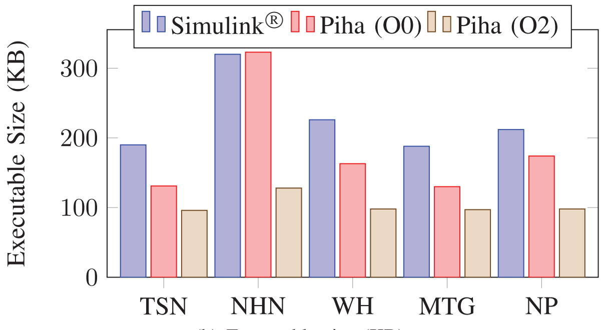
For all the benchmarks, the executable for the Simulink® models are generated using the in-built Real-time Workshop® 'C' code generator. Similarly, for Piha, we generate equivalent 'C' code. The execution times and executable sizes of the generated programs are presented in Figure 6.

Execution time: Figure 6(a) shows that for all benchmarks the execution time of Piha (both without optimisation and with optimisation level `O2`) is faster than that of Simulink®. On average, Piha is 9.8 times faster than Simulink®. For our most complicated example, the NHN, we observe an improvement of 20.3 times.

Code size: Figure 6(b) shows that the code generated by Piha is also, generally, more compact than that generated by Simulink®. On average, the optimised code of Piha is 54% smaller than Simulink® when compiled. For the NHN example, the unoptimised code of Piha is comparable to that of Simulink® while the optimised code sees improvements similar to that of the other benchmarks.



(a) Normalised execution time relative to Simulink®



(b) Executable size (KB)

Fig. 6. Comparison of the execution time and executable size

VII. Conclusions

We propose an approach of modular code generation from hybrid input output automata (HIOA) for the emulation of diverse physical processes. We have used the developed approach to model a complex electrical conduction network of a human heart. Each node of the network is modelled as a single HIOA. This composition of nodes is achieved using the concept of delayed synchronous composition, similar to the SL language [9]. This enables the nodes to effectively operate in a truly parallel manner as all input-output dependencies are delayed. The developed approach, "compiles away" the continuous dynamics to produce pure synchronous code for each node with delayed composition between nodes for creating the behaviour of a large conduction pathway with many nodes. Such an approach for modular and scalable code generation from hybrid automata without the reliance on external numerical solvers is new.

We show that our approach for modular code generation from HIOA can provide, on average 9.8 times faster execution and 54% smaller executables when compared to Simulink®. We also show that for the Network of Heart Nodes (NHN) benchmark our approach is capable of emulation of networks that are 5 times more complex. In the future, we will compare our approach quantitatively with Zélus [11], which is a synchronous code generator from hybrid automata that relies on dynamic interactions with numerical solvers. We will also model re-entrant behaviour in the conduction pathway, unlike the forward conduction system modelled here.

References

[1] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, "Quantitative verification of implantable cardiac pacemakers over hybrid heart models," *Information and Computation*, vol. 236, pp. 87–101, 2014.

[2] H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman, "Analysis of safety-critical computer failures in medical devices," *Security & Privacy, IEEE*, vol. 11, no. 4, pp. 14–26, 2013.

[3] K. N. Patel and R. H. Javeri, "A Survey on Emulation Testbeds for Mobile Ad-hoc Networks," *Procedia Computer Science*, vol. 45, pp. 581–591, 2015.

[4] N. Trayanova, "Your personal virtual heart," *Spectrum, IEEE*, vol. 51, pp. 34–59, November 2014.

[5] M. Zhihao J, Pajic and R. Mangharam, "Cyber Physical Modeling of Implantable Cardiac Medical Devices," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 122–137, 2012.

[6] R. Alur, *Principles of Cyber-Physical Systems*. MIT press, April 2015.

[7] J.-F. Raskin, *Handbook of Networked and Embedded Control Systems*, ch. An introduction to hybrid automata, pp. 491–517. Springer, 2005.

[8] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, pp. 64–83, Jan 2003.

[9] F. Boussinot and R. de Simone, "The SL synchronous language," *Software Engineering, IEEE Transactions on*, vol. 22, pp. 256–266, Apr 1996.

[10] C. Ptolemaeus, *System Design, Modeling, and Simulation: Using Ptolemy II*. Ptolemy. org, 2014.

[11] T. Bourke and M. Pouzet, "Zélus: a synchronous language with ODEs," in *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pp. 113–118, ACM, 2013.

[12] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky, "Generating embedded software from hierarchical hybrid models," *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 171–182, 2003.

[13] J. Kim and I. Lee, "Modular code generation from hybrid automata based on data dependency," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pp. 160–168, IEEE, 2003.

[14] P. Ye, E. Entcheva, S. A. Smolka, and R. Grosu, "Modelling Excitable Cells Using Cycle-Linear Hybrid Automata," *IET Systems Biology*, vol. 2, pp. 24 – 32, Jan. 2008.

[15] "DATE 2016 Benchmarks." https://github.com/PRETgroup/PihaBenchmarks last accessed - 18.09.2015.

[16] H. Joao Pedro, "How to describe a hybrid system? Formal models for hybrid system." University of California at Santa Barbara, Course ECE229, Lecture 2. http://www.ece.ucsb.edu/˜hespanha/ece229/Lectures/ Lecture2.pdf, 2005.

[17] C. Brennon and E. Joshua, "Hybrid Systems." Tufts University, Course EE194, Lecture. http://www.eecs.tufts.edu/˜khan/Courses/Spring2013/EE194/ Lecs/ Hybrid_ Systems_Presentation_Elliott_Costello.pdf, 2013.