Department of Computer Science

Building Power Consumption Models from Executable Timed I/O Automata Specifications

Benoît Barbot, Marta Kwiatkowska, Alexandru Mereacre and Nicola Paoletti

CS-RR-16-01

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD

ABSTRACT

We develop a novel model-based hardware-in-the-loop (HIL) framework for optimising energy consumption of embedded software controllers. Controller and plant models are specified as networks of parameterised timed input/output automata and translated into executable code. The controller is encoded into the target embedded hardware, which is connected to a power monitor and interacts with the simulation of the plant model. The framework then generates a power consumption model that maps controller transitions to distributions over power measurements, and is used to optimise the timing parameters of the controller, without compromising a given safety requirement. The novelty of our approach is that we measure the real power consumption of the controller and use thus obtained data for energy optimisation. We employ timed Petri nets as an intermediate representation of the executable specification, which facilitates efficient code generation and fast simulations. Our framework uniquely combines the advantages of rigorous specifications with accurate power measurements and methods for online model estimation, thus enabling automated design of correct and energy-efficient controllers.

1. INTRODUCTION

Embedded devices are at the core of numerous safety-critical applications in areas such as avionics, automotive and biomedical. One of the main challenges in the design and implementation of embedded devices is to ensure that their behaviour meets design-time requirements while, at the same time, consuming the least amount of energy possible.

These contrasting aspects are typically addressed in two separate phases: requirements are enforced through developing formal models of the system and analysing their correctness using formal verification methods, whereas energy efficiency through selecting low-power hardware components and tuning the physical device to reduce consumption. An established approach to deal with this separation is to employ integrated design and analysis of software and hardware components, called hardware/software co-design, and in particular *hardware-in-the-loop (HIL) optimization*. This involves automated optimisation of hardware based on evaluating its behaviour in interaction with the simulation of the plant model, a method known as *HIL simulation* [3]. Since the device under test works in real-time, effective HIL simulation approaches must enable real-time simulations, as well as synchronisation and data transfer between hardware and software. However, existing HIL optimisation approaches are ad hoc and lack automated support that incorporates formal verification and synthesis methods.

In this paper, we develop a comprehensive and fully automated model-based framework for hardware-in-the-loop energy optimization of embedded devices that, for the first time, integrates rigorous specifications with data-driven energy optimisation and online estimation of power models. At the system-design level, we adopt the widely used MATLAB Stateflow modelling formalism. We support hybrid systems specified as networks of parameterised timed I/O automata and encoded in Stateflow, and employ parameter synthesis methods to restrict the search to the device parameters that guarantee a given safety property. At the HIL optimisation level, we implement a novel method to generate executable code from the Stateflow diagrams by resorting to an intermediate representation as timed Petri nets, which is compact and event-driven, and thus facilitates fast real-time simulations and hardware/software synchronisation. The novelty of our approach is that we derive predictive power consumption models from actual power measurement data, and query these models to find the device parameters that maximise battery lifetime. Our framework is sufficiently general to synthesise energy-efficient embedded software for a variety of applications, which we demonstrate through the evaluation on a temperature controller and a cardiac pacemaker.

1.1 Overview of the framework

The purpose of our framework is twofold. First, we aim to *estimate* detailed power consumption models for enabling the design of energy-efficient embedded software. Second, we aim to *parameterise* the device in order to optimise power consumption and prolong battery life, such that the correct functioning of the device is not compromised. Figure 1 shows the high-level structure of our framework. We consider systems characterised by a *controller* that acts on a *plant*.

At the system design level, the specifications for the plant and controller are given as *timed input/output automata with data (TIOA)* [31]. This formalism (described in Section 2.1) is able to represent networked systems with real-time constraints and discrete control actions (suitable to model the controller), as well as hybrid dynamics through continuous variables and non-linear update functions (to model the plant). Importantly, the framework supports the use of MATLAB Stateflow for specifying TIOA models. At this level, we additionally focus on guaranteeing correctness, which we achieve by excluding the region of controller parameter values that violate a given *safety* requirement. Such parameters describe, for instance, the switching frequency of the device from the active to the idle modes, or the frequency of data capture and processing. This phase is described in Section 2.2 and builds on a *parameter synthesis* method implemented using Satisfiability Modulo Theory (SMT) techniques. In order to optimise the battery lifetime, we also need a *battery model*, given, in our case, as a system of ordinary differential equations (ODEs) (see Section 2.3).

At the HIL optimisation level, HIL simulation has the role of executing the controller code, which is embedded in programmable hardware (e.g. a microcontroller or FPGA), in combination with the plant simulation, running on a computer system. The generation of executable code for HIL simulations from formal TIOA specifications leverages an intermediate representation in terms of timed Petri nets (TPNs), which enables real-time simulations as well as fast event scheduling and hardware-software synchronisation. In Section 3.1, we describe the translation from TIOAs into TPNs, the generation of executable code and the HIL simulation algorithm. The controller unit is attached to a *power monitor* that measures the power consumption of the device. In particular, we are interested in the amount of energy consumed by the controller when performing specific transitions, which is used to build the probabilistic power consumption model, as illustrated in Section 3.2. By taking consumption data from multiple HIL simulations, this step produces a *probabilistic power model* with the same structure as the controller TIOA, but annotated with rewards that describe, for transition t and energy value e, the probability that the device consumes an amount of energy equal to e when performing t. Finally, the optimisation algorithm uses

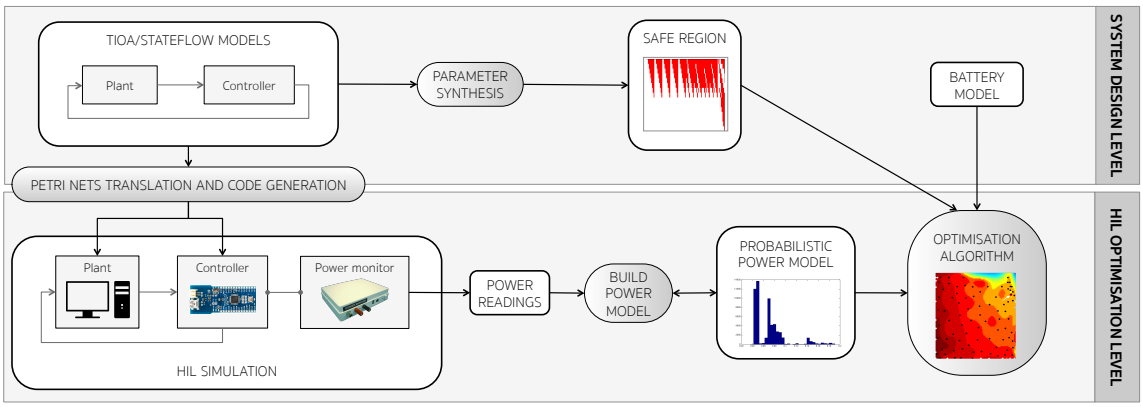


Figure 1: Modelling, hardware-in-the-loop optimisation and power model estimation framework

the battery and consumption models to optimise the battery lifetime. This objective is evaluated by simulating the plant, controller and power consumption models on the computer until the battery runs out. Note that, in this case, a real-time HIL simulation would take an excessive amount of time. Here, we speed up this task through pure software simulation that nevertheless utilises real power readings through the estimated power consumption model. Specifically, we follow a Gaussian process optimisation scheme (see Section 3.3), which has the additional advantage of deriving a statistical model of the objective function with respect to controller parameters. To avoid sampling unsafe parameters, the optimisation algorithm also requires the safe region synthesised at the system design level.

1.2 Related work

In recent years, a number of approaches have been proposed for the design of energy-efficient systems. Benini et.al. [8] provide a comprehensive survey, examining techniques for optimizing energy at different levels: modelling, system design and runtime management. The review by Unsal et.al. [37] focuses on techniques enabling low-power design for real-time systems, covering the whole span of architectural levels, from hardware to operating systems and computer networks. HIL simulation has been successfully employed in a number of industrial applications, including the testing of automotive control systems [22], power electronics [12], avionics [26] and biomedical devices [11, 20, 23]. HIL optimisation approaches are relatively more recent and have been used to optimise, e.g., the performance of wireless networks [33], or the speed of humanoid robots [21]. Despite much research on energy-aware design and combined hardware/software testing, our work is the first to seamlessly integrate, in a fully automated framework based on the widely used Simulink Stateflow notation, rigorous design methods, specifically parameter synthesis, with HIL optimisation and the generation of data-driven power consumption models from real hardware measurements.

In our previous work [7], we introduced a HIL optimisation approach for the energy consumption of cardiac pacemakers, using the Simulink code generation capabilities and evaluating the safety of the pacemaker parameters at HIL simulation time. In contrast, the framework presented in this paper supports general system specifications – even if we also evaluate the pacemaker case study, see Section 4.2 – and provides numerous improvements and novelties, including the derivation of probabilistic consumption models, optimisation of battery lifetime and the formal synthesis of safe controller parameters. Importantly, here we implement a dedicated code generation method in place of the one provided by Simulink. Indeed, the code generated through the latter method is not suitable for accurate energy consumption measurements because it keeps the device running, and thus consuming energy, even when the controller is inactive, e.g. waiting for events from the plant. On the other hand, our code and scheduling algorithm use the power saving modes of the embedded system when the controller is idle. In this way, energy readings consistently reflect the controller activity, at the same time improving energy efficiency of the code. Other approaches exist for generating executable code from timed automata and Petri nets specifications, to mention [2, 34, 5].

The method for synthesising safe parameters has been adapted from our previous work [31]. There, we consider the problem of maximising the robustness of parameters with respect to a given safety property, which we could solve without exploring the full parameter space. In contrast, in this work we have to synthesise the full safe region, and thus cannot exclude any parameter region from the analysis. Parameter synthesis for TIOA models was first considered in [16] and solved by combining parameter sampling and constraint solving. SMT-based verification of timed and hybrid systems has received a lot of attention recently, see e.g. [10]. In [29], the authors present an SMT-based timed system extension to the IC3 algorithm. [28] and [30] respectively develop real-time bounded model checking (BMC) approaches for LTL and CTL. [19] presents an SMT technique to generate inductive invariants for hybrid systems. Sturm et al [36] applies real quantifier elimination tools to synthesise continuous and switched dynamical systems. The dReal solver [17] uses a relaxed notion of satisfiability in order to provide decision procedures for non-linear hybrid systems.

2. SYSTEM DESIGN LEVEL

2.1 Timed I/O automata with data

We consider a set of variables $V = \mathcal{X} \cup \mathcal{D}$, where $\mathcal{X}$ and $\mathcal{D}$ are the set of *clocks* and *data*, respectively. A variable valuation $\eta : V \to \mathbb{R}$ is a function that maps data variables to the reals and clocks to the non-negative reals. We also consider

a set of real-valued *parameters* Γ and valuation functions $\gamma : \Gamma \to \mathbb{R}$. For a set $\mathcal{Y}$, we denote with $\mathcal{V}(\mathcal{Y})$ the set of all valuations over $\mathcal{Y}$. The *update* of a set of variables $V' \subseteq V$ is a real-valued function $r : V' \times \mathcal{V}(V) \times \mathcal{V}(\Gamma) \to \mathbb{R}$. Given valuations η and γ, η is updated by reset r to the valuation $\eta[r] = \{v \mapsto r(v, \eta, \gamma) \mid v \in V'\} \cup \{v \mapsto \eta(v) \mid v \notin V'\}$ that applies the reset r to the variables in V' and leaves the others unchanged. We denote with $\mathcal{R}$ the set of update functions. We consider guard constraints of the form $g = \bigwedge_i v_i \bowtie_i f_i$, where $v_i \in \mathcal{X}$ is a clock, $\bowtie_i \in \{<, \leqslant, >, \geqslant\}$ and f_i is a real-valued function over data variables and parameter valuations. We denote with $\mathcal{B}(V)$ the set of guard constraints over V.

In the following, we introduce our main modelling language, which extends [27] with priorities, data variables and parameters.

Definition 1 (TIOA). *A deterministic timed I/O automaton (TIOA) with priority and data* $\mathcal{A} = (\mathcal{X}, \Gamma, \mathcal{D}, Q, q_0, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \to)$ *consists of:*

- *A finite set of clocks* $\mathcal{X}$*, data variables* $\mathcal{D}$ *and parameters* Γ*.*
- *A finite set of locations* Q*, with initial location* $q_0 \in Q$*.*
- *Finite sets of input* Σ_{in} *and output* Σ_{out} *actions.*
- *A finite set of edges* $\to \subseteq Q \times (\Sigma_{\text{in}} \cup \Sigma_{\text{out}}) \times \mathbb{N} \times \mathcal{B}(V) \times \mathcal{R} \times Q$*. Each edge* $e = (q, a, pr, g, r, q')$ *is described by a source location* q*, an action* a*, a priority* pr*, a guard* g*, an update* r *and a target* q'*.*

TIOAs are able to express hybrid dynamics, since they support data variables and arbitrary functions in the right-hand side of guards and updates. Continuous flows, i.e. the update of variables through differential equations, cannot be expressed directly, but can be modelled with update functions using the explicit solution of the equations. Therefore, TIOAs can express any hybrid automata whose flows admit explicit solutions that can be effectively computed.

We require that priorities define a total ordering of the edges out of any location, and that output actions have higher priority than input actions. To facilitate modular designs, TIOAs are able to synchronise on matching input and output actions, thus forming *networks of communicating automata*. We say that an output edge is *enabled* when the associated guard holds. On the other hand, an input edge is enabled when both its guard holds and it can synchronise with a matching output action fired by another component of the network. Note that, unlike input edges, output edges can fire even without synchronising with a matching input action. A component of a network of TIOAs is enabled if, from its current location, there is at least one outgoing edge enabled. Also, we assume that output edges are *urgent*, meaning that they are taken as soon as they become enabled. As shown in [16], priority and urgency imply that *the TIOA is deterministic.*

Definition 2 (Network of TIOAs). *A network of TIOAs with* m *components is a tuple* $\mathcal{N} = (\{\mathcal{A}^1, \dots, \mathcal{A}^m\}, \mathcal{X}, \Gamma, \mathcal{D}, \Sigma_{\text{in}}, \Sigma_{\text{out}})$ *of TIOAs, where*

- *for* $j = 1, \dots, m$, $\mathcal{A}^j = (\mathcal{X}, \Gamma, \mathcal{D}, Q^j, q_0^j, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \to^j)$ *is a TIOA,*
- $\mathcal{X}$*,* Γ*,* $\mathcal{D}$*,* Σ_{in} *and* Σ_{out} *are the common sets of clocks, parameters, data variables, input and output actions, respectively.*

We define the set of network modes by $\vec{Q} = Q^1 \times \cdots \times Q^m$*, with initial mode* $\vec{q_0} = (q_0^1, \dots, q_0^m)$ *and the initial variable valuation* η_0*. A state of the network is a pair* $(\vec{q}, \eta)$*, where* $\vec{q} \in \vec{Q}$ *is the vector of active locations and* $\eta \in \mathcal{V}(V)$ *is the variable valuation.*

A *parametric network of TIOAs* is a network where the parameter valuation is unknown. $\mathcal{N}(\gamma)$ denotes the concrete network obtained by instantiating the valuation γ.

The execution of a network $\mathcal{N}(\gamma)$ of TIOAs is described by a path $\rho = (\vec{q_0}, \eta_0) \xrightarrow{t_0} (\vec{q_1}, \eta_1) \xrightarrow{t_1} \cdots$, where, for each i, $\rho[i] = (\vec{q_i}, \eta_i)$ is a state of the network and t_i is the time spent in that state. A step in the path occurs as soon as at least one component is enabled. In this case, each enabled component fires the enabled edge with maximum priority, moving to the corresponding target location. The new variable valuation reflects the updates of the fired edges and time spent in the previous state. For a detailed account of the formal semantics of TIOA networks, see [31].

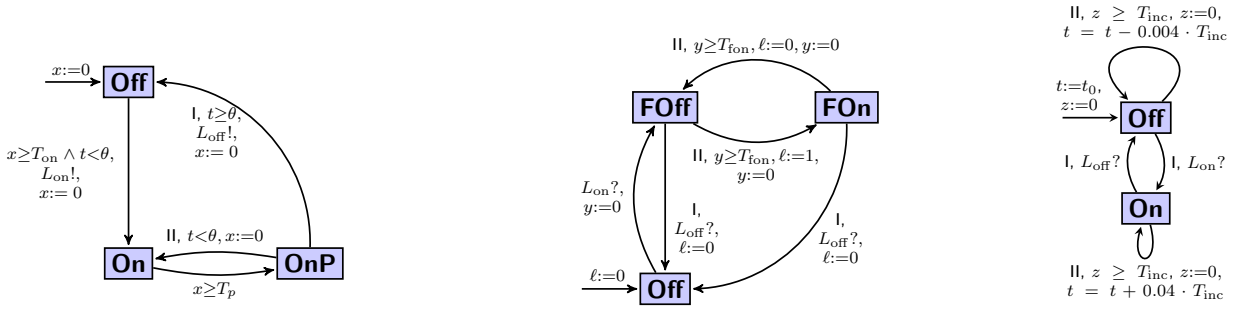
Example 1. *Consider the TIOAs* $\mathcal{A}_1$*,* $\mathcal{A}_2$ *and* $\mathcal{A}_3$ *from Fig. 2. The automata model the bang-bang temperature control system given in [1]. The controller switches the boiler on or off depending on the current temperature (variable* t*) and on a predefined* threshold θ*. Automaton* $\mathcal{A}_1$ *has the role of controlling the boiler, while* $\mathcal{A}_2$ *models the led light that notifies the user when the boiler is heating or switched off. The boiler automaton* $\mathcal{A}_3$ *changes the room temperature. Parameters are:* T_{on}*, which describes the minimum time before the controller switches on the boiler, if the temperature (variable* t*) is below the threshold* θ*;* T_{p}*, which defines the polling period of the temperature sensor; the switching frequency* T_{fon} *for the led, whose state is given by variable* ℓ*; and* T_{inc}*, the time step for updating the temperature in the boiler automaton. Starting from the initial temperature* t_0*,* t *increases by* $0.04°\text{C}$ *per milliseconds when the heater is on, while it decreases by* $0.004°\text{C}$ *when it is off.*

The automata $\mathcal{A}_1$*,* $\mathcal{A}_2$ *and* $\mathcal{A}_3$ *can be composed to form a network. Call this* $\mathcal{N}$*. The states of* $\mathcal{N}$ *are of the form* $((q_{\mathcal{A}_1}, q_{\mathcal{A}_2}, q_{\mathcal{A}_3}), \eta)$ *where, for* $i = 1, 2, 3$*,* $q_{\mathcal{A}_i}$ *is the active location in* $\mathcal{A}_i$ *and* $\eta = (\eta(x), \eta(\ell), \eta(y), \eta(t), \eta(z))$ *is the variable valuation. Network components communicate with each other by means of actions* L_{on} *and* L_{off}*. By instantiating the set of parameters* T_{on}*,* T_{p}*,* θ*,* t_0 *and* T_{inc}*,* $\mathcal{N}$ *induces an execution* ρ*, for instance:*

$$\rho = ((\mathbf{Off}, \mathbf{Off}, \mathbf{Off}), (0, 0, 0, t_0, 0)) \xrightarrow{T_{\text{inc}}}$$
$$((\mathbf{Off}, \mathbf{Off}, \mathbf{Off}), (T_{\text{inc}}, 0, T_{\text{inc}}, t_0 - 0.004 \cdot T_{\text{inc}}, 0)) \xrightarrow{T_{\text{on}} - T_{\text{inc}}}$$
$$((\mathbf{On}, \mathbf{FOff}, \mathbf{On}), (0, 0, 0, t_0 - 0.004 \cdot T_{\text{inc}}, T_{\text{on}} - T_{\text{inc}})) \cdots$$

We now define the model of TIOAs with rewards labelling the transitions, which we will use to describe the power consumption model.

Definition 3 (TIOAs with rewards). *A deterministic timed I/O automaton (TIOA) with rewards is a tuple* $\mathcal{A}_r = (\mathcal{X}, \Gamma, \mathcal{D}, Q, q_0, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \to_r)$*, where* $\mathcal{X}, \Gamma, \mathcal{D}, Q, q_0, \Sigma_{\text{in}}$*, and* Σ_{out} *are defined as in Def. 1 and* $\to_r \subseteq Q \times (\Sigma_{\text{in}} \cup \Sigma_{\text{out}}) \times \mathbb{N} \times \mathcal{B}(V) \times \mathcal{R} \times \mathit{Distr}(\mathbb{Q}_{\geq 0}) \times Q$*.*



(a) Temperature controller, $\mathcal{A}_1$ (b) Led controller, $\mathcal{A}_2$ (c) Boiler (plant), $\mathcal{A}_3$

Figure 2: TIOA network for the bang-bang temperature control example [1]. Roman numbers indicate edge priorities. Symbols "!" and "?" denote output and input actions, respectively.

In the above definition, $Distr(\mathbb{Q}_{\geq 0})$ is the set of all probability distribution functions with finite support over the set of positive rational numbers. For instance, we can associate to an edge $e \in \rightarrow_r$ the function $Prob \in Distr(\mathbb{Q}_{\geq 0})$ that assigns $Prob(23) = \frac{1}{3}$ and $Prob(\frac{1}{4}) = \frac{2}{3}$. We let rewards accumulate over executions of $\mathcal{A}_r$. For example, when the total reward accumulated so far is c and $\mathcal{A}_r$ takes the edge e, c will be increased by 23 with probability $\frac{1}{3}$ and by $\frac{1}{4}$ with probability $\frac{2}{3}$. The definition of network of TIOAs with rewards follows from Definitions 2 and 3.

Encoding in Stateflow. TIOA models can be expressed as MATLAB Stateflow diagrams, which are generally richer than TIOAs. Our framework only supports the TIOA fragment of the Stateflow language, thus excluding features like hierarchical components and continuous flows. However, unlike TIOAs, Stateflow diagrams do not support the definition of arbitrary clocks and clock updates. In particular, each Stateflow component only possesses an implicit clock, which is reset to 0 whenever an edge is taken, and guards are specified through Stateflow temporal operators. Specifically, we use the operator $\mathsf{after(t)}$ in place of the guard $x \geq t$, $\mathsf{before(t)}$ for $x \leq t$ and $\mathsf{at(t)}$ for $x = t$, where x is the implicit clock and t is a time value that can be specified as a function over data variables and parameters.

2.2 Computation of safe region

We compute the set of parameter valuations such that the TIOA network meets a given safety property using a method adapted from [31] and based on satisfiability modulo theory (SMT) solving [14]. For a path ρ of length k, we focus on bounded safety properties of the form $\phi = \bigwedge_{i=0}^{k-1} f_i(\rho[0], \ldots, \rho[i])$, where f_i is a predicate over the states of the network up to position i. Then, for property ϕ, we seek to compute the set $\mathcal{S}$ of safe parameter valuations, defined as $\mathcal{S} = \{\gamma \in \mathcal{V}(\Gamma) \mid \rho(\gamma) \models \phi\}$.

The algorithm for computing the set $\mathcal{S}$ relies on a symbolic SMT-based encoding of the network and the property, and works by exhaustively exploring bounded counter-examples to safety, which amounts to finding valuations such that $\neg\phi$ holds at some point in the path. To ensure decidability, we provide a discrete encoding in the theory of bit-vectors (SMT UF_BV), where non-integer values and non-linear functions are expressed in a sound way through a conservative interval-based abstraction. We remark that the framework can be generalised to support more general path properties, see [31].

The algorithm, given in Alg. 1, extends the SMT-based method for *bounded model checking (BMC)*. We encode an SMT problem, where we incrementally build the set $\bar{S}$ of unsafe parameters by searching for counter-examples (CEs) to safety, which amounts to finding valuations s.t. $\neg\varphi$ holds at some point in the path.

To speed up the computation, the method uses *incremental solving*, so that CEs are computed stepwise, for increasing path lengths, thus exploiting the fact that SMT solvers can use the clauses learned in the previous steps to improve the solution time of the current step. In addition, we include an algorithm for *counter-example generalization* (procedure GeneralizeCE, Alg. 2) that, given a CE, attempts to derive a larger unsafe region that contains the CE.

Algorithm 1: Incremental Synthesis

Input: Parametric network $\mathcal{N}(\cdot)$, safety property ϕ, path length $n \in \mathbb{N}^+$
Output: Unsafe region $\bar{S}$

Function IncrementalSynth($\mathcal{N}(\cdot), \phi, n$)
  $\bar{S} := \bot$
  Assert $Init(\rho[0])$
  for $k = 0, \ldots, n-1$ **do**
    Assert $p_k : \phi_k$
    repeat `/* CE generation cycle */`
      (SAT, γ_{CE}) := Solve $\neg p_k$
      γ'_{CE} := GeneralizeCE(γ_{CE})
      Assert $\neg\gamma'_{CE}$
      $\bar{S} := \bar{S} \vee \gamma'_{CE}$
    until SAT
    if $k < n-1$ **then**
      Assert $T(\rho[k], \rho[k+1])$
  return $\bar{S}$

Algorithm 2: Counter-example generalization

Input: Counter-example γ_{CE}
Output: Generalization γ'_{CE} s.t. $\gamma_{CE} \implies \gamma'_{CE}$

Function GeneralizeCE(γ_{CE})
  Push
  Assert p_k
  forall the $p \in \Gamma$ **do**
    Assert $g_p : p = \gamma_{CE}(p)$
  (SAT, γ) := Solve $\bigwedge_{p \in \Gamma} g_p$
  $\gamma'_{CE} := \bigwedge_{p.g_p \in \mathsf{UnsatCore}} p = \gamma_{CE}(p)$
  Pop
  return γ'_{CE}

In Alg. 1, the *Init* predicate (line 3) is used to constrain the initial automata locations and variable valuation. Command Assert adds in the SMT solver a formula that must hold true. At a generic step k of the path, we first assert the safety property up to k, $\phi_k = \bigwedge_{i=0}^{k-1} f_i(\rho[0], \ldots, \rho[i])$ (line 5). In this case, the assertion is named with a literal p_k, meaning that the satisfaction value of ϕ_k is the same as p_k. During the counter-example generation cycle (lines 6-12), Solve $\neg p_k$ checks if the negated safety is satisfied under the current assertions. If so, the solver returns a model, in our case a counter-example γ_{CE}, which we generalise to γ'_{CE} by calling GeneralizeCE. γ'_{CE} is then excluded from the search space (line 10) and added to $\bar{S}$ (line 11). When no further CEs can be found, we can exit the CE generation loop, assert the transition constraints (line 14) and increase the step to $k+1$. In the algorithm, T indicates the transition predicate between states of the path, i.e. $T(s, s') = \exists t.\ s \xrightarrow{t} s'$. Note that, the discretisation of the parameter space ensures that the cycle at lines 6-12 always terminates.

The GeneralizeCE procedure is executed on top of the solver used in Alg. 1 and exploits the ability of SMT solvers to generate unsatisfiable cores, i.e. when a formula is unsatisfiable under the current assertions, to produce a subset its clauses whose conjunction is still unsatisfiable. Given a CE γ_{CE}, the idea is to derive a larger unsafe region γ'_{CE} that contains γ_{CE}. This is achieved by asserting the safety property (line 3), and the valuation γ_{CE} (line 4). In particular, we associate each assertion $p = \gamma_{CE}(p)$ for $p \in \Gamma$ with a literal g_p. Even if the outcome is clearly negative (we have asserted safety), we need to check the satisfiability of formula $\bigwedge_{p\in\Gamma} g_p$ (line 5) in order to produce an unsat core, i.e. a set UnsatCore $\subseteq \{g_p \mid p \in \Gamma\}$. If UnsatCore is a strict subset of the g_p literals, we say that the generalization is successful, since we obtain a larger region: $\gamma'_{CE} = \{\gamma \mid \gamma(p) = \gamma_{CE}(p)$ if $g_p \in$ UnsatCore$\}$. Otherwise, $\gamma'_{CE} = \gamma_{CE}$. As an example, let $\gamma_{CE} = (p_1 = 3 \wedge p_2 = 5)$, and let g_{p_1} and g_{p_2} be the corresponding literals. If UnsatCore $= \{g_{p_2}\}$, then the generalization $\gamma'_{CE} = (p_2 = 5)$ strictly contains γ_{CE}. In the algorithm, the Pop command removes from the solver all the constraints asserted after the last Push.

2.3 Kinetik battery model

Using a battery model, one can describe the state of the battery over time, which in our framework enables the development of power usage models and the optimisation of battery lifetime. We consider the Kinetik Battery Model, which describes variations of the battery capacity as a function of charge and discharge currents [32]. The model is given by the following system of ODEs:

$$\frac{dy_1(t)}{dt} = -i(t) + k\left(\frac{y_2(t)}{1-c} - \frac{y_1(t)}{c}\right) \qquad (1)$$
$$\frac{dy_2(t)}{dt} = -k\left(\frac{y_2(t)}{1-c} - \frac{y_1(t)}{c}\right)$$

The battery charge is distributed in two wells: available charge $y_1(t)$ and bound charge $y_2(t)$. The function $i(t)$ denotes the current applied to the battery. When the value of $i(t)$ is zero the battery enters the recovery mode, where the energy flows from the bound-charge well to the available-charge well. However, when the current $i(t)$ is not zero, both $y_1(t)$ and $y_2(t)$ decrease over time. If C [Ah] (ampere-hour) is the initial total capacity of the battery then $y_1(0) = c \cdot C$ and $y_2(0) = (1-c) \cdot C$, where c is a fraction of the total capacity. The conduction parameter k represents the flow rate of charge from the bound-charge well to the available-charge well. The battery is considered to be empty when $y_1(t) = 0$. As explained in Section 3.3, integration with the TIOA power consumption model results in a piecewise constant $i(t)$, and thus in a hybrid battery model.

2.4 Timed Petri nets

Petri nets are a well known formalism for modelling the control flow of concurrent systems [15]. They have an intuitive semantics and are ideally suited to generating event based executable code. We use them as an intermediate model in the process of generating code from networks of TIOAs. The main advantage of the Petri net formalism compared to TIOAs is the ability to compute in advance the synchronisation event between different transitions, which is central to the real-time scheduling algorithm for HIL simulation, as we will explain in Section 3.1.

DEFINITION 4 (PETRI NET). *A Petri net or Place/Transition net is a tuple* $\mathcal{O} = (P, T, W^-, W^+, W^0, m_0)$ *with* $P \cap T = \varnothing$ *where*

- P *is a finite set of places.*
- T *is a finite set of transitions.*
- $W^- : P \times T \to \mathbb{N}$ *is the* pre incidence matrix.
- $W^+ : P \times T \to \mathbb{N}$ *is the* post incidence matrix.
- $W^0 : P \times T \to \mathbb{N}$ *is the* inhibitor incidence matrix.
- $m_0 \in \mathbb{N}^P$ *is the initial marking.*

We call a *marking* $m \in \mathbb{N}^P$ of a Petri net $\mathcal{N}$ a vector assigning an integer to each place of the net. A transition $\mathbf{t} \in T$ is enabled in a marking m if $\forall p \in P.\ m(p) - W^-(p, \mathbf{t}) \geq 0 \wedge m(p) - W^0(p, \mathbf{t}) < 0$. The firing of an enabled transition $\mathbf{t}$ from a marking m leads to marking m', denoted $m \xrightarrow{t} m'$, where m' is defined, for each $p \in P$, as $m'(p) = m(p) - W^-(p, \mathbf{t}) + W^+(p, \mathbf{t})$.

Timed behaviours are introduced in Petri nets by adding time constraints on transitions [9]. We denote by $\mathcal{R}^d$ the set of update functions defined as in the case of TIOAs, except that we restrict the functions only to data variables.

DEFINITION 5 (TPN). *A timed Petri net (TPN) with priority and data is a tuple* $\mathcal{O} = (P, T, W^-, W^+, W^0, m_0, \mathcal{D}, \alpha, \beta, Pr, up)$ *with* $P \cap T = \varnothing$ *where*

- $(P, T, W^-, W^+, W^0, m_0)$ *is a Petri net with inhibitor arcs.*
- $\mathcal{D}$ *is a finite set of data variables.*
- $\alpha : T \to (\mathbb{R}^+ \cup \infty)$ *is a function assigning to each transition its earliest firing time.*
- $\beta : T \to (\mathbb{R}^+ \cup \infty)$ *is a function assigning to each transition its latest firing time.*
- $\forall\, \boldsymbol{t} \in T.\ \alpha(\boldsymbol{t}) \leq \beta(\boldsymbol{t})$.
- $Pr : T \to \mathbb{N}$ *is a function assigning to each transition a priority.*
- $up : T \to \mathcal{R}^d$ *mapping transitions to update functions over data variables.*

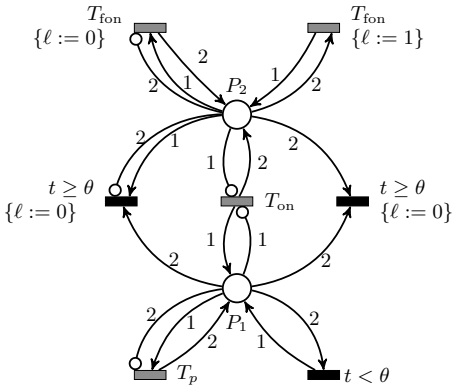


Figure 3: Petri net for the bang-bang controller of Fig. 2. Arcs with arrowheads are input and output arcs. Arcs ending with a circle are inhibitor arcs. Each arc is labelled by its valuation. Transitions in grey are timed; transition in black, instantaneous. Updates of data variables are enclosed in brackets.

An implicit clock is associated with each transition $\mathbf{t}$ of the net. The clock is reset when the transition becomes enabled. The transition is *enabled* when the clock valuation lies between the earliest ($\alpha(\mathbf{t})$) and latest ($\beta(\mathbf{t})$) firing times. A transition can fire only if no transition with higher priority is enabled. In our setting we work with TPNs for which the timed behaviour is *deterministic*, that is, for any transition $\mathbf{t}$, $\alpha(\mathbf{t}) = \beta(\mathbf{t})$ and all transitions have different priorities.

From TIOA networks to TPNs. For our purposes, we only need to consider the subset of TIOAs derived from Stateflow diagrams. As explained in Section 2.1, this corresponds to having, for each TIOA $\mathcal{A}$, exactly one clock $x_{\mathcal{A}}$, which is reset every time an edge is taken.

For each automaton $\mathcal{A}^j = (\mathcal{X}, \Gamma, \mathcal{D}, Q^j, q_0^j, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \rightarrow^j)$ of a network $\mathcal{N}$, we build a TPN $\mathcal{O}_j = (\{p_j\}, T_j, W_j^-, W_j^+, W_j^0, m_{0,j}, \mathcal{D}, \alpha_j, \beta_j, Pr_j, up_j)$. We define an injective function $f_j : Q^j \rightarrow \mathbb{N}$ that allows encoding each location $q \in Q^j$ of the automaton as a marking of the place p_j. For each edge $(q, a, pr, g, r, q') \in \rightarrow^j$ of $\mathcal{A}^j$, a transition $\mathbf{t}$ is added to the TPN $\mathcal{O}_j$. Three arcs are added between p_j and $\mathbf{t}$: an input arc with valuation $f_j(q)$, an inhibitor arc with valuation $f_j(q) + 1$ and an output arc with valuation $f_j(q')$. The update function is defined such that $up_j(\mathbf{t}) = r$. The firing time of the transition is set to the smallest time satisfying the guard g. A temporary labelling function $\Lambda(\mathbf{t}) = a$ keeps the action of $\mathbf{t}$. Pr_j reflects the priorities in the original automaton. The algorithm for composing all TPNs $\mathcal{O}_1, \ldots, \mathcal{O}_j$ obtained from $\mathcal{N}$ into a single TPN $\hat{\mathcal{O}}$ is described in the paragraph below. The resulting TPN is of the same order of magnitude as the input TIOA and, thus is very compact and facilitates efficient code generation. It can be shown that deterministic delays and the structure of W_j^-, W_j^+ and W_j^0 imply that such a derived TPN preserves the determinism of the input TIOA.

Example 2. *In Fig. 3 we depict the composed TPN $\hat{\mathcal{O}}$ obtained from the network of automata $\mathcal{A}_1$ and $\mathcal{A}_2$ from Example 1. Locations of $\mathcal{A}_1$ and $\mathcal{A}_2$ are encoded as markings of the net $\hat{\mathcal{O}}$ as follows. Place P_1 encodes the locations of $\mathcal{A}_1$ with the function $f_1 = \{\mathbf{Off} \mapsto 0, \mathbf{On} \mapsto 1, \mathbf{OnP} \mapsto 2\}$ that maps each locations of the automaton to the corresponding number of tokens. The function $f_2 = \{\mathbf{Off} \mapsto 0, \mathbf{FOn} \mapsto 1, \mathbf{FOff} \mapsto 2\}$ encodes the locations of $\mathcal{A}_2$ in P_2. Automata edges that do not synchronise with other components are translated into transitions that are connected to only one place (visible at the top and bottom of Fig. 3). For example, the edge from* **On** *to* **OnP** *in $\mathcal{A}_1$ corresponds to the TPN transition on the bottom-left corner. Given the weights on the connected arcs, this transition can be fired when P_1 contains exactly 1 token and, after firing, puts 2 tokens back in place P_1. Each possible synchronisation between automata edges is translated into a single TPN transition, which is connected to all places encoding for the automata involved in the synchronisation. An example is the transition in the centre of the figure between P_1 and P_2. Edges with a time guard are translated into timed transitions in $\hat{\mathcal{O}}$ (depicted in grey and labelled with the delay). Edges without time guards are translated into immediate transitions (depicted in black).*

TPN composition algorithm. First, a number of constraints have to be imposed on each TIOA:

1. For all edges $(q, a, pr, g, r, q') \in \rightarrow$, if $a \in \Sigma_{\text{in}}$, then $q \neq q'$. Any automaton can be transformed to satisfy this property by adding a new state q'' and an immediate edge, such that (q, a, pr, g, r, q) is replaced by two edges, (q, a, pr, g, r, q'') and $(q'', \varepsilon, pr, \top, r, q)$.

2. For all $a \in \Sigma_{\text{in}}$, for all $q \in Q$, for all valuations $\eta \in \mathcal{V}(V)$ and $\gamma \in \mathcal{V}(\Gamma)$, and for all time values $x \in \mathbb{R}^+$, there exist pr, g, r, q' such that $(q, a, pr, g, r, q') \in \rightarrow$ and g holds under valuations $\eta + x$ and γ, where $\eta + x$ corresponds to the variable valuation η after time x has elapsed. Again, any automaton can be adapted to satisfy this property by adding loops.

Some additional definitions for Petri nets are required to describe the composition algorithm. Let $\sigma = \sigma_1 \cdots \sigma_n \in T^*$ be a sequence of TPN transitions. We say that σ is enabled from a marking m leading to m', denoted by $m \xrightarrow{\sigma} m'$, if there exists a sequence of markings $m = m_1 \cdots m_{n+1} = m'$ such that for all $1 \leq k \leq n$, $m_k \xrightarrow{\sigma_k} m_{k+1}$. We denote by $Reach(\mathcal{O}, m_0)$ the set of reachable markings, corresponding to $\{m \mid \exists \sigma \in T^* \text{ s.t. } m_0 \xrightarrow{\sigma} m\}$. This set is in general infinite. For a transition $\mathbf{t} \in T$ we denote by ${}^\bullet t$ the *preset* of transition t defined as $\{p \in P \mid W^-(p, \mathbf{t}) > 0\}$. Similarly, we denote by $\mathbf{t}^\bullet$ the *postset* of $\mathbf{t}$ defined as $\{p \in P \mid W^+(p, \mathbf{t}) > 0\}$.

We now provide the algorithm for composing the set of TPNs $\{\mathcal{O}_1, \ldots, \mathcal{O}_m\}$ obtained through the translation of the TIOA network components. Recall that all such TPNs are disjoint. Therefore, we can equivalently consider as input the union TPN $\mathcal{O} = (P, T, W^-, W^+, W^0, m_0, \mathcal{D}, \alpha, \beta, Pr, up) = \cup_{j=1}^m \mathcal{O}_j$. The algorithm consists of the following steps:

1. Choose a place $p \in P$ and an action a.

2. Compute the set of transitions

$$T_r = \{\mathbf{t}_r \mid {}^\bullet\mathbf{t}_r = \{p\} \text{ and } \Lambda(\mathbf{t}_{ri}) = ?a\}.$$

3. Compute the set of transitions

$$T_w = \{\mathbf{t}_w \mid p \notin {}^\bullet\mathbf{t}_w \text{ and } \Lambda(\mathbf{t}_{ri}) = !a\}.$$

4. For all $\mathbf{t}_r \in T_r$ and $\mathbf{t}_w \in T_w$, add to $\mathcal{O}$ a new transition $\mathbf{t}$ with $\Lambda(\mathbf{t}) = !a$ and for all $q \in P$, update $\mathcal{O}$ as follows:

$$W^-(q,\mathbf{t}) = max(W^-(q,\mathbf{t}_r), W^-(q,\mathbf{t}_w)),$$
$$W^0(q,\mathbf{t}) = max(W^0(q,\mathbf{t}_r), W^0(q,\mathbf{t}_w)),$$
$$W^+(q,\mathbf{t}) = max(W^+(q,\mathbf{t}_r), W^+(q,\mathbf{t}_w)),$$
$$Pr(\mathbf{t}) = Pr(\mathbf{t}_r).$$

5. Remove from $\mathcal{O}$ all transitions in T_r and T_w.

The above steps are iterated until step (2) returns an empty set for any place p and action a.

To obtain a proper TPN, it remains to map the priority function to natural numbers. Let $f : \mathcal{P}(\mathbb{N}) \to (\mathbb{N})^*$ be a function that maps a finite subset of $\mathbb{N}$ to a finite word where the letters are elements of $\mathbb{N}$. For a set $E \subseteq \mathbb{N}$, $f(E) = w$ implies that $|w| = |E|$, $\forall i \leq |w|.w_i \in E$ and $\forall i, j \leq |w|.\ i < j \Rightarrow w_i > w_j$. We now define an order on $\mathcal{P}(\mathbb{N})$, corresponding to the lexical order over $f(\mathcal{P}(\mathbb{N}))$. From the function $Pr : T \to \mathcal{P}(\mathbb{N})$ a new function $Pr' : T \to \mathbb{N}$ is built by using a bijection from the finite part of $\mathcal{P}(\mathbb{N})$ to $\mathbb{N}$ that preserves the order. The resulting TPN with priority function Pr' implements the semantics of the TIOA network.

3. HIL OPTIMISATION LEVEL

3.1 Code generation for HIL simulation

In this step, we generate executable C code from the TPN translation of the TIOA network. Importantly, the generated code facilitates cross-platform deployment, since it uses the same simulation and event scheduling algorithm for the plant and the controller, which are run on two separate hardware platforms. In this way, the HIL simulation algorithm can execute the plant code and the controller code as if they were two TIOAs in a network, except that the actions are sent and received using hardware communication protocols (serial, Ethernet, Bluetooth, etc). The only platform-dependent aspects are related to the functions used for handling simulation time and data communication, which may vary depending on the target architecture.

We implemented the code generation procedures as an extension of the Cosmos tool [4], which already provides features for encoding TPN models into C code.

The executable code consists of two main parts, responsible for encoding the structure of the TPN and for simulating the execution of the TPN, respectively. The main challenge is scheduling the execution of the next transition/event in a very short amount of time, which is crucial to ensure real-time HIL simulations. Importantly, by analysing the structure of the TPN, we can pre-compute both the maximum number n of transitions enabled for scheduling and, for each transition $\mathbf{t}$, the set of transitions that might become enabled or disabled after firing $\mathbf{t}$. This significantly increases the speed of simulation because it restricts the number of transitions to test, thus enabling fast HIL simulations.

The algorithm for simulating a TPN relies on a heap data structure to store the scheduled events. This structure guarantees the removal and insertion in $O(\log(n))$. Each event is of the form $e = (\mathsf{id}, \mathsf{time}, \mathsf{transition})$, where id is the identifier of e, time is the firing time and $\mathsf{transition}$ the corresponding TPN transition. The memory consumption of the heap is constant and equals $n \cdot (2s_{id} + s_{float})$ bits, where s_{id} is the number of bits required to store the identifier of an event, and s_{float} is the number of bits required to store the floating point representation of the firing time. The code for simulation and scheduling is presented in Algorithm 3. The main functions used by the algorithm are as follows:

initialEventHeap() - computes the event heap in the initial state of the simulator.

realTime() - returns the current simulation time.

sleep(t) - enters the idle mode and waits for t milliseconds. If an event is received, the wait period is interrupted.

IsDataAvailable() - returns true if there is new data on the hardware communication link. The data transferred between plant and controller correspond to the identifiers of the fired transitions.

readData() - receives and reads data, and updates the state of the TPN accordingly.

fire(**t**) - fires transition **t**, updating both the marking of the net and the value of the variables related to **t**.

update(EQ) - updates the event queue according to the new state. A naive implementation of this function is to examine each transition **t** and check if it is enabled; if so, add **t** to the event queue together with the minimal time for which **t** can fire. Our implementation is more involved, but much faster, because it exploits the precomputed information about the enabled transitions, as discussed above.

Algorithm 3: HIL Simulation Algorithm

```
Function Simulate()
    EQ := initialEventHeap()
    ctime := realTime()
    while EQ ≠ ∅ do
        e := min(EQ)
        if e.time > ctime then
            sleep(e.time - ctime)
            ctime := RealTime()
            if IsDataAvailable() then
                readData()
                update(EQ)
        else
            ctime := e.time
            fire(e.transition)
            update(EQ)
```

The algorithm iterates over the elements of the heap and picks the transitions with the minimal absolute firing time. If this is greater than the current simulation time (ctime), the algorithm stays in the idle mode for time e.time − ctime. If there is an event available, which is tested through function IsDataAvailable(), the algorithm wakes up. We remark that, at soon as an event is received, the algorithm exits the sleep mode and processes the event, which implies a deterministic waiting time. When the sleep function is called by the plant, we just pause its execution. When it is called by the controller, the hardware platform that embeds the controller code enters the power saving mode. In this way, the algorithm optimises the power consumption of the device by changing its power states only when transitions are enabled.

3.2 Power model builder

The controller unit is attached to a power monitor device that measures the consumption of the unit in order to build the probabilistic power model for the controller. This process takes in the readings from the power monitor,

corresponding to measurements of the electric current consumed by the controller at each instant of time. Note that we can equate power and current consumption in our setting, because the voltage applied by the power monitor to the hardware is constant, and thus power consumption is proportional to current consumption.

As output, the process produces a TIOA network with the same structure as the controller network, but annotated with rewards that characterise the consumption of each transition. Given that the actual power consumption depends on many physical parameters that we cannot control or model, we choose to construct a probabilistic model through multiple executions of the HIL simulator. In this way, each controller transition is mapped to a probability distribution (see Definition 3) which is built from the power measurements recorded for that transition. When, instead, the controller is idle, we assign a constant current in the model, taken as the average of the readings obtained in the idle mode.

3.3 Optimisation of battery lifetime

We aim to find the controller parameters that maximise the expected battery lifetime, that is, the time T at which the available charge is depleted in the battery model: $y_1(T) = 0$ (see Eq. (1)).

As explained in Section 1.1, the objective function is evaluated by simulating the plant and the controller on the computer until the battery runs out. With a sufficiently large number of simulations, this provides a good estimate of the expected battery lifetime under the current parameters. Clearly, this step cannot be performed through HIL simulation (real-time) due to the excessive time needed to deplete the battery. To incorporate real measurements in the simulation, we integrate the battery model with the power consumption model as follows. For each transition performed by the controller during a simulation, we sample an electrical current value r from the corresponding probability distribution in the power consumption model. Then, r is used to update the current function $i(t)$ applied to the battery model (see Eq. (1)). Therefore, $i(t)$ is a piecewise constant function. Finally, the battery lifetime T is computed by deriving the analytical solution for $y_1(t)$ at each sub-domain of $i(t)$.

The objective function T is maximized with respect to the set $\mathcal{S}$ of safe parameters (computed as per Section 2.2) by running the optimisation algorithm described below.

Optimisation algorithm. We use a black-box optimisation method known as *Gaussian process optimisation (GPO)*. The main advantage of GPO is that, together with a sub-optimal solution to the optimisation problem, it provides a statistical model of the (unknown) response function f, corresponding in our case to the battery lifetime T. As new samples are evaluated, these are used to improve *online* the accuracy of the statistical model, which, in turn, is queried to draw new samples. The algorithm is based on [25] and consists of the following steps:

i) select n initial samples (by e.g. Latin hypercube) from $\mathcal{S}$ and compute their objective values; ii) estimate a statistical model from the current samples;

iii) use the model to predict the point $x^* \in \mathcal{S}$ that maximises the expected improvement and obtain the objective value $f(x^*)$;

iv) add $(x^*, f(x^*))$ to the set of samples and go to step ii).

The algorithm terminates after performing steps ii-iv) for a given number of iterations. The statistical model is built following the *Gaussian process regression (GPR)* method [35], which can be seen as a stochastic generalisation of classical regression. Given n samples $x_1, \ldots, x_n$ and their respective objective function values $f(x_1), \ldots, f(x_n)$, the method assumes that they are drawn from a model of the form:

$$f(x_i) = \vec{g}(x_i)^T \cdot \vec{\beta} \; + \epsilon(x_i) \quad i = 1, 2, \ldots, n \tag{2}$$

$\vec{g}(x_i)^T \cdot \vec{\beta}$ is called the *regression part*, where $\vec{g}(x_i)$ is the vector of basis functions and $\vec{\beta}$ is the vector of unknown coefficients estimated through classical regression techniques. ϵ is normally distributed with zero mean and correlation dependent on a weighted Euclidean distance of the n samples. Such weights are the parameters of the statistical model, and are estimated by maximising the likelihood function. For a point x^*, GPR is able to predict both an approximate value for $f(x^*)$, assuming it is randomly distributed according to Eq. (2), and an estimate of the prediction standard error.

4. THE SETUP AND CASE STUDIES

The hardware setup for the case studies consists of three main components: a desktop computer, Arduino board containing AVR microcontroller, and Monsoon$^{\text{TM}}$ power monitor. Figure 4 depicts a schematic diagram of our setup. The controller model runs on the AVR board and communicates with the computer through a USB to serial converter. The USB link between the computer and the USB to serial converter has been altered to ensure that no energy flows through it. We use two transistors and two resistors controlled by the microcontroller to simulate the energy consumed by the controller unit. For instance, in the pacemaker case study (see Section 4.2), these act as the leads of the pacemaker. The power monitor supplies the circuit with a voltage while monitoring power consumption.

For the software setup, we use Stateflow to specify TIOA models, the Z3 theorem prover [13] to compute the safe region, the MATLAB-based SUMO toolbox [18] to perform GPO, and the Cosmos tool [4] to generate code and perform simulations. Specifically, we implemented conversion procedures from Stateflow diagrams to TPNs and extended Cosmos in order to generate code for the AVR microcontroller. Finally, we have consolidated and integrated each software component into a single script that realises the full HIL loop. Stateflow diagrams and corresponding Petri net translations are available at `http://www.veriware.org/heart_pm_methods.php#HIL`.

4.1 Temperature Controller

In our experimental evaluation, we aim to find values for parameters T_{on} and T_{p} that maximise the battery lifetime. Figure 7 summarises the results of the safe region computation and optimisation. We set the temperature threshold θ to 24°C and define a safety property ensuring that the room temperature is always within 23.6°C and 24.4°C.

In the computation of the safe region (depicted in Fig. 7a), we consider $T_{\text{on}} \in [1, 200]$ ms, $T_{\text{p}} \in [1, 200]$ ms and a path length of 50. However, we find safe parameters only in the region $T_{\text{on}} \in [1, 110]$ ms and $T_{\text{p}} \in [1, 20]$ ms, which

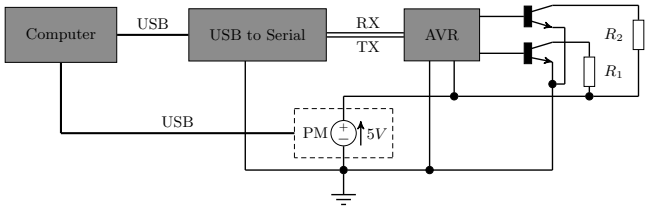
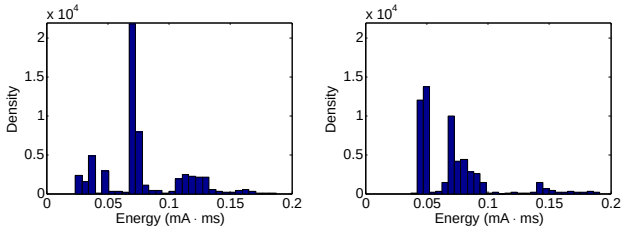


Figure 4: Electrical circuit of the hardware setup. AVR denotes an Arduino Uno board featuring an AVR ATmega328 microcontroller.



(a) **Off → On** (b) **On → OnP**

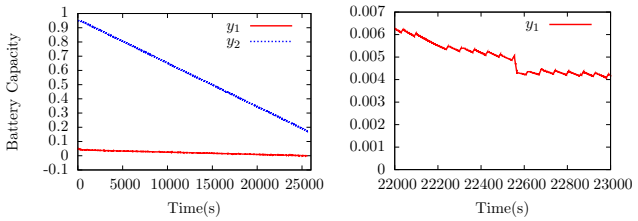
Figure 5: Power model for the temperature controller. Plots show the distributions over the energy consumed by controller actions for: switching the boiler on (a) and polling the sensor (b).

we use as the parameter space in the optimisation loop. We also observe that, with $T_{\mathrm{p}} > 11$ ms, the property cannot be guaranteed in a robust way, that is, several unsafe parameter values exist. This suggests that a relatively high polling frequency is necessary to keep the temperature within tight bounds.

Figure 5 shows the synthesised power model for the temperature controller. The model associates controller's actions with the discrete probability distributions over the energy measurements. In particular, we report the electric charge (measured in mA·ms) that, under fixed voltage, is proportional to the electric energy. As regards the action of switching the boiler on (Fig. 5a), the most likely energy value is around 0.07 mA·ms, whereas for the action of polling the sensor (Fig. 5b) it is around 0.05 mA·ms. In general, we observe that the shapes of the two histograms cannot by adequately approximated with an analytical distribution (e.g. Gaussian), which supports our choice of using discrete distributions for the power model.

Figure 6 illustrates the evolution of the battery capacity along a simulation trace, up to the point where the available charge is depleted. In Figure 6b, we zoom into a shorter time window, to show the detailed evolution of the available capacity. The effects of the periodic polling of the sensor are clearly visible, as well as the activation of the heater in the middle of the graph.

In this case study, we employ 5000 simulations for each evaluation of the expected battery lifetime, that is, the function we seek to optimise. The optimisation algorithm returns a statistical model of the objective function (Figures 7b and 7c) together with the maximising parameters, in this case being: $T_p = 16$ ms and $T_{on} = 104$ ms. However, we notice that all the simulated parameters (black dots in Fig. 7b) yield similar objective function values. This results in "almost uniform" sampling strategy by the GPO algorithm, with a slightly higher number of sampled parameters around



(a) (b)

Figure 6: (a) Evolution of battery capacity during one simulation. y_1 (red) is the available charge. y_2 (blue) is the bound charge. (b) Zoom of the available capacity.

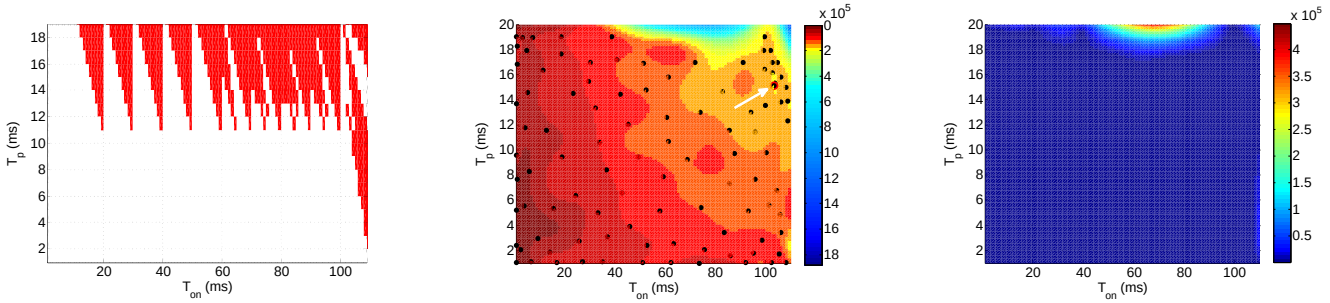
the optimal one. Thus, the standard deviation (SD) of the estimation is "almost constant" (see Fig. 7c). Due to the regression algorithm, SD values tend to decrease in proximity of sampled parameters. Indeed, we detect higher SD values only for high T_{p}, where many parameters are unsafe and thus excluded from the search. The analysis of the standard deviation is crucial because we generally aim to derive parameters that not just yield (sub-)optimal mean values, but also with low uncertainty.

4.2 Heart and Pacemaker

To demonstrate the versatility of our framework, we consider a system composed of a cardiac pacemaker (the controller) and the heart (the plant). The heart model is used to reproduce the propagation of the cardiac action potential from the atrium to the ventricle. The pacemaker has the role of maintaining the synchronisation between the two chambers, by delivering impulses that establish correct heart rhythm. We consider the TIOA model for the heart recently presented in [6]. The pacemaker model is a TIOA adaptation of the model in [24]. The heart-pacemaker TIOA network comprises 14 components, 38 locations and 75 edges, leading to more than 1 billion reachable states. The TPN translation resulted in 21 places, 95 transitions and 427 arcs, which is less than an order of magnitude larger than the automaton and, thus provides evidence of a very efficient representation.

We consider the following two parameters: TAVI, which mimics the conduction time from the atrium to the ventricle, and TURI, which sets an upper bound on the heart rate. In particular, TURI is the amount of time that the pacemaker waits before pacing the ventricle, after an impulse from the atrium has occurred and TAVI elapsed. The nominal values as suggested by pacemaker manufacturers are TAVI = 150 ms and TURI = 500. The safe region (Fig. 8a) is constructed by considering TAVI $\in [1, 2000]$ ms, TURI $\in [1, 2000]$, a path length of 25 and a safety property requiring that the time between two consecutive beats in the ventricle is always within the range $[500, 1000]$ ms, which implies a heart rate between 60 and 120 BPM. The algorithm is able to find safe parameters only in the interval TAVI $\in [1, 620]$ ms and TURI $\in [1, 1000]$ ms, used as the search space in the optimisation algorithm.

In the first experiment, we aim to minimise the total electric current during 1 minute of HIL simulation. In this alternative configuration of our framework, we bypass the construction of the power model and feed the measurements directly into the optimisation algorithm, which, in turn, sends the parameters to evaluate to the HIL simulator. Results in-



(a) Safe region (b) Mean (c) SD

Figure 7: Battery lifetime optimisation results for the temperature controller. Parameters are T_{on} (x-axis) and T_p (y-axis). (a) depicts the safe (white) and the unsafe (red) parameters. (b) The heat map shows the mean values of the estimated Gaussian process (red: short battery life, blue: long battery life). In this case, the colour scale is not linear. Black dots indicate the simulated samples, the red dot the optimal sample (indicated also by a white arrow). (c) Standard deviation (blue: low, red: high). Max expected battery lifetime: 177193 ms at $T_p = 16$ ms and $T_{on} = 104$ ms (white arrow in plot b).

dicate that the optimal parameters overestimate the nominal ones and are obtained at TAVI = 208 ms and TURI = 778 ms, but we can achieve similar consumption values for most of the simulated points (see Fig. 8b). In the region approximately given by TAVI $\in [300, 600]$ ms and TURI $\in [600, 800]$ ms, we observe much higher consumption values. However, these are just estimations by the GP regression algorithm, since no parameters are actually simulated in this region. High standard deviation values are registered in this region (see Fig. 8c) and, more prominently, in the unsafe half-plane of the parameter space. As explained in Section 4.1, unsafe parameters are excluded from the optimisation, resulting in high uncertainty in their statistical estimation.

Second, we maximise the expected battery lifetime, thus running the default HIL optimisation loop. Results are reported in Fig. 8d and 8e. In this case, the optimal parameters underestimate the nominal ones and are: TAVI = 64 ms and TURI = 205 ms. The other simulated parameters give comparable objective function values (all above 89% of the optimal lifetime), except for the parameters TAVI = 207 ms and TURI = 382 ms, which yield a lifetime 44% lower than the optimal. This can be ascribed to the probabilistic nature of the power model. By regression, we register a red area (low lifetime) around this point (plot 8d), and a high standard deviation in its surroundings (plot 8e).

We highlight the strengths of our approach, which is not limited to the evaluation of the single optimal parameter valuation, but also allows for a detailed analysis of the whole parameter space, in terms of correctness (through parameter synthesis) and energy efficiency (through the statistical estimation by GPO). These aspects are crucial for the design of safety-critical systems like cardiac pacemakers.

5. CONCLUSION

We presented a framework for optimising energy consumption of embedded software and estimating consumption models from measurement data, which supports hybrid systems specified as parametric timed automata with data, and encoded in MATLAB Stateflow. We implemented this framework through a fully automated workflow, employing a wide range of methods, including SMT-based parameter synthesis, embedded code generation from Petri nets, real energy measurements and Gaussian process optimisation. Our approach is the first to integrate HIL optimisation with rigorous design methods, parameter synthesis and online power model construction, thus providing, in addition to the optimised parameters, a wealth of information on the design space of the system. The evaluation on the temperature controller and cardiac pacemaker case studies demonstrates the versatility and effectiveness of the approach, which ultimately enables the synthesis of embedded controllers that are both *correct-by-design* and *energy-efficient-by-design* for a wide variety of embedded systems. As future work, we aim at extending our framework to support probabilistic plant models and the synthesis of controller code and components.

Acknowledgements. This work is supported by the ERC AdG VERIWARE, ERC PoC VERIPACE and the Institute for the Future of Computing, Oxford Martin School.

6. REFERENCES

[1] Bang-bang control using temporal logic. `http://tinyurl.com/ngp4epu`.

[2] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. *Nord. J. Comput.*, 9(4):269–300, 2002.

[3] M. Bacic. On hardware-in-the-loop simulation. In *CDC-ECC*, pages 3194–3198. IEEE, 2005.

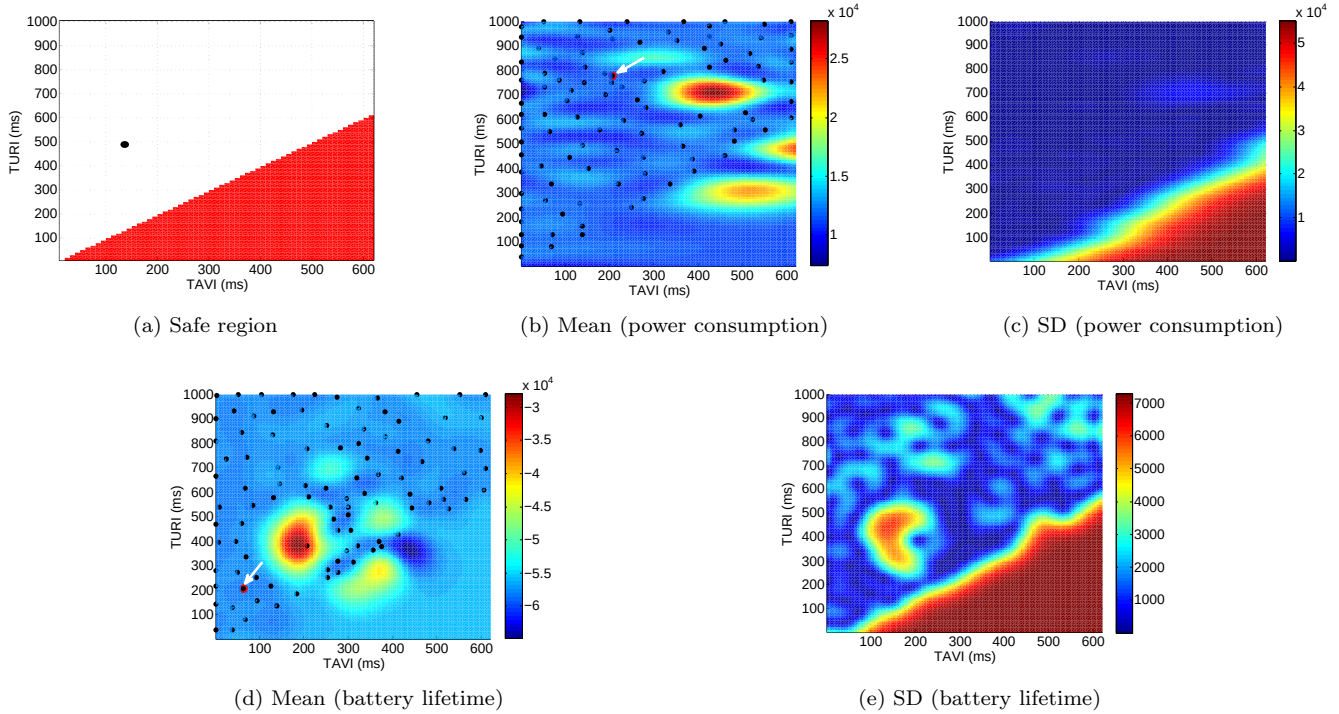
[4] P. Ballarini, B. Barbot, M. Duflot, S. Haddad, and N. Pekergin. HASL: A new approach for performance evaluation and model checking from concepts to experimentation. *Performance Evaluation*, 2015.

[5] V. Bandur, W. Kahl, and A. Wassyng. Microcontroller assembly synthesis from timed automaton task specifications. In *Formal Methods for Industrial Critical Systems*, pages 63–77. Springer, 2012.

[6] B. Barbot et al. Estimation and verification of hybrid heart models for personalised medical and wearable devices. In *CMSB*, pages 3–7, 2015.

[7] C. Barker et al. Hardware-in-the-loop simulation and energy optimization of cardiac pacemakers. In *EMBC*, pages 7188–7191. IEEE, 2015.

[8] L. Benini and G. d. Micheli. System-level power optimization: techniques and tools. *ACM TODAES*, 5(2):115–192, 2000.



(a) Safe region (b) Mean (power consumption) (c) SD (power consumption)

(d) Mean (battery lifetime) (e) SD (battery lifetime)

Figure 8: Optimisation of the total power consumption (b,c) and battery lifetime (d,e) of the pacemaker. Legend is as in Fig. 7. The black dot in plot (a) indicates the default pacemaker parameters. (b) Mean value of the Gaussian process for min current: 11803 mA at TAVI = 208 ms and TURI = 778 ms (red: high consumption, blue: low consumption). (d) Max expected battery lifetime: 58303 ms at TAVI=64 ms and TURI=205 ms.

[9] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE TSE*, 17(3):259–273, 1991.

[10] A. Cimatti, S. Mover, and S. Tonetta. Smt-based verification of hybrid systems. In J. Hoffmann and B. Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press, 2012.

[11] E. Dassau et al. In silico evaluation platform for artificial pancreatic β-cell development-a dynamic simulator for closed-loop control with hardware-in-the-loop. *Diabetes technology & therapeutics*, 11(3):187–194, 2009.

[12] E. de Jong et al. European white book on real-time power hardware in the loop testing: Derlab report no. r-005.0. 2012.

[13] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer, 2008.

[14] L. De Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *SBMF*, volume 5902 of *LNCS*, pages 23–36. Springer, 2009.

[15] M. Diaz. *Petri Nets: Fundamental models, verification and applications*. Wiley, 2010.

[16] M. Diciolla, C. H. P. Kim, M. Kwiatkowska, and A. Mereacre. Synthesising Optimal Timing Delays for Timed I/O Automata. In *EMSOFT*. ACM, 2014.

[17] S. Gao, S. Kong, and E. M. Clarke. Satisfiability modulo ODEs. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 105–112. IEEE, 2013.

[18] D. Gorissen et al. A surrogate modeling and adaptive sampling toolbox for computer based design. *Journal of Machine Learning Research*, 11:2051–2055, 2010.

[19] S. Gulwani and A. Tiwari. Constraint-based approach for analysis of hybrid systems. In *Computer Aided Verification*, pages 190–203. Springer, 2008.

[20] B. Hanson, M. Levesley, K. Watterson, and P. Walker. Hardware-in-the-loop-simulation of the cardiovascular system, with assist device testing application. *Medical engineering & physics*, 29(3):367–374, 2007.

[21] T. Hemker et al. Hardware-in-the-loop optimization of the walking speed of a humanoid robot. In *Proc. of CLAWAR*, 2006.

[22] R. Isermann et al. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *Control Engineering Practice*, 7(5):643–653, 1999.

[23] Z. Jiang et al. Cyber–physical modeling of implantable cardiac medical devices. *Proc. of the IEEE*, 100(1):122–137, 2012.

[24] Z. Jiang et al. Modeling and verification of a dual chamber implantable pacemaker. In *TACAS*, pages 188–203. Springer, 2012.

[25] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[26] D. Jung and P. Tsiotras. Modeling and hardware-in-the-loop simulation for a small unmanned aerial vehicle. *AIAA Infotech at Aerospace, AIAA*,

pages 07–2763, 2007.

[27] D. Kaynar et al. The theory of timed I/O automata. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–137, 2010.

[28] R. Kindermann, T. Junttila, and I. Niemelä. Beyond lassos: Complete SMT-based bounded model checking for timed automata. In *Formal Techniques for Distributed Systems*, pages 84–100. Springer, 2012.

[29] R. Kindermann, T. Junttila, and I. Niemelä. Smt-based induction methods for timed systems. In *Formal Modeling and Analysis of Timed Systems*, pages 171–187. Springer, 2012.

[30] M. Knapik and W. Penczek. Bounded model checking for parametric timed automata. In *Transactions on Petri Nets and Other Models of Concurrency V*, pages 141–159. Springer, 2012.

[31] M. Kwiatkowska, A. Mereacre, N. Paoletti, and A. Patanè. Synthesising robust and optimal parameters for cardiac pacemakers using symbolic and evolutionary computation techniques. In *HSB*, 2015.

[32] J. F. Manwell and J. G. McGowan. Lead acid battery storage model for hybrid energy systems. *Solar Energy*, 50(5):399–405, 1993.

[33] M. Mehari et al. Efficient multi-objective optimization of wireless network problems on wireless testbeds. In *CNSM*, pages 212–217. IEEE, 2014.

[34] S. Philippi. Automatic code generation from high-level petri-nets for model driven systems engineering. *Journal of Systems and Software*, 79(10), 2006.

[35] C. E. Rasmussen. *Gaussian processes for machine learning*. MIT press, 2006.

[36] T. Sturm and A. Tiwari. Verification and synthesis using real quantifier elimination. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, pages 329–336. ACM, 2011.

[37] O. S. Unsal and I. Koren. System-level power-aware design techniques in real-time systems. *Proc. of the IEEE*, 91(7):1055–1069, 2003.