A Hoare Logic for Energy Consumption Analysis[⋆]

Rody Kersten[1], Paolo Parisen Toldin[2], Bernard van Gastel[1,3], and Marko van Eekelen[1,3]

[1] Radboud University Nijmegen, Institute for Computing and Information Sciences, The Netherlands
{r.kersten,m.vaneekelen}@cs.ru.nl

[2] University of Bologna, Department Computer Science and Engineering, Italy
parisent@cs.unibo.it

[3] Open University of the Netherlands, Faculty of Management, Science and Technology, The Netherlands
{bernard.vangastel,marko.vaneekelen}@ou.nl

Abstract. Energy inefficient software implementations may cause battery drain for small systems and high energy costs for large systems. Dynamic energy analysis is often applied to mitigate these issues. However, this is often hardware-specific and requires repetitive measurements using special equipment.
We present a static analysis deriving upper-bounds for energy consumption based on an introduced energy-aware Hoare logic. Software is considered together with models of the hardware it controls. The Hoare logic is parametric with respect to the hardware. Energy models of hardware components can be specified separately from the logic. Parametrised with one or more of such component models, the analysis can statically produce a sound (over-approximated) upper-bound for the energy-usage of the hardware controlled by the software.

1 Introduction

Power consumption and green computing are nowadays important topics in IT. From small systems such as wireless sensor nodes, cell-phones and embedded devices to big architectures such as data centers, mainframes and servers, energy consumption is an important factor. Small devices are often powered by a battery, which should last as long as possible. For larger devices, the problem lies mostly with the costs of powering the device. These costs are often amplified by inefficient power-supplies and cooling of the system.

Obviously, power consumption depends not only on hardware, but also on the software *controlling* the hardware. Currently, most of the methods available to programmers to analyse energy consumption caused by software use dynamic analysis: measuring while the software is running. Power consumption measurement of a system and especially of its individual components is not a trivial task.

[⋆] This work is partially financed by the IOP GenCom GoGreen project, sponsored by the Dutch Ministry of Economic Affairs, Agriculture and Innovation.

A designated measuring set-up is required. This means that most programmers currently have no idea how much energy their software consumes. A static analysis of energy consumption would be a big improvement, potentially leading to more energy-efficient software. Such a static analysis is presented in this paper.

Since the software interacts with multiple components (software and hardware), energy consumption analysis needs to incorporate different kinds of analysis. Power consumption may depend on hardware state, values of variables and bounds on the required number of clock-cycles.

Related Work There is a large body of work on energy-efficiency of software. Most papers approach the problem on a high level, defining programming and design patterns for writing energy-efficient code (see e.g. [1,2,3]). In [4], a modular design for energy-aware software is presented that is based on a series of rules on UML schemes. In [5] and [6], a program is divided into “phases” describing similar behaviour. Based on the behaviour of the software, design level optimisations are proposed to achieve lower energy consumption. A lot of research is dedicated to building compilers that optimize code for energy-efficiency, e.g. in GCC [7] or in an *iterative* compiler [8]. Petri-net based energy modelling techniques for embedded systems are proposed in [9] and [10]. Analyses for consumption of generic resources are built using recurrence relation solving [11], amortized analysis [12], amortization and separation logic [13] and a Vienna Development Method style program logic [14]. The main differences with our work are that we include an explicit hardware model and a context in the form of component states. This context enables the inclusion of state-dependent energy consumption. Relatively close to our approach are [15] and [16], in which energy consumption of embedded software is analysed for specific architectures ([15] for SimpleScalar, [16] for XMOS ISA-level models), while our approach is hardware-parametric.

Our Approach Contrary to the approaches above, we are interested in statically deriving bounds on energy-consumption using a novel, generic approach that is parametrised with hardware models. Energy consumption analysis is an instance of resource consumption analysis. Other instances are worst-case execution time [17], size [18], loop bound [19,20] and memory [20] analysis). The focus of this paper is on energy analysis. Energy consumption models of hardware components are input for our analysis. The analysis requires information about the software, such as dependencies between variables and information about the number of loop iterations. For this reason we assume that a previous analysis (properly instantiated for our case) has been made deriving loop bounds (e.g. [21,20]) and variable dependency information (e.g. [22]).

Our approach is essentially an energy-aware Hoare logic that is proven sound with respect to an energy-aware semantics. Both the semantics and the logic assume energy-aware component models to be present. The central control is however assumed to be in the software. Consequently, the analysis is done on a hybrid system of software and models of hardware components. The Hoare logic yields an upper bound on the energy consumption of a system of hardware components that are controlled by software.

Our contribution The main contributions of this paper are:

- A novel hardware-parametric energy-aware software semantics.
- A corresponding energy-aware Hoare logic that enables formal reasoning about energy consumption such as deriving an upper-bound for the energy consumption of the system.
- A soundness proof of the derived upper-bounds with respect to the semantics.

The basic modelling and semantics are presented in Sect. 2. Energy-awareness is added and the logic is presented in Sect. 3. An example is given in Sect. 4 and the soundness proof is outlined in Sect. 5. The paper is concluded in Sect. 6.

2 Modelling Hybrid Systems

Most modern electronic systems consist of hardware and software. In order to study the energy consumption of such hybrid systems we will consider both hardware and software in one single modelling framework. This section defines a hybrid logic in which software plays a central role controlling hardware components. The hardware components are modelled in such a way that only the relevant information for the energy consumption analysis is present. In this paper, the controlling software is assumed to be written in a small language designed just for illustrating the analysis.

2.1 Language

Our analysis is performed on a 'while' language. The grammar for our language is defined as follows (where $\boxdot \in \{+, -, *, >, \geq, \equiv, \not\equiv, \leq, <, \wedge, \vee\}$.):

$$\begin{aligned} c \in \textsc{Const} &= n \in \mathbb{N} \\ x \in \textsc{Var} &= \text{'A'} \mid \text{'B'} \mid \text{'C'} \mid \ \ldots \\ e \in \textsc{Expr} &= c \mid x \mid x = e_1 \mid e_1 \boxdot e_2 \mid C_i :: f(e_1) \mid f(e_1) \mid S, e_1 \\ S \in \textsc{Statement} &= \textbf{skip} \mid S_1; S_2 \mid e \mid \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if} \\ &\quad \mid \textbf{while } e \textbf{ do } S \textbf{ end while} \mid F\, S_1 \\ F \in \textsc{Func} &= \textbf{function } f(x) \textbf{ begin } e \textbf{ end} \end{aligned}$$

This language is used just for illustration purposes, so the only supported type in the language is unsigned integer. There are no explicit booleans. The value 0 is handled as a **False** value, while all the other values are handled as a **True** value. There are no global variables and parameters are passed by-value, so functions do not have side-effects on the program state. `while` loops are supported but recursion is not. Functions are statically scoped. There are explicit statements for operations on hardware components, like the processor, memory, storage or network devices. By explicitly introducing these statements it is easier to reason about those components, as opposed to, for instance, using conventions about certain memory regions that will map to certain hardware devices. Functions on components have a fixed number of arguments and always return a value. The notation $C_i :: f$ will refer to a function f of a component C_i.

2.2 Modelling Components

To reason about hybrid systems we need a way to model hardware components (e.g. memory, harddisk, network controller) that captures the behaviour of those components with respect to resource consumption. Hence, we introduce a *component model* that consists of a state and a set of functions that can change the state: *component functions.* A component state $C_i :: s$ is a collection of variables of any type. They can signify e.g. that the component is on, off or in stand-by.

A component function is modelled by a function that produces the return value (rv_f) and a function that updates the internal state of the component (δ_f). Both functions are functions over the state variables. The update function $C_i :: \delta_f$ and the return value function $C_i :: rv_f$ take the state s and the arguments *args* passed to the component function and return respectively the new state of the component and the return value. Each component C_i may have multiple component functions. All the state changes in components must be explicit in the source code as an operation, a *component function*, on that specific component.

2.3 Semantics

Standard, non-energy-aware semantics can be defined for our language. Full semantics are given in a technical report [23]. Below, the assignment rule ($sAssign$) and the component function call rule ($sCallCmpF$) are given to illustrate the notation and the way of handling components.

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma \rangle \Downarrow^e \langle n, \sigma', \Gamma' \rangle}{\Delta \vdash \langle x_1 = e, \sigma, \Gamma \rangle \Downarrow^e \langle n, \sigma'[x_1 \leftarrow n], \Gamma' \rangle}\text{(sAssign)}$$

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma \rangle \Downarrow^e \langle a, \sigma', \Gamma' \rangle \quad C_i :: rv_f(C_i^\Gamma :: s, a) = n \quad \Gamma' = \Gamma[C_i :: s \leftarrow C_i :: \delta_f(C_i^\Gamma :: s, a)]}{\Delta \vdash \langle C_i :: f(e), \sigma, \Gamma \rangle \Downarrow^e \langle n, \sigma, \Gamma' \rangle}\text{(sCallCmpF)}$$

The rules are defined over a triple $\langle e, \sigma, \Gamma \rangle$ with respectively a program expression e (or statement S), the program state function σ and the component state environment Γ. The *program* state function returns for every variable its actual value. Δ is an environment of function definitions. We use the following notation for substitution: $\sigma[x_i \leftarrow n]$. The reduction symbol $\Downarrow^e$ is used for expressions, which evaluate to a value and a new state function. We use $\Downarrow^s$ for statements, which only evaluate to a new state function.

In the following sections we will define energy-aware semantics and energy analysis rules. We used a consistent naming scheme for the different variants of the rules (e.g. **s**Assign, **e**Assign and **a**Assign for the Assignment rule in respectively the **s**tandard non-energy-aware semantics, the **e**nergy aware semantics and the energy **a**nalysis rules).

3 Energy Analysis of Hybrid Systems

In this section we extend our hybrid logic in order to reason about the energy consumption of programs. We distinguish two kinds of energy usage: *incidental* and *time-dependent.* The former represents an operation that uses a constant amount of energy, disregarding any time aspect. The latter signifies a change in the state of the component; while a component is in a certain state it is assumed to draw a constant amount of energy *per time unit.*

3.1 Energy-Aware Semantics

As energy consumption can be based on time, we first need to extend our semantics to be time-aware. We effectively extend all the rules of the semantics with an extra argument, a global timestamp $\mathfrak{t}$. Using this timestamp we are able to model and analyse time-dependent energy usage.

We track energy usage for each component individually, by using an accumulator $\mathfrak{e}$ that is added to the component model. For time-dependent energy usage, with each component state change, the energy used while the component was in the previous state is added to the accumulator. To enable calculation of the time spent in the current state, we add τ to the component model, signifying the timestamp at which the component entered the current state. We assume that each component has a constant *power draw* while in a state. Therefore, the component model function $C_i :: \phi(s)$ maps component states onto the corresponding power draw, independent of time. To calculate the power consumed while in a certain state we define the *td* function, with as arguments the component and the current timestamp:

$$td(C_i, t) = C_i :: \phi(s) \cdot (t - C_i :: \tau)$$

We model *incidental energy usage* associated with a component function f with the constant $C_i :: \mathfrak{E}_f$. For each call to a component function we add this constant to the energy accumulator.

A component function call can influence energy consumption in two ways: through its associated incidental energy consumption and by changing the state, thereby influencing time-dependent energy usage. This is expressed by the energy-aware semantic rule (*eCallCmpF*) for component functions as defined below, with $C_i :: \mathfrak{T}_f$ representing the time it costs to execute this component function.

$$\frac{\begin{array}{c}\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle a, \sigma', \Gamma', \mathfrak{t}'\rangle \quad C_i :: rv_f(C_i^{\Gamma'} :: s, a) = n \\ \Gamma'' = \Gamma[C_i :: \mathfrak{e} \mathrel{+}= C_i :: \mathfrak{E}_f + td(C_i^{\Gamma'}, \mathfrak{t}), C_i :: s \leftarrow C_i :: \delta_f(C_i^{\Gamma'} :: s, a), C_i :: \tau \leftarrow \mathfrak{t}']\end{array}}{\Delta \vdash \langle C_i :: f(e), \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle n, \sigma, \Gamma'', \mathfrak{t}' + C_i :: \mathfrak{T}_f\rangle}\text{(eCallCmpF)}$$

Note the addition of the incidental and time dependent energy usages ($C_i :: \mathfrak{E}_f$ and $td(C_i^{\Gamma}, \mathfrak{t})$ respectively) to the energy accumulator $C_i :: \mathfrak{e}$, the incrementation of the global time with $C_i :: \mathfrak{T}_f$ and the update of the component timestamp $C_i :: \tau$. Evaluation by $\Downarrow$ in the energy-aware semantics extends the original semantics with a timestamp and an energy accumulator, which are used to calculate the total energy consumption of the evaluation ($\mathfrak{e}_{system}$ as defined below). The full energy-aware semantics are given in Fig. 1.

The energy accumulator of the components is not always up to date with respect to the current time, as it is only updated in the (*eCallCmpF*) rule. This is done for simplicity; otherwise each rule that adjusts the global time needs to update the energy accumulator of all components.

To calculate the total actual energy usage, the time the components are in their current state should still be accounted for. This means we have to add the result of the *td* function for each component. The total energy consumption of the system can be calculated at any time as follows:

$$\mathfrak{e}_{system}(\Gamma, \mathfrak{t}) = \sum_i C_i^{\Gamma} :: \mathfrak{e} + td(C_i^{\Gamma}, \mathfrak{t})$$

We can now make the distinction between non-energy-aware component state $C_i :: s$, and energy-aware component state, which also includes the time-stamp τ and the energy accumulator $\mathfrak{e}$.

Most energy consuming actions are explicit in our language: $C_i :: consume()$. However, basic language features, such as evaluation of arithmetic expressions, also implicitly consume energy. We capture this behaviour in the C_{imp} component. This component is an integral part of our energy-aware semantics and logic. The C_{imp} component should at least have resource consumption constants defined for the following operations:

- $C_{imp} :: \mathfrak{E}_e$ and $C_{imp} :: \mathfrak{T}_e$ for expression evaluation.
- $C_{imp} :: \mathfrak{E}_a$ and $C_{imp} :: \mathfrak{T}_a$ for assignment.
- $C_{imp} :: \mathfrak{E}_w$ and $C_{imp} :: \mathfrak{T}_w$ for while loops.
- $C_{imp} :: \mathfrak{E}_{ite}$ and $C_{imp} :: \mathfrak{T}_{ite}$ for conditionals.

To capture the resource consumption of these basic operations, we extend the associated rules in the semantics. The energy-aware rule for assignment (*eAssign*) is listed below, with $C_{imp} :: \mathfrak{E}_a$ for the incidental energy usage of an assignment and $C_{imp} :: \mathfrak{T}_a$ for the time it takes to perform an assignment.

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \quad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\Delta \vdash \langle x = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle n, \sigma'[x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_a\rangle}\text{(eAssign)}$$

All computations of resource consumption and new component states are done symbolically. In the logic, these values are added, multiplied and subtracted or their `max` is taken. Hence, every $\mathfrak{t}$, $\mathfrak{e}$ and τ, as well as the values in component states, are polynomial expressions, extended with the `max` operator, over program variables. Additionally, symbolic states are used, both as input for the program and as start state for the components. The aforementioned polynomials also range over the symbols used in these symbolic states.

3.2 Energy-Aware Modelling

Energy-aware models will be used to derive upper-bounds for energy consumption of the modelled system. In order for the energy-aware model to be suited for the analysis the model should reflect an upper-bound on the actual consumption. This can be based on detailed documentation or on actual energy measurements.

To provide a sound analysis, we need to assume that components are modelled in such a way that the component states reflect different power-levels and are partially ordered. Greater states should imply greater power draw. We will use finite state models only to enable fixpoint calculation in our analysis of while loops. The modelling should be such that the following properties hold (in the context of the full soundness proof in [23] these properties are axioms):

- *Components states form a finite lattice* with a partial order based on the ordering of polynomials (extended with `max`) over symbolic variables. Within the lattice each pair of component states has a least upper-bound.
- *Energy-aware component states are partially ordered.* This ordering extends the ordering on component states in a natural way by adding an energy accumulator and a timestamp. The timestamp stores the time of the latest

$$\frac{}{\Delta \vdash \langle c, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle c, \sigma, \Gamma, \mathfrak{t} \rangle}(\text{eConst}) \qquad \frac{}{\Delta \vdash \langle x, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle \sigma(x), \sigma, \Gamma, \mathfrak{t} \rangle}(\text{eVar})$$

$$\frac{\begin{array}{cc}\Delta \vdash \langle e_1, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}' \rangle & C_{imp} :: \boxdot(n, m) = p \\ \Delta \vdash \langle e_2, \sigma', \Gamma', \mathfrak{t}' \rangle \Downarrow^e \langle m, \sigma'', \Gamma'', \mathfrak{t}'' \rangle & \Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_e]\end{array}}{\Delta \vdash \langle e_1 \boxdot e_2, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle p, \sigma'', \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_e \rangle}(\text{eBinOp})$$

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}' \rangle \quad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\Delta \vdash \langle x = e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma'[x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_a \rangle}(\text{eAssign})$$

$$\frac{\begin{array}{c}\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle a, \sigma', \Gamma', \mathfrak{t}' \rangle \quad C_i :: rv_f(C_i^{\Gamma'} :: s, a) = n \\ \Gamma'' = \Gamma[C_i :: \mathfrak{e} \mathrel{+}= C_i :: \mathfrak{E}_f + td(C_i^{\Gamma'}, \mathfrak{t}), C_i :: s \leftarrow C_i :: \delta_f(C_i^{\Gamma'} :: s, a), C_i :: \tau \leftarrow \mathfrak{t}']\end{array}}{\Delta \vdash \langle C_i :: f(e), \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma, \Gamma'', \mathfrak{t}' + C_i :: \mathfrak{T}_f \rangle}(\text{eCallCmpF})$$

$$\frac{\begin{array}{c}\Delta(f) = (e_1, \Delta', x) \\ \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle a, \sigma', \Gamma', \mathfrak{t}' \rangle \quad \Delta' \vdash \langle e_1, [x \leftarrow a], \Gamma', \mathfrak{t}' \rangle \Downarrow^e \langle n, \sigma'', \Gamma'', \mathfrak{t}'' \rangle\end{array}}{\Delta \vdash \langle f(e), \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma', \Gamma'', \mathfrak{t}'' \rangle}(\text{eCallF})$$

$$\frac{\Delta \vdash \langle S, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma', \Gamma', \mathfrak{t}' \rangle \quad \Delta \vdash \langle e, \sigma', \Gamma', \mathfrak{t}' \rangle \Downarrow^e \langle n, \sigma'', \Gamma'', \mathfrak{t}'' \rangle}{\Delta \vdash \langle S, e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma'', \Gamma'', \mathfrak{t}'' \rangle}(\text{eExprConcat})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}' \rangle}{\Delta \vdash \langle e_1, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma', \Gamma', \mathfrak{t}' \rangle}(\text{eExprAsStmt}) \qquad \frac{}{\Delta \vdash \langle \mathbf{skip}, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t} \rangle}(\text{eSkip})$$

$$\frac{\Delta \vdash \langle S_1, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma', \Gamma', \mathfrak{t}' \rangle \quad \Delta \vdash \langle S_2, \sigma', \Gamma', \mathfrak{t}' \rangle \Downarrow^s \langle \sigma'', \Gamma'', \mathfrak{t}'' \rangle}{\Delta \vdash \langle S_1; S_2, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma'', \Gamma'', \mathfrak{t}'' \rangle}(\text{eStmtConcat})$$

$$\frac{\begin{array}{cc} & \Delta \vdash \langle S_2, \sigma', \Gamma', \mathfrak{t}' \rangle \Downarrow^s \langle \sigma'', \Gamma'', \mathfrak{t}'' \rangle \\ \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle 0, \sigma', \Gamma', \mathfrak{t}' \rangle & \Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{ite}]\end{array}}{\Delta \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if}, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma'', \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_{ite} \rangle}(\text{eIf-False})$$

$$\frac{\begin{array}{cc} n \neq 0 & \Delta \vdash \langle S_1, \sigma', \Gamma', \mathfrak{t}' \rangle \Downarrow^s \langle \sigma'', \Gamma'', \mathfrak{t}'' \rangle \\ \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}' \rangle & \Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{ite}]\end{array}}{\Delta \vdash \langle \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if}, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma'', \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_{ite} \rangle}(\text{eIf-True})$$

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle 0, \sigma', \Gamma', \mathfrak{t}' \rangle \quad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_w]}{\Delta \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ S_1\ \mathbf{end\ while}, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma', \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_w \rangle}(\text{eWhile-False})$$

$$\frac{\begin{array}{c}\Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_w] \quad \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}' \rangle \\ \Delta \vdash \langle S_1; \mathbf{while}\ e\ \mathbf{do}\ S_1\ \mathbf{end\ while}, \sigma', \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_w \rangle \Downarrow^s \langle \sigma'', \Gamma''', \mathfrak{t}'' \rangle \quad n \neq 0\end{array}}{\Delta \vdash \langle \mathbf{while}\ e\ \mathbf{do}\ S_1\ \mathbf{end\ while}, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma'', \Gamma''', \mathfrak{t}'' \rangle}(\text{eWhile-True})$$

$$\frac{\Delta[f \leftarrow (e, \Delta, x)] \vdash \langle S, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma', \Gamma', \mathfrak{t}' \rangle}{\Delta \vdash \langle \mathbf{function}\ f(x)\ \mathbf{begin}\ e\ \mathbf{end}\ S, \sigma, \Gamma, \mathfrak{t} \rangle \Downarrow^s \langle \sigma', \Gamma', \mathfrak{t}' \rangle}(\text{eFuncDef})$$

Fig. 1. Energy-aware semantics.

change to the component state. So, the earliest timestamp reflects the highest energy usage. Therefore, with respect to timestamps the energy-aware component state ordering should be defined such that smaller timestamps lead to bigger energy-aware component states.

– *Power draw functions preserve the ordering*, i.e. larger states consume more energy than smaller states.
– *Component state update functions δ preserve the ordering.* For this reason, δ_f cannot depend on the arguments of f. To signify this, we will use $\delta(s)$ instead of $\delta(s, args)$ in the logic. As a result, component models cannot influence each other. Our soundness proof (Theorem 2 in Sect. 5) requires this assumption.

Severeness of model restrictions There are several restrictions to the modelling that may seem far from reality.

1. *Component state functions take up a constant amount of time and incidental energy.* This is needed for the soundness proof. For instance, when a radio component sends a message, the duration of the function call cannot directly depend on the number of bytes in the message. In most cases this can be dealt with by using a different way of modelling. First, one can use an overestimation. Second, such dependencies can be removed by distributing the costs over multiple function calls. For instance, the radio component can have a function to send a fixed number of bytes. If it internally keeps a queue, the additional costs of sending the full queue can be modelled by distributing it over separate queueing operations. energy consumption of components must remain fixed per component state.
2. *With each component state a constant power draw is associated.* However, some hardware may accumulate heat over time incurring increasing energy consumption over time. Such a 'heating' problem can be modelled e.g. by changing state to a higher energy level with every call of a component function. This is still an approximation of course. In the future, we want to study models with time driven state change or with time-dependent power draw.
3. *Component model must be finite state machines.* Modelling systems with finite state machines is not uncommon, e.g using model checking and the right kind of abstraction for the property that is studied. In our models the abstraction should be such that the energy consumption is modelled as close as possible.
4. *The effect of component state functions on the component states cannot depend on the arguments of the function.* Also, component models cannot influence each other. Both restrictions are needed for soundness guarantee of our analysis. This restricts the modelling. Using multiple component state functions instead of dynamic arguments and cross-component calls is a way of modelling that can mitigate these restrictions in certain cases. Relieving these restrictions in general is part of future work.

3.3 A Hoare Logic for Energy Analysis

This section treats the definition of an energy-aware logic with energy analysis rules that can be used to bound the energy consumption of the analysed system. The full set of rules is given in Fig. 2. These rules are deterministic; at each moment only one rule can be applied.

Our energy consumption analysis depends on external symbolic analysis of variables and loop analysis. The results of this external analysis are assumed to be accessible in our Hoare Logic in two ways.

Firstly, we restrict the scope of our analysis to programs that are bound in terms of execution. We assume that all loops and component functions terminate on any input. Each loop is annotated with a bound: $\textbf{while}_{ib}$. The bound is a polynomial over the input variables, which expresses an upper-bound on the number of iterations of the loop. We have added the *ib* to the while rule in the energy analysis rules to make this assumption explicit. Derivation of bounds is

$$\frac{}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} n \{\mathit{\Gamma}; \mathfrak{t}; \rho\}}(\text{aConst}) \qquad \frac{}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} x \{\mathit{\Gamma}; \mathfrak{t}; \rho\}}(\text{aVar})$$

$$\frac{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} e_1 \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1\} \quad \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1\} e_2 \{\mathit{\Gamma}_2; \mathfrak{t}_2; \rho_2\} \quad \mathit{\Gamma}_3 = \mathit{\Gamma}_2[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_e]}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} e_1 \boxdot e_2 \{\mathit{\Gamma}_3; \mathfrak{t}_2 + C_{imp} :: \mathfrak{T}_e; \rho_2\}}(\text{aBinOp})$$

$$\frac{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} e \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1\} \quad \mathit{\Gamma}_2 = \mathit{\Gamma}_1[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} x = e \{\mathit{\Gamma}_2; \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_a; \rho_2\}}(\text{aAssign})$$

$$\frac{\mathit{\Gamma}_1 = \mathit{\Gamma}[C_i :: s \leftarrow C_i :: \delta_f(C_i :: s), C_i :: \tau \leftarrow \mathfrak{t}, C_i :: \mathfrak{e} \mathrel{+}= C_i :: \mathfrak{E}_f + td(C_i, \mathfrak{t})]}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} C_i :: f(args) \{\mathit{\Gamma}_1; \mathfrak{t} + C_i :: \mathfrak{T}_f; \rho\}}(\text{aCallCmpF})$$

$$\frac{\begin{array}{cc} \mathit{\Delta}(f) = (e_1, x) & \{\mathit{\Gamma}; \mathfrak{t}; \rho\} e \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1\} \\ e = a \in \rho & \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1[x' \leftarrow a]\} e_1[x \leftarrow x'] \{\mathit{\Gamma}_2; \mathfrak{t}_2; \rho_2\} \quad x' \text{ fresh in } e_1 \end{array}}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} f(e) \{\mathit{\Gamma}_2; \mathfrak{t}_2; \rho_2\}}(\text{aCallF})$$

$$\frac{}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} \mathbf{skip} \{\mathit{\Gamma}; \mathfrak{t}; \rho\}}(\text{aSkip}) \qquad \frac{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} S_1 \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1\} \quad \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1\} S_2 \{\mathit{\Gamma}_2; \mathfrak{t}_2; \rho_2\}}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} S_1; S_2 \{\mathit{\Gamma}_2; \mathfrak{t}_2; \rho_2\}}(\text{aConcat})$$

$$\frac{\begin{array}{cc} \{\mathit{\Gamma}; \mathfrak{t}; \rho\} e \{\mathit{\Gamma}_1; \mathfrak{t}_1; \rho_1\} & \{\mathit{\Gamma}_2; \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_{ite}; \rho_1\} S_1 \{\mathit{\Gamma}_3; \mathfrak{t}_2; \rho_2\} \\ \mathit{\Gamma}_2 = \mathit{\Gamma}_1[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{ite}] & \{\mathit{\Gamma}_2; \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_{ite}; \rho_1\} S_2 \{\mathit{\Gamma}_4; \mathfrak{t}_3; \rho_3\} \end{array}}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if} \{\mathbf{lub}(\mathit{\Gamma}_3, \mathit{\Gamma}_4); \max\{\mathfrak{t}_2, \mathfrak{t}_3\}; \rho_4\}}(\text{aIf})$$

$$\frac{\begin{array}{cc} \mathit{\Gamma}_1 = \mathbf{process\text{-}td}(\mathit{\Gamma}, \mathfrak{t}) & \mathit{\Gamma}_3 = \mathit{\Gamma}_2[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_w] \\ \{\mathbf{wci}(\mathit{\Gamma}_1, e; S); \mathfrak{t}; \rho\} e \{\mathit{\Gamma}_2; \mathfrak{t}_1; \rho_1\} & \{\mathit{\Gamma}_3; \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_w; \rho_1\} S \{\mathit{\Gamma}_4; \mathfrak{t}_2; \rho_2\} \end{array}}{\{\mathit{\Gamma}; \mathfrak{t}; \rho\} \mathbf{while}_{ib}\ e\ \mathbf{do}\ S\ \mathbf{end\ while} \{\mathbf{oe}(\mathit{\Gamma}_1, \mathfrak{t}, \mathit{\Gamma}_4, \mathfrak{t}_2, ib); \rho_3\}}(\text{aWhile})$$

Fig. 2. Energy analysis rules.

considered out of scope for our analysis. We assume that an external analysis has produced a sound bound.

Secondly, the symbolic state environment ρ gives us a symbolic state of every variable for each line of code, e.g, $\{x_1 = e_1\} x_1 = x_1 + x_2 + x_3 \{x_1 = e_1 + x_2 + x_3\}$, plus other non-energy related properties invariants that have already been proven. In Fig. 2 we included this prerequisite by explicitly denoting it as ρ, ρ_1, $\ldots$.

All the judgements in the rules have the following shape: $\{\mathit{\Gamma}; \mathfrak{t}; \rho\} S\ \{\mathit{\Gamma}'; \mathfrak{t}'; \rho'\}$, where $\mathit{\Gamma}$ is the set of all energy aware component states, $\mathfrak{t}$ is the global time and ρ represents the symbolic state environment retrieved from the earlier standard analysis. The notation $\mathit{\Gamma}[n \mathrel{+}= m]$ is a shorthand for $\mathit{\Gamma}[n \leftarrow n + m]$. As (energy-aware) component states are partially ordered, we can take a least upper bound of states $\mathbf{lub}(s_1, s_2)$ and sets of energy-aware component states $\mathbf{lub}(\mathit{\Gamma}_1, \mathit{\Gamma}_2)$.

We will highlight the most relevant aspects of the rules. The (*aCallCmpF*) rule uses the $td(C_i, t)$ function to estimate the time-dependent energy consumption of component function calls. The (*aIf*) rule takes the least upper bound of the energy-aware component states and the maximum of the time estimates.

Special attention is warranted for the (*aWhile*) rule. We study the body of the while loop in isolation. This requires processing the time-dependent energy consumption that occurred before the loop (**process-td**). An over-estimation (**oe**) of the energy consumption of the loop will be calculated by taking the product of the bound on the number of iterations and an over-estimation of the energy consumption of a single iteration, i.e. the worst-case iteration (**wci**). The worst-case-iteration is determined by taking the least upper-bound of the set of all states that can occur during the execution of the loop. As there are a finite number of states for each component, this set can be determined via a fix point construction (**fix**). The fixpoint is calculated by iterating the component iteration function (**ci**).

In order to support the analysis of statements after the loop, also an overestimation of the component states after the loop has to be calculated. For brevity in Fig. 2, this is dealt with in the calculation of **oe**.

Five calculations are needed:

1. Component iteration function **ci**. The component iteration function $\mathbf{ci}_i(S)$ aggregates the (possibly overestimated) effects of S on C_i. It performs the analysis on S, then considers only the effects on C_i. If there are nested loops or conditionals, the effects on the state of C_i are overestimated in the same manner as in the rest of the analysis. By $\mathbf{ci}_i^n(S)$ we mean the component iteration function applied n times: $\mathbf{ci}_i(S) \circ \mathbf{ci}_i^{n-1}(S)$, with $\mathbf{ci}_i^1(S) = \mathbf{ci}_i(S)$.
2. Fixpoint function **fix**. Because component states are finite, there is an iteration after which a component is in a state that it has already been in, earlier in the loop (unless the loop is already finished before this point is reached). Since components are independent, the behaviour of the component will be identical to its behaviour the first time it was in this state. This is a fixpoint on the set of component states that can be reached in the loop. It can be found using the $\mathbf{fix}_i(S)$ function, which finds the smallest n for which $\exists k.\mathbf{ci}_i^{n+1}(S) = \mathbf{ci}_i^k(S)$. The number of possible component states is an upper bound for n.
3. Worst-Case Iteration function **wci**. To make a sound overestimation of the energy consumption of a loop, we need to find the iteration that consumes the most. As our analysis is non-decreasing with respect to component states, this is the iteration which starts with the largest component state in the precondition. For this purpose, we introduce the worst-case iteration function $\mathbf{wci}_i(S)$, which computes the least-upper bound of all the states up to the fixpoint: $\mathbf{wci}_i(S) = \mathbf{lub}(\mathbf{ci}_i^0(S), \mathbf{ci}_i^1(S), \ldots, \mathbf{ci}_i^{\mathbf{fix}_i(S)}(S))$. The global version $\mathbf{wci}(\mathit{\Gamma}, S)$ is defined by iteratively applying the $\mathbf{wci}_i(S)$ function to each component C_i in $\mathit{\Gamma}$.
4. Overestimation function **oe**. This function overestimates the energy-aware output states of the loop. It needs to do three things: find the largest non-energy-aware output states, find the minimal timestamps and add the resource consumption of the loop itself. This function gets as input: the start state of the loop $\mathit{\Gamma}_{in}$, the start time $\mathfrak{t}$, the output state from the analysis of the worst-case iteration $\mathit{\Gamma}_{out}$, the end time from the analysis of the worst-case iteration $\mathfrak{t}'$ and the iteration bound ib. It returns an overestimated energy-aware component state and an overestimated global time.
   Because component state update functions δ preserve the ordering, the analysis of the worst-case iteration results in the maximum output state for any iteration. This, however, does not yet address the case where the loop is not entered at all. Therefore, we need to take the least-upper bound of the start state and the result of the analysis of the worst-case iteration.
   To overestimate time-dependent energy usage, we must revert component timestamps to the time of entering the loop. So, if a component is switched to a greater state at some point in the loop, the analysis assumes it has been in this state since entering the loop. Note that the least-upper bound of energy-aware component states does exactly this: maximise the non-energy-aware

component state and minimise the timestamp. Taking $\Gamma_{base} = \mathbf{lub}(\Gamma_{in}, \Gamma_{out})$ we find both the maximum output states and the minimum timestamps. Now, we can add the consumption of the loop itself. We perform the following calculation for each component: $C_i^{\Gamma_{base}} :: \mathfrak{e} = C_i^{\Gamma_{in}} :: \mathfrak{e} + (C_i^{\Gamma_{out}} :: \mathfrak{e} - C_i^{\Gamma_{in}} :: \mathfrak{e}) \cdot ib$. We do something similar for the time consumption: $\mathfrak{t}_{ret} = \mathfrak{t} + ((\mathfrak{t}' - \mathfrak{t}) \cdot ib)$.

5. Processing time-dependent energy function **process-td**. When analysing an iteration of a loop, we must take care not to include any energy consumption outside of the iteration. This would lead to a large overestimation, since it would be multiplied by the (possibly overestimated) number of iterations. Therefore, before analysing the body, we add the time-dependent energy consumption to the energy accumulator for each component and set all timestamps to the current time. Otherwise, the time-dependent consumption before entering the loop would also be included in the analysis of the iteration. We introduce the function **process-td**$(\Gamma, \mathfrak{t})$, which adds $td(C_i, \mathfrak{t})$ to $C_i :: \mathfrak{e}$ and sets $C_i :: \tau$ to $\mathfrak{t}$, for each component C_i in Γ.

Applying the rules overestimates the sum of the incidental energy consumption and the time-dependent energy consumption. However, the time-dependent energy consumption is only added to the accumulator at changes of component states. So, as for the energy-aware semantics, the time the components are in their current state should still be accounted for by calculating $\mathfrak{e}_{system}(\Gamma_{end}, \mathfrak{t}_{end})$.

4 Example: Wireless Sensor Node

To illustrate our analysis, we model a wireless sensor node, which has a sensor C_s and a radio C_r. Furthermore, it has a basic C_{imp} component for the implicit resource consumption. We analyse the energy usage of a program that repeatedly measures the sensor for 10 seconds, then sends the measurement over the radio, shown in Fig. 3. The example illustrates both *time-dependent* (sensor) and *incidental* (radio) energy usage. We choose a highly abstract modelling to keep the example brief and simple. A more elaborate example can be found in [23], in which a less abstract modelling is used and two algorithms are compared.

```
while_n $n > 0$ do
    $C_s :: on()$;
        ... some code taking 10 seconds ...
    $x = C_s :: off()$;
    $C_r :: send(x)$;
    $n = n - 1$;
end while;
```

Fig. 3. Example program.

Modelling The sensor component C_s has two states: s_{on} and s_{off}. It does not have any incidental energy consumption. It has a power draw (thus time-dependent consumption) only when on. For this power draw we introduce the constant $\mathfrak{e}_{\text{on}}$. There are two component functions, namely `on` and `off`, which switch between the two component states.

The radio component C_r only has incidental energy consumption. It does not have a state. Its single component function is `send`, which uses $C_r :: \mathfrak{T}_{send}$ time and $C_r :: \mathfrak{E}_{send}$ energy.

The C_{imp} component models the *implicit* resource consumption by various language constructs. For the sake of presentation, we choose a very simple model here, in which only assignment consumes time and energy. We set both the associated constants $C_{imp} :: \mathfrak{T}_a$ and $C_{imp} :: \mathfrak{E}_a$. The other six constants in the C_{imp} model (see Sect. 3.1 for a list) are set to 0.

Application of the energy-aware semantics from Fig. 1 on the loop body results in a time consumption $\mathfrak{t}_{\text{body}}$ of $10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a$ and an energy consumption $\mathfrak{e}_{\text{body}}$ of $10 \cdot \mathfrak{e}_{\text{on}} + C_r :: \mathfrak{E}_{send} + 2 \cdot C_{imp} :: \mathfrak{E}_a$. Intuitively, the time and energy consumptions of the whole loop are $n(\mathfrak{t}_{\text{body}})$ and $n(\mathfrak{e}_{\text{body}})$.

Energy consumption analysis The analysis (Fig. 2) always starts with a symbolic state. Note that only the sensor component C_s has a state. We introduce the symbol $\mathbf{on_0^s}$ for the symbolic start-state (on or off) of the sensor.

We start the analysis with the while loop. So, we apply the (*aWhile*) rule:

$$\frac{\begin{array}{c}\mathit{\Gamma}_1 = \textbf{process-td}(\mathit{\Gamma}_0, \mathfrak{t}_0) \\ \{\textbf{wci}(\mathit{\Gamma}_1, n > 0; S_{\text{body}}); \mathfrak{t}_0; \rho_0\} n > 0 \{\mathit{\Gamma}_2; \mathfrak{t}_1; \rho_1\} \quad \{\mathit{\Gamma}_2; \mathfrak{t}_1; \rho_1\} S_{\text{body}} \{\mathit{\Gamma}_{it}; \mathfrak{t}_{it}; \rho_{it}\}\end{array}}{\{\mathit{\Gamma}_0; \mathfrak{t}_0; \rho_0\} \textbf{while}_n\ n > 0 \textbf{ do } S_{it} \textbf{ end while} \{\textbf{oe}(\mathit{\Gamma}_1, \mathfrak{t}_0, \mathit{\Gamma}_{it}, \mathfrak{t}_{it}, \rho(n)); \rho_{end}\}} \text{(aWhile)}$$

Since $C_{imp} :: \mathfrak{E}_w$ and $C_{imp} :: \mathfrak{T}_w$ are 0, we omit them here. We will first solve the **process-td** and **wci** functions, then analyse the loop guard and body (i.e. the part above the line), then determine the final results with the **oe** function.

We first add time-dependent energy consumption and set timestamps to $\mathfrak{t}_0$ for all components using the **process-td** function. If we would not do this, the time-dependent energy consumption *before* the loop would be included in the calculation of the resource consumption of the worst-case iteration. As this would be multiplied by the number of iterations, it would lead to a large overestimation. C_r and C_{imp} do not have a state, so we only need to add the time-dependent consumption of C_s: $td(C_s, \mathfrak{t}_0) = C_s :: \phi(\mathbf{on_0^s}) \cdot (\mathfrak{t}_0 - C_s :: \tau_0)$, where $C_s :: \tau_0$ is the symbolic value of the sensor timestamp before starting the analysis.

We must now find the worst-case iteration, using the **wci** function. For the C_s component we need the $\mathbf{ci}_s(n > 0; S_{\text{body}})$ function. As the other components do not have a state, $\mathbf{ci}_{imp}(n > 0; S_{\text{body}})$ and $\mathbf{ci}_r(n > 0; S_{\text{body}})$ are simply the identity function. The loop body sets the state of the sensor to s_{off}, independent of the start state. So, $\mathbf{ci}_s(n > 0; S_{\text{body}})$ always results in s_{off}. Now we can find the fixpoint. In the first iteration, we enter the loop with symbolic state $\mathbf{on_0^s}$. In the second iteration, the loop is entered with state s_{off}. In the third iteration, the loop is again entered with state s_{off}. We have thus found the fixpoint. The worst-case iteration can now be calculated by $\mathbf{wci}_s(n > 0; S_{\text{body}}) = \mathbf{lub}(\mathbf{ci}_s^0(n > 0; S_{\text{body}}), \mathbf{ci}_s^1(n > 0; S_{\text{body}})) = \mathbf{on_0^s}$. Intuitively this means that, since after any number of iterations the sensor is off, the symbolic start state, in which it is unknown whether the sensor is on or off, yields the worst-case.

As there are no costs associated with the evaluation of expressions, the analysis of $n > 0$ using the (*aBinOp*) rule does not have any effect on the state. We continue with analysis of the loop body, which starts with a call to component function `on`. We apply the (*aCallCmpF*) rule:

$$\frac{\mathit{\Gamma}_3 = \mathit{\Gamma}_2[C_s :: s \leftarrow C_s :: \delta_{on}(C_s :: s), C_s :: \tau \leftarrow \mathfrak{t}_1, C_s :: \mathfrak{e} \mathrel{+}= td(C_s, \mathfrak{t}_1)]}{\{\mathit{\Gamma}_2; \mathfrak{t}_1; \rho_1\} C_s :: on() \{\mathit{\Gamma}_3; \mathfrak{t}_1; \rho_2\}}\text{(aCallCmpF)}$$

There is no incidental energy consumption or time consumption associated with the call. We must however add the time-dependent energy consumption to the energy accumulator, by adding $td(C_s, \mathfrak{t})$. Since we have just set $C_s :: \tau$ to $\mathfrak{t}_0$ and the evaluation of $n > 0$ costs 0 time, hence $\mathfrak{t}_1 = \mathfrak{t}_0$, $td(C_s, \mathfrak{t}_1)$ results in 0. The function $C_r :: \delta_{\text{on}}$ produces new component state s_{on}. It also saves the current timestamp to the component state, in order to know when the last state transition happened. For simplicity, we omit the application of the concatenation rule (*aConcat*) in the following.

After ten seconds of executing other statements (which we assume only cost time, not energy), the sensor is turned off. The call to the function *off* returns the measurement, which is assigned to x. We must therefore first apply the (*aAssign*) rule. This adds $C_{imp} :: \mathfrak{T}_a$ to the global time and $C_{imp} :: \mathfrak{E}_a$ to the energy accumulator. We now apply the (*aCallCmpF*) rule to the right-hand side of the assignment, i.e. the call to $C_s :: \mathit{off}$. This updates the state of the component to s_{off}. It also executes the $td(C_s, \mathfrak{t}_2)$ function in order to determine the energy cost of the component being on for ten seconds. Because $\mathfrak{t}_2 = C_s :: \tau + 10$ and our model specifies a power draw of $\mathfrak{e}_{\text{on}}$ for s_{on}, this results in $10 \cdot \mathfrak{e}_{\text{on}}$. We add this to the energy accumulator of the sensor component.

We apply the (*aCallCmpF*) rule again, this time to the *send* function of the C_r component. As the transmission costs a fixed amount of energy, all time-dependent constants associated with transmitting are set to zero. So, the (*aCall-CmpF*) rule will only add the incidental energy usage specified by $C_s :: \mathfrak{E}_{send}$ and the constant time usage $C_s :: \mathfrak{T}_{send}$. Finally, we apply the (*aBinOp*) rule, which has no costs, and the (*aAssign*) rule, which again adds $C_{imp} :: \mathfrak{E}_a$ and $C_{imp} :: \mathfrak{T}_a$.

Analysis of the worst-case iteration results in global time $\mathfrak{t}_{it}$ and energy-aware component state environment $\mathit{\Gamma}_{it}$. We can now apply the overestimation function $\mathbf{oe}(\mathit{\Gamma}_1, \mathfrak{t}_0, \mathit{\Gamma}_{it}, \mathfrak{t}_{it}, \rho(n))$. This takes as base the least-upper bound of $\mathit{\Gamma}_1$ and $\mathit{\Gamma}_{it}$, which in this case is exactly $\mathit{\Gamma}_1$ (note that the state of the sensor is overestimated as $\mathbf{on}^{\mathbf{s}}_{\mathbf{0}}$). It then adds the consumption of the worst-case iteration, multiplied by the number of iterations. The worst-case iteration results in a global time of $\mathfrak{t}_0 + 10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a$. So, $\mathbf{oe}$ results in a global time $\mathfrak{t}_{end}$ of $\mathfrak{t}_0 + n \cdot (10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a)$. Note that this is equal to the time consumption resulting from the energy-aware semantics.

A similar calculation is made for energy consumption, for each component. Then, we can calculate $\mathfrak{e}_{system}$. In total, the $\mathbf{oe}$ function results in an energy usage of $\mathfrak{e}_0 + n \cdot (10 \cdot \mathfrak{e}_{\text{on}} + C_r :: \mathfrak{E}_{send} + 2 \cdot C_{imp} :: \mathfrak{E}_a)$. However, we still need to add the time-dependent energy consumption for each component. This is where potential overestimation occurs in this example. Since C_r and C_{imp} do not have a state, we only need to add the time-dependent consumption of C_s. After the analysis of the loop, the state of the sensor is overestimated as $\mathbf{on}^{\mathbf{s}}_{\mathbf{0}}$. We must therefore add a consumption of $td(C_s, \mathfrak{t}_{\text{end}}) = C_s :: \phi(\mathbf{on}^{\mathbf{s}}_{\mathbf{0}}) \cdot (\mathfrak{t}_{end} - \mathfrak{t}_0) = C_s :: \phi(\mathbf{on}^{\mathbf{s}}_{\mathbf{0}}) \cdot n \cdot (10 + C_r :: \mathfrak{T}_{send} + 2 \cdot C_{imp} :: \mathfrak{T}_a)$. This leads to an overestimation only in case $\mathbf{on}^{\mathbf{s}}_{\mathbf{0}} = s_{\text{on}}$ and $n > 0$. Otherwise, the result of the analysis is equal to that of the energy-aware semantics.

5 Soundness

In this section we outline a proof of the soundness of the energy-aware Hoare logic with respect to the energy-aware semantics. Intuitively, this means we prove that the analysis over-estimates the actual energy consumption. Here, we present only the fundamental theorems. The reader is referred to [23] for the full proof. Soundness of the annotations (loop bounds and symbolic states) is assumed in order to guarantee soundness of the final result.

We first show that the logic over-estimates time consumption. In order to establish soundness of the analysis of the energy consumption of the program, we need to establish first soundness of the timing analysis.

Theorem 1 (Timing over-estimation). *If* $\langle S, \sigma, \Gamma, \mathtt{t}\rangle \Downarrow^s \langle \sigma', \Gamma', \mathtt{t}'\rangle$*, then for any derivation* $\{\Gamma; \mathtt{t}; \rho\} S \{\Gamma_1; \mathtt{t}_1; \rho_1\}$ *holds that* $\mathtt{t}_1 \geq \mathtt{t}'$*.*

Proof. Theorem 1 derives from the property that the analysis does not depend on the timestamp in the precondition. For $\{\Gamma_1; \mathtt{t}_1; \rho_1\} S \{\Gamma_2; \mathtt{t}_2; \rho_2\}$, the duration $\mathtt{t}_2 - \mathtt{t}_1$ always over-estimates the duration of every possible real execution of the statement S. Theorem 1 is proved by induction on the energy-aware semantics and the energy analysis rules. The only source of any over-estimation are the rules (*aIf*) and (*aWhile*). The (*aIf*) rule computes a `max` between the final timestamps of then-branch and the else-branch. In the (*aWhile*) rule, the execution time of one iteration of the loop is over-estimated and multiplied by the loop bound, which is an over-estimation of the number of iterations of the loop. □

Over-estimating the component state is fundamental for over-estimating the total energy consumption. A larger component state requires more power and hence consumes more energy.

Theorem 2 (Component state over-estimation). *If* $\{\Gamma; \mathtt{t}; \rho\} S \{\Gamma_1; \mathtt{t}_1; \rho_1\}$ *and* $\langle S, \sigma, \Gamma, \mathtt{t}\rangle \Downarrow^s \langle \sigma', \Gamma', \mathtt{t}'\rangle$ *then* $\Gamma_1 \geq \Gamma'$*.*

Proof. Induction on the energy-aware semantics and the energy analysis rules, yields that the update function δ preserves the ordering on component states (see Sect. 3.2). □

Now, we can formulate and prove the main soundness theorem:

Theorem 3 (Soundness). *If* $\{\mathtt{t}; \Gamma; \rho\} S \{\mathtt{t}_1; \Gamma_1; \rho_1\}$ *and* $\langle S, \sigma, \Gamma, \mathtt{t}\rangle \Downarrow^s \langle \sigma', \Gamma', \mathtt{t}'\rangle$ *then* $\mathfrak{e}_{system}(\Gamma_1; \mathtt{t}_1) \geq \mathfrak{e}_{system}(\Gamma'; \mathtt{t}')$*.*

Proof. By induction on the energy-aware semantics and the energy analysis rules. Theorem 1 ensures that the final timestamp is an over-estimation of the actual time-consumption, hence the calculation of energy usage is based on an over-estimated period of time. Theorem 2 ensures (given that the analysis is non-decreasing with respect to component states, a larger input state means a larger output state) that we find the maximum state (including incidental energy-usage) that can result from an iteration of a loop body with the logic. This depends on the **wci** function determining the maximal initial state for any iteration. It follows, by the definition of $\mathfrak{e}_{system}$ that $\mathfrak{e}_{system}(\Gamma_1; \mathtt{t}_1) \geq \mathfrak{e}_{system}(\Gamma'; \mathtt{t}')$. The total energy consumption resulting from the analysis is larger than that of every possible execution of the analysed program. □

6 Conclusion and Future Work

We presented a hybrid, energy-aware Hoare logic for reasoning about energy consumption of systems controlled by software. The logic comes with an analysis which is proven to be sound with respect to the semantics. To our knowledge, our approach is the first attempt at bounding energy-consumption statically in a way which is parametric with respect to hardware models. This is a first step towards a hybrid approach to energy consumption analysis in which the software is analysed automatically together with the hardware it controls.

Future Work Many future research directions can be envisioned: e.g. providing an implementation of an automatic analysis for a real programming language[4], performing energy measurements for defining component models, modelling of software components and enabling the development of tools that can automatically derive energy consumption bounds for large systems, finding the most suited tool(s) to provide the right loop bounds and annotations for our analysis and study energy usage per time unit on systems that are always running, removing certain termination restrictions.

Acknowledgements We would like to thank the anonymous referees for their extensive feedback which helped us to considerably improve the paper.

References

1. Albers, S.: Energy-efficient algorithms. Commun. ACM **53**(5) (2010) 86–96
2. Saxe, E.: Power-efficient software. Commun. ACM **53**(2) (2010) 44–48
3. Ranganathan, P.: Recipe for efficiency: principles of power-aware computing. Commun. ACM **53**(4) (2010) 60–67
4. te Brinke, S., Malakuti, S., Bockisch, C., Bergmans, L., Akşit, M.: A design method for modular energy-aware software. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, New York, NY, USA, ACM (2013) 1180–1182
5. Cohen, M., Zhu, H.S., Senem, E.E., Liu, Y.D.: Energy types. SIGPLAN Not. **47**(10) (October 2012) 831–850
6. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: Enerj: approximate data types for safe and general low-power computation. SIGPLAN Not. **46**(6) (June 2011) 164–174
7. Zhurikhin, D., Belevantsev, A., Avetisyan, A., Batuzov, K., Lee, S.: Evaluating power aware optimizations within GCC compiler. In: GROW-2009: International Workshop on GCC Research Opportunities. (2009)
8. Gheorghita, S.V., Corporaal, H., Basten, T.: Iterative compilation for energy reduction. J. Embedded Comput. **1**(4) (2005) 509–520
9. Junior, M.N.O., Neto, S., Maciel, P.R.M., Lima, R.M.F., Ribeiro, A., Barreto, R.S., Tavares, E., Braga, F.: Analyzing software performance and energy consumption of embedded systems by probabilistic modeling: An approach based on coloured petri nets. In: ICATPN'06. (2006) 261–281

[4] A proof-of-concept implementation for our While language is described in [24] and available at http://resourceanalysis.cs.ru.nl/energy

10. Nogueira, B., Maciel, P., Tavares, E., Andrade, E., Massa, R., Callou, G., Ferraz, R.: A formal model for performance and energy evaluation of embedded systems. EURASIP J. Embedded Syst. (January 2011) 2:1–2:12
11. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In: FMCO'07. Volume 5382 of LNCS., Springer (2008) 113–133
12. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: POPL'11, ACM (2011) 357–370
13. Atkey, R.: Amortised resource analysis with separation logic. In: ESOP. LNCS 6012 (2010) 85–103
14. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.W., Momigliano, A.: A program logic for resources. Theor. Comput. Sci. **389**(3) (December 2007) 411–445
15. Jayaseelan, R., Mitra, T., Li, X.: Estimating the worst-case energy consumption of embedded software. In: RTAS'06, IEEE (2006) 81–90
16. Kerrison, S., Liqat, U., Georgiou, K., Mena, A.S., Grech, N., Lopez-Garcia, P., Eder, K., Hermenegildo, M.V.: Energy consumption analysis of programs based on XMOS ISA-level models. In: LOPSTR'13, Springer (September 2013)
17. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. **7**(3) (2008)
18. Shkaravska, O., van Eekelen, M.C.J.D., van Kesteren, R.: Polynomial size analysis of first-order shapely functions. Logical Methods in Comp. Sc.. **5**(2) (2009) 1–35
19. Shkaravska, O., Kersten, R., Van Eekelen, M.: Test-based inference of polynomial loop-bound functions. In: PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, ACM (2010) 99–108
20. Kersten, R., van Gastel, B.E., Shkaravska, O., Montenegro, M., van Eekelen, M.: ResAna: a resource analysis toolset for (real-time) JAVA. Concurrency Computat.: Pract. Exper. (2013)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Verification, Model Checking, and Abstract Interpretation. Volume 2937 of Lecture Notes in Computer Science. Springer (2004) 465–486
22. Hunt, J.J., Tonin, I., Siebert, F.: Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time java programs. In Bollella, G., Locke, C.D., eds.: JTRES. Volume 343 of ACM International Conference Proceeding Series., ACM (2008) 97–105
23. Parisen Toldin, P., Kersten, R., van Gastel, B., van Eekelen, M.: Soundness Proof for a Hoare Logic for Energy Consumption Analysis. Technical Report ICIS–R13009, Radboud University Nijmegen (October 2013)
24. Schoolderman, M., Neutelings, J., Kersten, R., van Eekelen, M.: ECAlogic: Hardware-parametric energy-consumption analysis of algorithms. In: FOAL '14, ACM (2014) 19–22