Towards Practical, Precise and Parametric Energy Analysis of IT Controlled Systems

Bernard van Gastel   Marko van Eekelen

Faculty of Management, Science and Technology,
Open University of the Netherlands, Heerlen, The Netherlands

Institute for Computing and Information Sciences,
Radboud University, Nijmegen, The Netherlands

{Bernard.vanGastel,Marko.vanEekelen}@ou.nl

Energy consumption analysis of IT-controlled systems can play a major role in minimising the overall energy consumption of such IT systems, during the development phase, or for optimisation in the field. Recently, a precise energy analysis was developed, with the property of being parametric in the hardware. In principle, this creates the opportunity to analyse which is the best software implementation for given hardware, or the other way around: choose the best hardware for a given algorithm.

The precise analysis was introduced for a very limited language: ECA. In this paper, several important steps are taken towards practical energy analysis. The ECA language is extended with common programming language features. The application domain is further explored, and threats to the validity are identified and discussed. Altogether, this constitutes an important step towards analysing energy consumption of IT-controlled systems in practice.

1 Introduction

Energy analysis of IT systems is an important emerging field. Its focus is on analysing the software that controls the IT system using models of the components of the system under analysis. Components can vary from small components such as a sensor in the Internet of Things to large subsystems as present in self-driving cars.

As traditionally many savings did occur on the hardware side of a computer, energy consumption is almost a blind spot when developing software. Each next hardware generation consumed less energy to perform the same amount of work. However, recently this development has lost its pace. At the same time, it becomes more and more clear that software has a huge impact on the behaviour and the properties of devices it runs on. A recent example of software influencing the working of a device is the Volkswagen scandal. The car manufacturer used software to detect if the car was being tested. If this was found to be the case, the diesel motor was programmed to operate in such a way that it exhausted less toxic gases and fumes. In [13] it is calculated that 44,000 years of human life are lost in Europe because of the fraud, which lasted at least six years. Another example is fridges from Panasonic, which could detect if a test was going on and suppressed energy intensive defrost cycles during this test. These are negative examples, but they do make clear that the software is in control of the device and its (energy) behaviour.

Although the software is evidently in control of the devices, there is almost no time dedicated in most computer science curricula to the energy efficiency of software. This is peculiar since energy is of vital importance to the modern (software) industry. For years, data centres have been located at places where the energy is cheap, and since the rise of the smartphone more software engineers recognise that to get good user reviews, their software should not rapidly deplete the battery charge of the user's phone. Due to this lack of educational attention to energy-aware programming, most aspiring programmers never learn to produce energy efficient code. Software engineers have trouble assessing how much energy will be consumed by their software on a target device, especially when the software is run on a multitude of different systems. With the advent of the internet of things, where software is increasingly embedded in our daily life, the *software industry should become aware of their energy footprint*, and methods must be developed to assist in reducing this footprint.

Furthermore, the combination of many individual negative effects can also affect our society at large. Although this effect is less direct, it is no less essential. If devices that are present in large quantities in our society all exhibit the same negative behaviour, such as incurring needlessly a too high energy consumption, they can impact public utilities and our economy and will consume the finite resources of Earth even faster. Governments increasingly recognise this societal effect, as indicated by the new laws in the European Union issuing ecodesign requirements for many kinds of devices. One of the aims of these requirements is to make devices more energy efficient. Examples of product categories with ecodesign requirements include vacuum cleaners, electrical motors, lighting, heaters, cooking appliances, televisions and coffee machines. Even requirements leading to relative small improvements in energy efficiency can yield large results at scale, even in the case of devices of which one would expect no significant electricity savings to be possible.

Modern devices and appliances are controlled by software, which makes analysing the energy consumption challenging of these devices, as the behaviour of its software is difficult to predict. To analyse the consumption of hardware, the software controlling the hardware needs to be analysed together with the hardware.

Our approach To this end, we proposed in [5] a hybrid approach, joining energy behaviour models of the hardware with the energy-aware semantics of software and a program transformation. The interface between hardware and software is made explicit and has to be well defined, allowing for exchanging of hardware or software components. Using this parametric approach, multiple implementations can be analysed. Such an approach can be used on design level or for optimisation. One can e.g. choose the best software implementation for given hardware, or the other way around: choose the best hardware for a given algorithm.

The described approach derives energy consumption functions. These energy functions signify the exact energy behaviour when the software is executing and controlling the modelled hardware. Hardware is modelled as a finite state machine, with both the states and transitions labelled with energy consumption. The programs that can be analysed are written in the software language ECA, which is an imperative language inspired by C and Java. Currently, only a limited set of program constructs is supported.

The most important contributions of this article are:

- extended support for common language features in the software to be analysed: adding types, data structures, global variables, and recursion;

- description of application domain of the ECA energy analysis method;
- identified threats to the validity of proposed approach and a discussion of how to deal with these threats.

Overview Section 2 introduces the ECA energy analysis. In section 3 extensions of the ECA language are defined including the new derivation rules that are needed for the analysis. Section 4 explores the application domain of ECA. The validity of the results of the analysis is discussed in section 5. Finally, we conclude with related work, future work and conclusions in sections 6 and 7.

2 Introduction to energy analysis with ECA

Energy analysis combines hardware modelling with the energy-aware semantics of software. To this end the language ECA is specified, on which our analysis is targeted. Based on this, a semantics of this language can be defined, which includes energy consumption. Using this energy-aware semantics, a program transformation is given. This transformation generates an executable model, using a parametric function. If both a concrete input and one or multiple hardware models are specified, the parametric function will result in the energy consumption occurring when running the software on the given hardware.

To illustrate this process, we start with the hardware modelling and continue with describing for some program constructs the semantic rules, the program transformations and the effect on hardware. This is an introductory overview. For further details, the reader is referred to [5, 6, 10].

The hardware conceptually consists of a *component state* and a set of *component functions* which operate on the component state. We use finite state models to model these hardware components, with the transitions constituting function calls on the hardware. Energy usage is expressed by labelling both the vertices and edges with energy consumption, which can be in any unit. Labels on vertices constitute time-bound energy consumption, i.e. power draw. Edges are labelled with the consumption of a certain amount of energy, not time-bound but corresponding to the transition. Depending on your needs, you can model energy consumption in one way or the other. Besides the ones above described, there are no additional requirements. We use the power draw function ϕ which translates a component state to a power draw. The result of this function is used to calculate energy consumption for the time spent in a specific state.

All transitions in a component model are explicit in the ECA source code. We use the notation $C::f$ to refer to a function `f()` operating on a component C. Multiple different hardware components can be used simaltaniously from the same program, the components are differentiated by a unique name (substituted in the rules for C). Besides this addition, the ECA language is a fairly default imperative language sporting functions, a single signed integer type, conditionals, looping constructs and expressions that can be used as statements.

Next, we move on to the semantics, for now without energy added. Besides the function environment Δ and the program state σ, we have the (hardware) component states Γ. This makes the effect on the hardware explicit. The effect of the component function $C::f$ is split into two: the effect function $\delta_{C::f}$ and a function $\mathrm{rv}_{C::f}$ calculating the return value of the component call. Both are working on the component

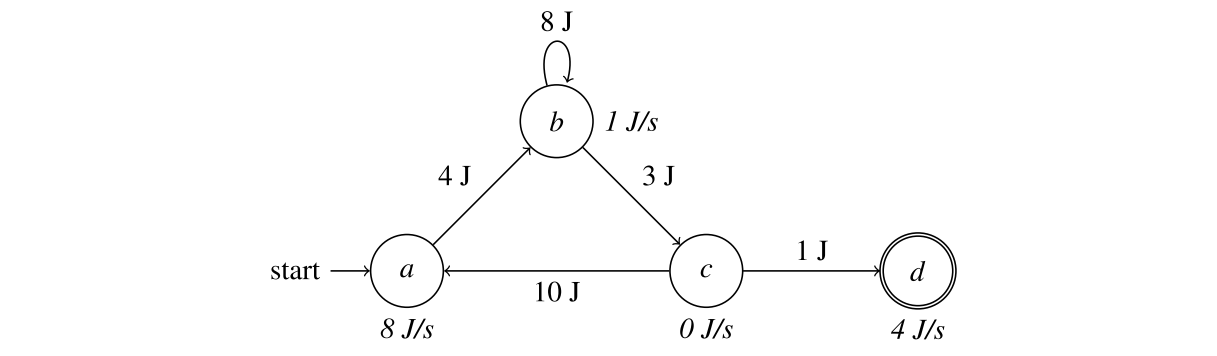

Figure 1: Hardware model with the energy consumption expressed in Joule.

state retrieved from Γ (by $\Gamma(C)$). The straightforward semantic definition of the component function *sCmp*, not taking energy into account, is given below.

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma'\rangle}{\Delta, C{::}f = (\delta_{C::f}, \mathrm{rv}_{C::f}) \vdash \langle C{::}f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle \mathrm{rv}_{C::f}(\Gamma'(C), a), \sigma', \Gamma'[C \leftarrow \delta_{C::f}(\Gamma'(C), a)]\rangle} \text{ (sCmp)}$$

Adding to it, the energy cost of a component function call consists of including the time taken to execute this function and the explicit energy cost attributed to this call resulting in rule *esCmp*.

$$\frac{\Delta;\Phi \vdash \langle e_1, \sigma, \Gamma\rangle \xrightarrow{e} \langle a, \sigma', \Gamma', E'\rangle}{\Delta, C{::}f = (\ldots, \ldots, t_f, E_f);\Phi \vdash \langle C{::}f(e_1), \sigma, \Gamma\rangle \xrightarrow{e} \langle \ldots, \ldots, \ldots, E' + \Phi(\Gamma')\cdot t_f + E_f\rangle} \text{ (esCmp)}$$

The approach works by transforming these semantic rules into higher order expressions. When executed on a concrete program state PState and component state CState, this expression yields the energy consumption (and new states and possible value of an ECA expression). Expressions from the ECA language are transformed into rules that result in a tuple of three elements: a value function V, a state update function Σ (for both program state and the hardware state), and an energy consumption function E. Statements from the ECA language are only transformed into the latter two.

These compositional expressions are composed with higher order combinators. One of these combinators is the composition operator $\ggg$, which first applies the left-hand side, and on the resulting state applies the right-hand side. Another is the $\overline{+}$ operator, which is a higher order addition operator: when executed, it calculates the energy consumptions of the two operands based on the input states and adds them together. To explain the component call rule *btCmp* and function call rule *btCall*, we need an operator for higher order scoping. This operator creates a new program environment but retains the component state. It can even update the component state given a Σ function, which is needed because this Σ needs to be evaluated using the original program state. The definition is as follows:

$$\begin{aligned}
&\overline{[x \mapsto V, \Sigma]} : \mathrm{Var} \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{Value}) \\
&\qquad \times (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\qquad \to (\mathrm{PState} \times \mathrm{CState} \to \mathrm{PState} \times \mathrm{CState}) \\
&\overline{[x \mapsto V, \Sigma]}(ps, cs) = ([x \mapsto V(ps, cs)], cs') \text{ where } (\_, cs') = \Sigma(ps, cs)
\end{aligned}$$

We also need an additional operator split, because the program state is isolated, but the component state is not. The split function forks the evaluation into two state update functions and joins the results together. The first argument defines the resulting program state; the second defines the resulting component state.

$$\frac{\Delta \vdash e : \langle V_{ex},\, \Sigma_{ex},\, \ldots \rangle}{\begin{array}{l}\Delta,\ C{::}f = (x_f, V_f, \Sigma_f, \ldots, \ldots) \vdash \\ \quad C{::}f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg V_f,\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \Sigma_f),\ \ldots \rangle\end{array}}\ (\mathrm{btCmp})$$

The environment Δ is extended for each component function $C :: f$ with two elements: an energy judgment E_f and a run-time t_f. Time independent energy usage can be encoded into this E_f function. For functions defined in the language, the derived energy judgement is inserted into the environment. Using the patterns described above the component function call is expressed as:

$$\frac{\Delta \vdash e : \langle V_{ex},\, \Sigma_{ex},\, E_{ex} \rangle}{\begin{array}{l}\Delta,\ C{::}f = (x_f, \ldots, \ldots, E_f, t_f) \vdash \\ \quad C{::}f(e) : \langle \ldots,\ \ldots,\ E_{ex} \mathbin{\overline{+}} (\Sigma_{ex} \ggg (\mathrm{td}^{ec}(t_f) \mathbin{\overline{+}} E_f)) \rangle\end{array}}\ (\mathrm{etCmp})$$

This concludes the short introduction to energy analysis with ECA. For a more thorough coverage, see [5, 6, 10].

3 Increasing the expressivity of ECA

To bridge the gap between practical programming languages and ECA, several extensions to ECA are introduced in this section: adding data structures and types, global variables and recursion.

3.1 Adding data structures and types

The only supported type in the ECA language was a signed integer. There were no explicit Booleans, floating point numbers or data structures. To add those, types of variables need to be supported. We need multiple modifications for this change: modifying the grammar, and adding type distinction to both the semantic environments and program transformations.

We consider variables to be passed *by-value*. Functions can have side effects on the components and, as introduced in section 3.2, on the global variables. Functions are statically scoped. *Recursion is now supported*, and the changes needed to the semantic rules and program transformations are discussed in section 3.3.

The extended BNF grammar for the ECA language is defined in listing 2. We presume there is a way to differentiate between identifiers that represent variables ⟨*var*⟩, function names ⟨*fun-name*⟩, components ⟨*component*⟩, and constants ⟨*const*⟩.

Functions on components can now have a variable number of arguments and optionally return a value (if not, the type void should be used). A constructor for data structures is included, with a syntax like a function call with as name the type of the data structure), and a construct to access fields of a data structure (with the `.` operator). A type checking phase is now needed to detect typing errors, like using a data structure as a condition in the `if` construct. Only correctly typed programs are considered. The language retains an explicit construct for operations on hardware components (e.g. memory, storage or network devices). The notation $C::f$ refers to a function `f()` operating on a component C. This allows us to reason about components in a straightforward manner.

```
⟨program⟩       ::= ⟨struct-def⟩ ⟨program⟩ | ⟨fun-def⟩ ⟨program⟩ | ⟨type⟩ ⟨var⟩ ‘=’ ⟨expr⟩ | ε

⟨struct-def⟩    ::= ‘struct’ ⟨struct-name⟩ ‘begin’ ⟨struct-fields⟩ ‘end’

⟨struct-fields⟩ ::= ⟨type⟩ ⟨field-name⟩ ‘;’ ⟨struct-fields⟩ | ε

⟨type⟩          ::= ‘void’ | ‘bool’ | ‘int’ | ‘float’ | ⟨struct-name⟩

⟨fun-def⟩       ::= ⟨type⟩ ⟨fun-name⟩ ‘(’ [⟨fun-args⟩] ‘)’ ‘begin’ ⟨expr⟩ ‘end’

⟨fun-args⟩      ::= ⟨type⟩ ⟨name⟩ ‘,’ ⟨fun-args⟩ | ⟨type⟩ ⟨name⟩

⟨bin-op⟩        ::= ‘+’ | ‘-’ | ‘*’ | ‘>’ | ‘>=’ | ‘==’ | ‘!=’ | ‘<=’ | ‘<’ | ‘and’ | ‘or’

⟨expr⟩          ::= ⟨const⟩ | ⟨var⟩ | ⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
                 |  ⟨struct-name⟩ ‘(’ ⟨args⟩ ‘)’
                 |  ⟨expr⟩ ‘.’ ⟨field-name⟩
                 |  ⟨type⟩ ⟨var⟩ ‘=’ ⟨expr⟩
                 |  ⟨var⟩ ‘=’ ⟨expr⟩
                 |  ⟨component⟩ ‘::’ ⟨fun-name⟩ ‘(’ [⟨args⟩] ‘)’
                 |  ⟨fun-name⟩ ‘(’ [⟨args⟩] ‘)’
                 |  ⟨stmt⟩ ‘,’ ⟨expr⟩

⟨args⟩          ::= ⟨expr⟩ ‘,’ ⟨args⟩ | ⟨expr⟩

⟨stmt⟩          ::= ‘skip’ | ⟨stmt⟩ ‘;’ ⟨stmt⟩ | ⟨expr⟩
                 |  ‘if’ ⟨expr⟩ ‘then’ ⟨stmt⟩ [‘else’ ⟨stmt⟩] ‘end’
                 |  ‘repeat’ ⟨expr⟩ ‘begin’ ⟨stmt⟩ ‘end’
                 |  ‘while’ ⟨expr⟩ ‘begin’ ⟨stmt⟩ ‘end’
```

Listing 2: Extended BNF grammar for the ECA language, with types and data structures added, as well as one construct in ⟨*program*⟩ for global variable support (see next section).

A typical (predictive recursive descent) parser of this extended language is in the LL(2) class of parsers, with a small second pass. This second pass is needed, to avoid a possible infinite lookahead that is needed to differentiate between expressions and statements. During the first phase expressions and statements are combined into one construct. The small post-processing step differentiates between the two. In this way, the language can still be efficiently parsed in a simple manner.

Next are the adjustments to the semantic rules. Because a type checking phase was added, no typing error can occur when applying the semantic rules. Although the meaning differs, the syntax of the rules remains largely the same. Likewise, we adjust the program transformation rules. To support the new grammar rules, we add additional rules to the existing body of rules. Below is the rule for field access on a variable listed.

$$\frac{\Delta;\Phi \vdash \langle e,\, \sigma,\, \Gamma\rangle \xrightarrow{e} \langle v,\, \sigma',\, \Gamma',\, E'\rangle}{\Delta;\Phi \vdash \langle e\,.\,a,\, \sigma,\, \Gamma\rangle \xrightarrow{e} \langle v(a),\, \sigma',\, \Gamma',\, E' + \Phi(\Gamma')\cdot t_{\text{fieldaccess}}\rangle}\ (\text{esField})$$

3.2 Global variables

Control software often works with global variables. To support analysis of this control software we consequently need support in our ECA language for global variables. Hardware components are already handled as global state. To also support global variables, we need to introduce an additional global program state environment in addition to the local program state environment as it is currently used.

In the semantics, an additional global program state G is added to all tuples in every rule. Lookups are first performed in the local scope, the already existing σ. Although for scoping a layered program state can be preferred, or one based on indirections, we use a different approach. Because of special handling of the, by definition global, hardware components, the global program state is handled in the same manner as the global hardware component states. We add rules in the semantics for global variable definitions. The assignment rule is split depending if you assign a global or local variable. The variable loop is adjusted to first look in the local scope and if nothing is found, continue in the global scope. We start with introducing the global assignment rule.

$$\frac{\Delta;\Phi \vdash \langle e_1,\, \sigma,\, G,\, \Gamma\rangle \xrightarrow{e} \langle n,\, \sigma',\, G',\, \Gamma',\, E'\rangle}{\Delta;\Phi \vdash \langle x := e_1,\, \sigma,\, G,\, \Gamma\rangle \xrightarrow{e} \langle n,\, \sigma',\, G'[x \leftarrow n],\, \Gamma',\, E' + \Phi(\Gamma')\cdot t_{\text{assign}}\rangle}\ (\text{esGlobAssign})$$

Next, we adjust the variable lookup rule, with a $\cup$ defined on two program environments. This $\cup$ creates one environment, according to the scoping rules. It a variable is defined in both, the left-hand argument to the $\cup$ is normative.

$$\frac{}{\Delta;\Phi \vdash \langle x,\, \sigma,\, G\rangle\Gamma \xrightarrow{e} \langle(\sigma \cup G)(x),\, \sigma,\, G,\, \Gamma,\, \Phi(\Gamma)\cdot t_{\text{var}}\rangle}\ (\text{esVar})$$

For the program transformation rules, more extensive changes are needed. We extend the (global) component states CState with a global program state of type PState. For clarity of presentation, we introduce the type GState (global state) which is a combination of PState $\times$ CState. In this way, only the higher order combinators need to be changed, as the rules themselves remain intact. Most changes are only to the signature of those combinators. As a result, these combinators retain their compositional properties. As an example, the higher order scoping rule of section 2 is redefined below.

$$\begin{aligned}
&\overline{[x \mapsto V, \Sigma]} : \text{Var} \times (\text{PState} \times \text{GState} \rightarrow \text{Value}) \\
&\qquad \times (\text{PState} \times \text{GState} \rightarrow \text{PState} \times \text{GState}) \\
&\qquad \rightarrow (\text{PState} \times \text{GState} \rightarrow \text{PState} \times \text{GState}) \\
&\overline{[x \mapsto V, \Sigma]}(ps, gs) = ([x \mapsto V(ps, gs)], gs') \text{ where } (\_, gs') = \Sigma(ps, gs)
\end{aligned}$$

The lookup function that the variable lookup rule depends on is redefined as follows:

$$\begin{aligned}
&\text{lookup}_x : \text{PState} \times \text{GState} \rightarrow \text{Value} \\
&\text{lookup}_x(ps, gs) = ps(x) && \text{if } x \text{ exists in local program scope } ps \\
&\text{lookup}_x(ps, gs) = \text{Variables}(gs)(x) && \text{if } x \text{ exists in the global variables part of } gs
\end{aligned}$$

3.3 Adding recursion

We can define the function call in a similar way as the component call, which was introduced in section 2. However, to support recursion, a special subst higher order function is introduced to unfold the function definition once, just before it is executed on a concrete environment. The subst is defined as follows, with PState signifying a program state, GState signifying the global state (extended in section 3.2 to be both the component states and the global variables) and T a type variable (depending on whether a state update or a value function is substituted):

$$\begin{aligned}
&\text{subst} : (\text{PState} \times \text{GState} \rightarrow T) \\
&\qquad \times (\text{PState} \times \text{GState} \rightarrow T) \\
&\qquad \rightarrow (\text{PState} \times \text{GState} \rightarrow T) \\
&\text{subst}(T, R)(ps, gs) = (T[R \leftarrow \text{subst}(T, R)])(ps, gs)
\end{aligned}$$

A recursive call is represented by the abstract higher-order function rec, which is a placeholder for applying substitution on. There are multiple variants, depending on the resulting type, with the rec_V one for a resulting value function, and the rec_Σ one for a resulting state update function.

$$\begin{aligned}
&\text{rec}_V : \text{PState} \times \text{GState} \rightarrow \text{Value} \\
&\text{rec}_\Sigma : \text{PState} \times \text{GState} \rightarrow \text{PState} \times \text{GState}
\end{aligned}$$

If there is a function body B computing a Value with for example rec_V in it, the value of the recursive function can be computed by executing $\text{subst}(B, \text{rec}_V)$. As long as the original function terminates on the given input environment, this analysis will terminate on the same input. This is the essential difference from the *btCmp* rule, as can be seen in the definition of *btCall* below:

$$\frac{\Delta, f = (x_f, V_f, \Sigma_f) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{aligned}\Delta^v, f = (x_f, V_f, \Sigma_f) \vdash f(e) : \langle &\overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} >\!\!>\!\!> \text{subst}(V_f, \text{rec}_V(f)), \\ &\text{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} >\!\!>\!\!> \text{subst}(\Sigma_f, \text{rec}_\Sigma(f))) \rangle\end{aligned}} \text{(btCall)}$$

For each language function, a definition is placed in Δ^v using the *btFuncDef* rule. The body of the function is analysed, and recursive calls to the function are replaced with rec placeholders using the *btRec* rule. To support this, the function definition rule inserts a special definition in the function environment Δ^v, on which the *btRec* rule works. This leads to the following definition of *btFuncDef*, with P the remaining program definition:

$$\frac{\Delta^v, f = (x) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle \qquad \Delta^v, f = (x, V_{ex}, \Sigma_{ex}) \vdash P : \Sigma_{st}}{\Delta^v \vdash \mathsf{function}\ f(x)\ \mathsf{begin}\ e\ \mathsf{end}\ P : \Sigma_{st}} \text{(btFuncDef)}$$

The placeholders are inserted using the *btRec* rule. This rule analyses the expression used as the argument, like the component and function call rules do. The definition is in fact very similar to those definitions:

$$\frac{\Delta^v, f = (x_f) \vdash e : \langle V_{ex}, \Sigma_{ex} \rangle}{\begin{array}{l}\Delta^v, f = (x_f) \vdash \\ \quad f(e) : \langle \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_V(f),\ \mathrm{split}(\Sigma_{ex}, \overline{[x_f \mapsto V_{ex}, \Sigma_{ex}]} \ggg \mathrm{rec}_\Sigma(f)) \rangle\end{array}} \text{(btRec)}$$

4 Exploring the application domain of ECA

The foreseen application area of the proposed analysis is in predicting the energy consumption of control systems, where software controls peripherals. This includes control systems in factories, cars, aeroplanes, smart-home applications, etc. Examples of hardware components range from heaters to engines, motors and urban lighting. Depending on the target device energy consumption can be electricity, gas, water, or any other resources where the consumption increases monotonically. The proposed analysis can predict the energy consumption of multiple algorithms and different hardware configurations. The choice of algorithm or configuration may depend on the expected workload. This makes the proposed technique useful for both programmers and operators. Below, we discuss the application domain of ECA in a way which is partly and informally published in the lecture notes for the TACLe PhD summer school in 2016 in Yspertal, Austria [7].

The possibility to abstract from the actual hardware specification makes the proposed approach still applicable even when no final hardware component is available for basing the hardware model on, or when such a model is not yet created. We observe that many decisions are based on relative properties between systems. Abstracting hardware models can be used to focus e.g. on the relative differences between component methods and component states.

Compared to the Hoare logic in [10], many restrictions are not present in ECA. Foremost, this type system does not have the limitation that state change cannot depend on the argument of a component function nor that the return value of a component function cannot depend on the state of the component. More realistic models can, therefore, be used. This widens the number of applications, as behaviour of hardware can be modelled that previously could not be expressed in the modelling.

However, there are still certain properties hardware models must satisfy for ECA to be applicable. Foremost, the models have to be discrete. Energy consumption that gradually increases or decreases over time can therefore not be modelled directly. However, discrete approximations may be used. Secondly, every state change has to be the consequence of an explicit application of a component function. So, implicit state changes by hardware components cannot be expressed.

The quality of the derived energy expressions is directly related to the quality of the used hardware models. Depending on the goal, it is possible to use multiple models for one and the same hardware component. For instance, if the hardware model is constructed as a worst-case model, this approach will produce worst-case information. Similarly one can use average-case models to derive average case information.

It can be difficult to obtain detailed hardware models, sometimes for the simple reason that the hardware is yet to be developed and not ready. We expect that, in cases where multiple software implementations are to be compared, the relative consumption information will be sufficient to support design decisions. This allows for constructing abstract models, with not much detail but including the relevant information which is needed to make a proper comparison. However, this abstraction could impact the validity of the results (as a realistic model could yield different results).

A class of applications where the approach described in this article could be useful is a company that produces many variations of the same device. Variations occur based on local requirements, or on regional differences in the electric grid, or on different requirements set by integrators or consumers, or any combination thereof. The ECA approach allows for quickly designing those variations, and having a clear view on how changes will impact all those variations.

5 Validity

The analysis is sound and complete as can be proven by induction on the syntactic structure of the program in a similar way as in an earlier version of our analysis described in [14], in which the proof is more complex due to the presence of approximations. An analysis method may be in itself sound and complete but the validity of applying the method in practice can not automatically be inferred from that.

There are several validity constraints to the technique as it is discussed in this article. The quality of the results depends directly on the quality of the component models used. There are severe restrictions on component models, e.g. the power draw is assumed to be constant in every state of the component. This is in practice not true for most devices, e.g. the power draw can be a function of time. This has to be modelled in an abstract discretised component model. It is to be seen whether with discrete approximations for real world hardware component models can be created with a level of precision that is suited to make accurate energy estimates. It is hard to construct and validation such component models on the right level of abstraction, as there are several real world practical issues. If basing the component model on specifications from hardware vendors, all kinds of errors in the specification are transferred to the component model. Production errors in the hardware, and eventually the degradation of hardware, can induce erratic energy consumption behaviour that does not conform to the specification/component model.

Validating a component model with a test setup is hard, as energy is hard to measure. Small differences in energy consumption are hard to measure correctly, and outside conditions like temperature can influence the results greatly. Energy differs significantly from other kinds of resources (e.g. memory and time), which are measurable with great precision within a computer by the computer itself. Introducing a standard energy consumption measurement interval might help in making measurements more uniform. Validating if the number of states of a component model is the same as the actual number of states of an actual hardware component, is a hard problem by itself. With powerful models, the actual validation process with real hardware might just take too much time forcing the user to settle for a feasible but not fully validated model.

There is another potential source of not matching the actual energy consumption of a realistic situation. Compiler errors and optimisations can impact the (energy) behaviour of a source program greatly. The compiler has influence on the timing of high-level language constructs. The timing constants used for these language constructs should match the time it takes to execute those language constructs. Such a match could be guaranteed by creating a resource consumption certified compiler in a similar way as was done in the CompCert certified compiler project [11]. Of course this would require the availability of energy aware semantics both on the source and the target level. An even more complicating matter may be the complex design of modern processors executing the software. Even relatively small embedded microprocessors have features (register bypass e.g.), which impact the execution timing of statements significantly. Proper documentation of such features may be hard to find since e.g. the inner details of the pipeline of modern CPU's can often not be found in the documentation.

These constraints on modelling hardware components and validity implications should be lifted and further investigated to make the technique discussed applicable to general, real-world problems. However, depending on the context and the precision needed, the current technique can already be applicable now. If the hardware component is relatively simple, a suited component model can be constructed. Another valid area for the techniques discussed is to give feedback to a prospective programmer, such that during construction of software the developer can optimise the energy consumption for various hardware configurations.

6 Related work and future work

A few options are available to a programmer who wants to write energy-efficient code. The programmer can look for programming guidelines and design patterns, which in most cases produce more energy-efficient programs, e.g. [15, 3]. Then, he/she might make use of a compiler that optimises for energy-efficiency, e.g. [20]. If the programmer is lucky, there is an energy analysis available for the specific platform at hand, such as [9] in which the energy consumption of a processor is modelled in SimpleScalar.

However, for most platforms, this is not a viable option. In that case, the programmer might use dynamic analysis with a measurement set-up. This, however, is not a trivial task and requires a complex set-up [8, 4]. Moreover, it only yields information for a specific benchmark [12]. Nevertheless, these approaches are always applicable. A programmer might, however, prefer an approach that yields additional insight in a more predictive manner.

In future work, we aim to fully implement this precise analysis to evaluate its suitability for larger systems and further explore practical applicability. We intend to experiment with additional implementations of various derived approximating analyses to evaluate which techniques/approximations work best in which context.

A current limitation of the analysis is that it allows only *one control process* (processor). Actual systems often consist of a network of interacting systems. Therefore, incorporating interacting systems would increase the applicability of the approach. Such systems can be seen as hybrid automata. Theoretical and practical results in modelling hybrid automata [1, 2] might be a useful starting point for further research.

To make the presentation more concise, it might be useful to use as a subject language a first-order strict-evaluation functional programming language. One can expect that this will alleviates the need to have a separate basic dependent type system which transforms all variables into expressions over input variables. However, expressions would still need to be expressed in terms over input variables. To support the analysis of data types, a size analysis of data types might be useful to enable iteration over data structures, e.g. using techniques similar to [16, 17].

On the language level, the type system is precise. However, it does not take into account optimisations and transformations below the language level. This can be achieved by analysing the software on a lower level, for example, the intermediate representation of a modern compiler, or even the binary level. For increased accuracy, this may certainly be worthwhile. Another motivation to use such an intermediate representation as the language that is analysed is the ability to support (combinations of) many higher level languages. In this way, programs written in and consisting of multiple languages can be analysed. It can also account for optimisations (such as common subexpression elimination, inlining, statically evaluating expressions), which in general reduce the execution time of the program and therefore impact the time-dependent energy usage (calls with side effects like component function calls are not optimised).

Compared with [10], the ECA energy consumption in this paper is precise instead of over-approximated. Future research can show if the expressions derived by this type system can be transformed in such a way that also upper bound expressions are derived. To support this, recursion and loops should be transformed in a Cost Relation System (CRS), a special case of recurrence relations. Solving this CRS, one can acquire a direct formula expressing the energy consumption of the recursive function or loop.

Another approach to providing more precise estimates is described in [19]. In this approach suitable program inputs are identified through which, by measurement, more precise results can be achieved in combination with an auxiliary energy model taking into account the energy consumption of instructions in relation to each other.

Finally, a systematic approach to constructing component models should be looked into. One can create a model from the specifications given by the vendor. Another way is using model learning [18] techniques, which create a finite state model from black box testing and measuring. All the states should have a time dependent energy consumption assigned to them, and all the transitions should be assigned incidental energy consumption. Such a model can then be used as a component model.

7 Conclusion

Energy analysis consists of a hybrid approach in analysing both hardware and software together, to derive energy consumptions when executing the software on the hardware. This can be used during the development phase, or for optimisation. One can e.g. choose the best software implementation for given hardware, or the other way around: choose the best hardware for a given algorithm. The key to analysing larger systems is compositionality. Many programs encountered in the real world feature language constructs such as global variables and recursion. To analyse these programs, support in the ECA language and program transformations is required. This article extends ECA with language support for multiple types, support for recursive functions and global variables. Important properties are retained: it remains a composable, precise, and parametric energy analysis.

Furthermore, to gain additional insights in the feasibility of the approach, the article explores the (im)possibilities of using the approach in practice by discussing validity and applications.

All in all, this constitutes a practical step towards the application of the proposed energy analysis on real world problems.

References

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger & Pei-Hsin Ho (1992): *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems*. In Robert L. Grossman, Anil Nerode, Anders P. Ravn & Hans Rischel, editors: *Hybrid Systems*, *Lecture Notes in Computer Science* 736, Springer, pp. 209–229, doi:10.1007/3-540-57318-6_30.

[2] Rajeev Alur, Salar Moarref & Ufuk Topcu (2016): *Compositional Synthesis with Parametric Reactive Controllers*. In Alessandro Abate & Georgios E. Fainekos, editors: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*, ACM, pp. 215–224, doi:10.1145/2883817.2883842.

[3] Steven te Brinke, Somayeh Malakuti, Christoph Bockisch, Lodewijk Bergmans & Mehmet Aksit (2013): *A design method for modular energy-aware software*. In Sung Y. Shin & José Carlos Maldonado, editors: *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, ACM, pp. 1180–1182, doi:10.1145/2480362.2480584.

[4] Miguel A Ferreira, Eric Hoekstra, Bo Merkus, Bram Visser & Joost Visser (2013): *SEFLab: A lab for measuring software energy footprints*. In: *2nd International Workshop on Green and Sustainable Software (GREENS), 2013*, IEEE, pp. 30–37, doi:10.1109/GREENS.2013.6606419.

[5] Bernard van Gastel, Rody Kersten & Marko C. J. D. van Eekelen (2015): *Using Dependent Types to Define Energy Augmented Semantics of Programs*. In Marko C. J. D. van Eekelen & Ugo Dal Lago, editors: *Foundational and Practical Aspects of Resource Analysis - 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015, Revised Selected Papers*, *Lecture Notes in Computer Science* 9964, pp. 20–39, doi:10.1007/978-3-319-46559-3_2.

[6] Bernard van Gastel (2016): *Assessing sustainability of software; analysing correctness, memory and energy consumption*. Ph.D. thesis, Open Universiteit. Available at `http://sustainablesoftware.info/download/thesis-met-cover.pdf`.

[7] Bernard van Gastel & Marko van Eekelen (2016): *Lecture notes on 'Analysing energy consumption by software', for the TACLe Summerschool 2016 in Yspertal, Austria*. Technical Report, Radboud University Nijmegen. Available at `https://pms.cs.ru.nl/iris-diglib/src/getContent.php?id=2016-van-Eekelen-EnergyAnalysing`.

[8] Erik A. Jagroep, Jan Martijn van der Werf, Sjaak Brinkkemper, Giuseppe Procaccianti, Patricia Lago, Leen Blom & Rob van Vliet (2016): *Software Energy Profiling: Comparing Releases of a Software Product*. In: *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, ACM, New York, NY, USA, pp. 523–532, doi:10.1145/2889160.2889216.

[9] Ramkumar Jayaseelan, Tulika Mitra & Xianfeng Li (2006): *Estimating the Worst-Case Energy Consumption of Embedded Software*. In: *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, pp. 81–90, doi:10.1109/RTAS.2006.17.

[10] Rody Kersten, Paolo Parisen Toldin, Bernard van Gastel & Marko van Eekelen (2014): *A Hoare Logic for Energy Consumption Analysis*. In: *Proceedings of the Third International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'13)*, *LNCS* 8552, Springer, pp. 93–109, doi:10.1007/978-3-319-12466-7_6.

[11] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814. Available at `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`.

[12] F. A. Moghaddam, T. Geenen, P. Lago & P. Grosso (2015): *A user perspective on energy profiling tools in large scale computing environments*. In: *Sustainable Internet and ICT for Sustainability (SustainIT), 2015*, pp. 1–5, doi:10.1109/SustainIT.2015.7101364.

[13] Rik Oldenkamp, Rosalie van Zelm & Mark A.J. Huijbregts (2016): *Valuing the human health damage caused by the fraud of Volkswagen*. *Environmental Pollution* 212, pp. 121 – 127, doi:10.1016/j.envpol.2016.01.053. Available at `http://www.sciencedirect.com/science/article/pii/S0269749116300537`.

[14] Paolo Parisen Toldin, Rody Kersten, Bernard van Gastel & Marko van Eekelen (2013): *Soundness proof for a Hoare logic for energy consumption analysis*. Technical Report ICIS–R13009, Radboud University Nijmegen. Available at `https://pms.cs.ru.nl/iris-diglib/src/getContent.php?id=2013-ParisenToldin-SoundnessLogic`.

[15] Eric Saxe (2010): *Power-efficient software*. *Communications of the ACM* 53(2), pp. 44–48, doi:10.1145/1646353.1646370.

[16] Olha Shkaravska, Marko C. J. D. van Eekelen & Alejandro Tamalet (2013): *Collected Size Semantics for Strict Functional Programs over General Polymorphic Lists*. In Ugo Dal Lago & Ricardo Peña, editors: *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8552, Springer, pp. 143–159, doi:10.1007/978-3-319-12466-7_9.

[17] Alejandro Tamalet, Olha Shkaravska & Marko C.J.D. van Eekelen (2009): *Size Analysis of Algebraic Data Types*. In Peter Achten, Pieter Koopman & Marco T. Morazán, editors: *Trends in Functional Programming*, *Trends in Functional Programming* 9, Intellect, pp. 33–48. Available at `http://www.intellectbooks.co.uk/books/view-Book,id=4648/`. ISBN 978-1-84150-277-9.

[18] Frits W. Vaandrager (2017): *Model learning*. *Commun. ACM* 60(2), pp. 86–95, doi:10.1145/2967606.

[19] Peter Wagemann, Tobias Distler, Timo Honig, Heiko Janker, Rudiger Kapitza & Wolfgang Schroder-Preikschat (2015): *Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems*. *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)* 00, pp. 105–114, doi:10.1109/ECRTS.2015.17.

[20] Dmitry Zhurikhin, Andrey Belevantsev, Arutyun Avetisyan, Kirill Batuzov & Semun Lee (2009): *Evaluating power aware optimizations within GCC compiler*. In: *GROW-2009: International Workshop on GCC Research Opportunities*, pp. 1–9, doi:10.1.1.470.8078.