Skylines for Symbolic Energy Consumption Analysis

Markus Klinik[1,⋆], Bernard van Gastel[1,2], Cynthia Kop[1], and Marko van Eekelen[1,2]

[1] Radboud University, Nijmegen, the Netherlands
{M.Klinik,B.vanGastel,C.Kop,Marko}@cs.ru.nl
[2] Open University, Heerlen, the Netherlands
{Bernard.vanGastel,Marko.vanEekelen}@ou.nl

Abstract. Energy consumption in embedded systems plays a large role as it has implications for the power supply and the batteries used. Programmers of these systems should consider how their programs control external devices, and where energy consumption hotspots lie. We present a static analysis to predict and visualize energy consumption of external devices controlled by programs written in a simple imperative programming language. Currently available energy consumption analysis techniques generate graphs over time, which makes it difficult to see from where in the source code the consumption originates. Our method generates graphs over source locations, called *skyline diagrams*, showing the maximum power draw for each line of source code.
Our method harnessess symbolic execution extended with support for controlling external devices. This gives accurate predictions and complete code path coverage, as far as the limits of computability allow. To make the diagrams easier to understand, we introduce a merge algorithm that condenses all skylines into a concise overview. We demonstrate the potential by analysing various example programs with our prototype implementation. We envision this approach being used to identify energy consumption hotspots of embedded systems during the design and development phase, in a less involved way than traditional approaches.

Keywords: Symbolic execution · Program analysis · Energy use.

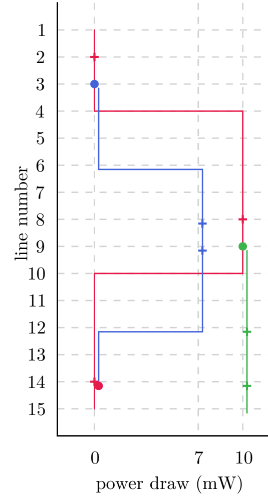
1 Introduction

Software that controls hardware is found in many places, such as washing machines, smartphones, or self-driving cars. The software running in such devices is in charge of orchestrating the hardware components, like sensors, motors, displays, or radios. Formal analysis of such devices is hard, because hardware and software have to be analysed together. In order to optimize energy consumption of such devices, especially when they are battery-powered, it is useful for programmers to have a prediction of energy-behaviour of all the components when

⋆ Corresponding author.

```
 1  int main() {
 2    x = SENS.readTemp();
 3    if( x ≤ 10 ) {
 4      LED1.switchOn();
 5    } else {
 6      LED2.switchOn();
 7    }
 8    sleep(100);
 9    if( x < 10 ) {
10      LED1.switchOff();
11    } else {
12      LED2.switchOff();
13    }
14    return 0;
15  }
```

(a)



(b)

Fig. 1. (a) Sensor input controls a lamp. (b) All possible runs of the program. One run does not end at power draw zero, which indicates a bug.

running their program. Simulation or actual measurement of running devices can give some insight, but only for one specific scenario and hardware configuration.

We develop a static analysis based on symbolic execution that can visualize the energy behaviour of all possible executions of a program at once. This allows programmers to quickly assess the energy impact of a change, already during development. Our method is parametrized with hardware models, so that programmers can swap components and explore different hardware configurations.

In the domain of embedded systems and control software, the energy use of the processor is sometimes negligible. We therefore limit our scope to the energy use of the hardware components controlled by the software. If desired, programmers can bypass this restriction by modelling the processor as a hardware component and switching between its power states explicitly with corresponding component calls. Modelling processors as a hardware components is possible, because they often have approximately constant energy consumption. For example the popular ATmega328P, used on the Arduino UNO, has an amperage of 0.2 mA in Active Mode, 0.75 μA in Power-save Mode, and 0.1 μA in Power-down Mode [21].

We illustrate our approach with an example. The program in fig. 1a reads a sensor value, and switches on either LED1 or LED2. It has a bug in line 9, where $<$ is used instead of $\leq$. There are three possible executions, one of which does not end with a power draw of zero, as there exists a sensor value where LED1 is not switched off. The skyline diagram in fig. 1b shows a merged view of the three executions. The horizontal axis shows power draw, and the vertical

axis line numbers. Skylines that would be drawn on top of each other are shifted by a small offset. In this view, programmers can see that some component still consumes energy at the end of the function, and can start investigating the issue.

This paper brings together two distinct lines of earlier work: the energy consumption analysis by van Gastel et al. [10] and the skyline diagrams by Klinik et al. [20]. Our contribution is threefold. First, we introduce a symbolic execution engine that tracks hardware state and works with the programming language SECA (Symbolic Energy Consumption Analysis). Second, we define visualization rules for the results of the symbolic execution as diagrams of power draw over points in the source code. Third, we define an algorithm to reduce the number of plotted graphs, hiding redundant information, to make the diagrams more concise. Our proof-of-concept implementation is available online [3].

Remark. Our goal is to explore the idea of drawing energy skylines over source lines; not (yet) to make an industry-ready tool. To focus on this goal, our method considers a C-like language that lacks the complexity of C itself. Likewise, the well-known problem of exponential state-space explosion, and reduction techniques that may be used to manage this problem, is not included in our scope. Thus, we do not currently consider programs with thousands of lines.

2 Methodology

Given a program in the SECA language (section 3), our system performs symbolic execution to examine all possible execution paths. For each path, the symbolic execution engine tracks the power draw of all components that the program controls, resulting in a graph that relates program points to energy consumption (section 4). We call such graphs *skylines.* The result of symbolic execution is a set of skylines for every function, considering all calls to a function, across all execution paths. Our system then condenses these skylines into a summary of the energy behaviour of the program by *merging* common segments, to emphasize where skylines differ (section 5). The merged skylines are rendered as *skyline diagrams*, with line numbers on the vertical axis and power draw on the horizontal axis. The paper ends with an analysis of a real-world example (section 6), a discussion of related work (section 7) and ideas for future work (section 8).

Control software. Our domain is control software, whose main purpose is to control hardware components like sensors or motors. It runs on embedded systems using low powered microprocessors, which have a negligable energy use compared to the software-controlled hardware. Control software has two key characteristics. First, it has low algorithmic complexity. We aim to analyse programs that, for example, regulate a central heating installation, not those that calculate square roots. Second, it contains statements that interact with hardware components. These component calls are the focus of our system, as we seek to find how their invocation influences the energy behaviour of the program. If programs have parts with high algorithmic complexity, which would overextend

the capabilities of symbolic execution, such parts could be hidden in library calls and left out of the analysis.

SECA represents the behaviour of hardware components in a model similar to the one in [10]; essentially a labelled transition system where every state has a power draw, and state changes can only be initiated by the code.

Energy consumption. Resources other than power draw can also be modelled, as long as they can be summed up. The analysis does not care; it sees resource consumption as a unitless number. We assume that components have rectangular power profiles, which means there is no ramp-up when switching them on.

Symbolic execution. Symbolic execution [18] is a program semantics that traces all possible program execution paths. Whenever a program asks for input, for example from a sensor or terminal, a symbolic input variable any_i is created. When conditionals are encountered, execution splits into two paths: one for evaluating the condition to true, one to false. Each path is coupled with the constraint on the symbolic inputs that must hold for this path to be followed.

To illustrate the idea, consider the program: `x = TEMP.readInt(); if (x <5) { y=7; } else { y=2*x+1; }`. Symbolic execution results in two paths, one through the then- and one through the else-branch. The first one terminates with global state $[x \mapsto any_0, y \mapsto 7]$ and path constraint $any_0 < 5$. The second one terminates with $[x \mapsto any_0, y \mapsto 2any_0{+}1]$ and path constraint $\neg(any_0 < 5)$.

Path constraints can be given to an SMT (satisfiability modulo theories) solver, to prune infeasible paths, and calculate example values for the *any*s.

Symbolic execution does not terminate if there is a path that loops indefinitely. To bypass this problem, our system exits loops after a pre-defined number of iterations, and generates a warning. In such situations there could be paths whose energy usage is not reported, and hence the analysis is unsound. However, due to the nature of symbolic execution, all possible energy behaviours of a loop will often be discovered in less iterations than what is needed for the behaviours to occur in concrete execution. We expect situations with missed energy behaviours to be uncommon in typical programs.

Program points. In previous work [20] we analysed resource consumption over time. This has several advantages, but does not clearly show which parts of the program contribute to which parts of a skyline. Here, we give up the time aspect and instead relate resource use directly to lines in the source code. This requires certain coding conventions; for example, there may be only one non-trivial expression or statement per line, and closing braces must be on their own line. The results are diagrams with a natural control flow from top to bottom with occasional jumps, which clearly relate parts of the program to their energy consumption. Consider for example the program in fig. 2a. Symbolic execution results in the two skylines in fig. 2b, which show the hotspots in lines 5 and 8.

Merging. Symbolic execution results in a set of skylines, one for every execution path. These skylines often have identical parts, only differing after or up to a certain point. Sometimes a skyline is equal to a second one for a few lines and then becomes equal to a third. This effect is common in loops, where a piece of code is executed repeatedly. The program in fig. 2a has two execution paths

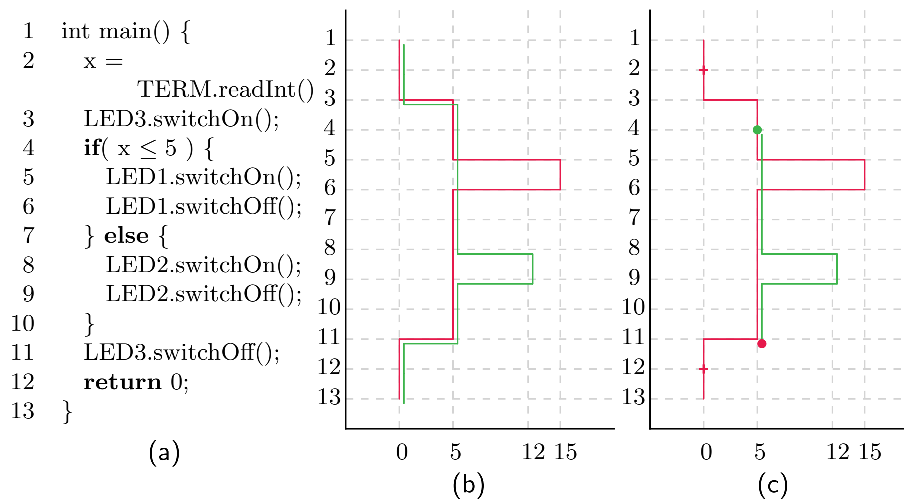


Fig. 2. (a) A program with two execution paths (b) Its skylines (c) Its merged skylines

that only differ during the execution of the conditional (lines 4–10). Figure 2c shows its skylines after merging. Until line 4 they are drawn as a single skyline. At line 4 is a *split point*, after which they are drawn separately. At line 11 they come together again, and continue so until the end of the function.

Our system aims to give programmers an idea of the energy behaviour of their programs, so that they gain insight where the hotspots lie. We argue that for this goal it is not required that skyline diagrams convey all information about all runs. Instead, we condense information such that unexpected spikes can be clearly identified. This comes at the cost of information loss about the exact number of runs, and losing the ability to fully trace individual runs.

3 The SECA Language

SECA (Symbolic Energy Consumption Analysis) is a small imperative programming language. We designed SECA to look like a simple form of C, without features like raw memory access and pointer arithmetic. Such features complicate the analysis and are not the focus of this paper. We believe that with some engineering effort, the analysis can be extended to support the style of C programs common in embedded and safety-critical systems. SECA is a variant of ECA [11], which is itself a variant of Nielson's **While** [22]. SECA programs can control external hardware through *component calls*. For example, the component call LED1.switchOn() invokes the switchOn functionality of the component LED1. Component calls can perform I/O and have return values, but no arguments. While it would be simple to allow arguments to the component calls for the *concrete* component models, it would require signifcant changes to the

$$
\begin{aligned}
e &::= \mathsf{true} \mid \mathsf{false} \mid i \mid x \mid e \ op\ e \\
&\quad \mid un\ e \mid id(\overline{e}) \mid id.id() \\
op &::= \mathsf{\&\&} \mid \mathsf{||} \mid \leq \ \mid < \mid + \mid - \mid * \\
un &::= - \mid \mathsf{!} \\
s &::= \textbf{if}(e)\,\{\,\overline{s}\,\}\,\textbf{else}\,\{\,\overline{s}\,\} \\
&\quad \mid \textbf{while}(e)\,\{\,\overline{s}\,\}\ i \\
&\quad \mid x = e \mid \textbf{return}\ e \mid e \\
funcDef &::= id(\overline{x})\{\ \overline{s}\ \}
\end{aligned}
$$

Fig. 3. Abstract syntax of SECA

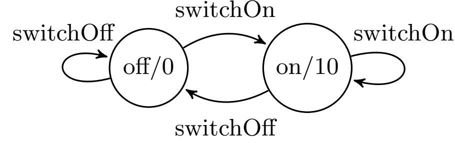


Fig. 4. Hardware component model for an LED. In state *on* it has a power draw of 10, in state *off* a power draw of 0. The transitions correspond to component functions.

symbolic component models. This would complicate the symbolic execution. For this paper we decided to keep this can of worms closed.

Assumptions. We provide no typing rules, but do require that programs are well-typed in the usual sense. We assume that all code paths of a function end in a return statement of the correct type, no references to undefined variables occur, and programs run on devices with all occurring hardware components. Void functions are allowed to end without return statements. In this case, the execution engine inserts an implicit return statement.

3.1 Syntax

A program is a list of function- and global variable declarations. There must be one function main. The abstract syntax of SECA is shown in fig. 3. Overlined symbols stand for lists of that symbol; for example $\overline{s}$ is a list of statements.

Expressions are Boolean or integer constants, program variables, applications of binary or unary operators, function calls, and component calls. Operators are the usual Boolean connectives, comparisons and arithmetic. Function calls have a list of expressions, the parameters. Component calls have the form *name.function*() and invoke the specified function of the specified component. Statements are conditionals, while loops, assignments, returns, or expressions.

While loops are annotated with a loop counter i, which the semantics uses to limit loop iterations, and to draw skylines differently in the first loop iteration. This is further discussed in section 3.2. The loop counter is not part of the concrete syntax, the programmer cannot access it, and it is initialised with zero. Assignments have a variable on the left hand side and an expression on the right hand side. Return statements end the current function call, and yield the given expression as the function's return value.

3.2 Semantics

SECA comes with four semantics, for different purposes. The *standard semantics* defines how programs are executed. The *energy-aware semantics* additionally traces the energy consumption during program execution in a skyline. The *symbolic execution semantics* executes all possible paths through a program. The

energy-aware symbolic execution semantics traces all possible skylines a program can produce. The focus of this paper is the last one; the others are formally defined in a technical report [19]. Below, we will informally discuss the energy-aware semantics, as it is a useful foundation to understand the energy-aware symbolic execution semantics.

Components. To start, we must define the semantics of component calls. In order to analyse the energy consumption of programs, we need an estimation of how much energy their hardware components consume. Such an estimation is called a *hardware component model*, or *component model* for short.

Component models are labelled transition systems, not necessarily finite, where each state has a power draw. Transitions are labelled with *component functions* (e.g. switchOn). Formally, a *hardware component model* $\langle S, L, \delta, o \rangle$ consists of a set of states S, a finite set of labels L, a transition function $\delta : L \times S \to IO(\mathbb{Z} \times S)$ and a power draw function $o : S \to \mathbb{N}$. A *configuration* of a model is an element of S: the current state. Every component has a start state. We borrow Haskell's notation $IO(\mathbb{Z} \times S)$ to indicate that to produce the return value $\mathbb{Z} \times S$, the function may perform arbitrary I/O. Input-producing hardware like sensors or terminals use the return value $\mathbb{Z}$ to return the input. Actuators like motors should return 0. The power draw function o specifies how much power the component consumes in each state.

Let us consider an example. A component model of an LED is shown in fig. 4. LEDs have two states, *on* and *off*, and transitions *switchOn* and *switchOff* between them. In the *on* state an LED has a power draw of 10, in the *off* state it has a power draw of 0. The component functions do not return values.

SECA programs always run in contexts where a number of component models are present. Such contexts are called *component states*, or CStates for short. A CState is a partial mapping from names to configurations. If the CState contains an LED, say under the name of LED1, programs in this context can contain the component calls LED1.switchOn() and LED1.switchOff().

Skylines. The energy-aware semantics generates *skylines*. A skyline is a list of *segments*. A segment is either a start point $S(l, p)$ at line l and power draw p, a forwards line $F(l)$, a backwards jump $J(l)$, or an edge $E(p)$. Every skyline has exactly one start point, which must be its first segment. Other segments are interpreted relative to their predecessors.

The y-axis of a skyline refers to line numbers. Using line numbers to identify program points requires the source code to be formatted so that every skyline-producing program point is on its own line, to avoid segments being drawn over each other. This concerns the left-hand side of assignments, **if** keywords, **while** keywords, the closing brace of the body of while loops, **return** keywords, and the closing parentheses of function- and component calls. Expressions that contain function- or component calls as subexpressions should be split over multiple lines. Even with these restrictions, segments may end up on top of each other when the same lines of code are executed more than once.

The energy-aware standard semantics. We explain by example how the semantics executes a program and constructs its skyline on the way. The program

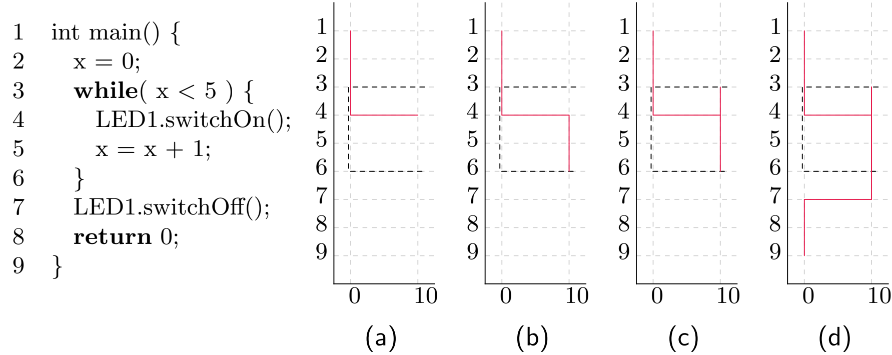


Fig. 5. Stepwise construction of a skyline (a) After switching on LED1 (b) After the first loop iteration (c) After the second iteration (d) The final skyline

in fig. 5 switches LED1 on five times in a loop, and then switches it off. The skyline fragments generated during execution are shown in fig. 6.

Execution of this program starts in a CState where LED1 is in state *off*. Figure 5a shows the skyline just after executing line 4, where the LED has been switched on. Lines 2 and 3 do not change the power draw, which yields two forward segments from line 1 to 2 and from line 2 to 3. LED1 is switched on in line 4, which extends the skyline with a forward segment from 3 to 4, followed by a rising edge to power draw 10.

$$
\begin{aligned}
&[S(1,0), F(2), F(3), F(4), E(10), && \text{(a)}\\
&F(5), F(6), J(3), && \text{(b)}\\
&F(4), E(10), F(5), F(6), J(3), && \text{(c)}\\
&F(4), E(10), F(5), F(6), J(3), &&\\
&F(4), E(10), F(5), F(6), J(3), &&\\
&F(4), E(10), F(5), F(6), J(3), &&\\
&F(7), E(0), F(8), F(9)] && \text{(d)}
\end{aligned}
$$

Fig. 6. Skyline fragments for fig. 5.

Figure 5b shows the skyline after one loop iteration, when the loop condition in line 3 has been executed a second time. Line 5 has caused a forward segment from 4 to 5. Execution of line 3 has caused a forward segment from the last statement of the loop in line 5 to the closing brace of the loop in line 6, followed by a backwards jump to line 3, which is not visible in the diagram.

Figure 5c shows the skyline after two iterations. The second iteration starts at line 3 with power draw 10, and gives of three forward segments 3 to 4, 4 to 5, and 5 to 6. None of them change the power draw, as the LED is already on. These segments overlap with the segments of the previous iteration, and are drawn on top of each other. All subsequent iterations also generate identical segments.

After five loop iterations, the program exits the loop, with the skyline shown in fig. 5d. Switching off the LED generates a falling edge to power draw 0 in line 7. The return statement finally generates two forward segments, one for itself from 7 to 8 and one for exiting the function from 8 to 9.

4 Energy-Aware Symbolic Execution

The energy-aware symbolic execution semantics tracks path constraints and energy skylines for each execution path. The result is a set of skylines for each function together with their path constraints.

In the symbolic semantics it is undesirable for component calls to perform I/O, because exploring all execution paths causes component calls to be executed multiple times. The symbolic semantics therefore uses component models where component calls return *symbolic values* with constraints. Symbolic values SVal are syntax trees whose leaves are constants or *symbolic inputs* any_i (variables that stand for an arbitrary integer). SVal is given by the grammar:

$$sv ::= \mathsf{true} \mid \mathsf{false} \mid i \mid any_i \mid sv\ op\ sv \mid un\ sv$$

A *symbolic component model* $\langle S, L, \delta, o\rangle$ consists of a set of states S, a finite set of labels L, a transition function $\delta : L \times S \to \text{SVal} \times \text{SVal} \times S$, and a power draw function $o : S \to \mathbb{N}$. As opposed to the concrete models in section 3.2, δ can not perform I/O. The first returned SVal of δ is typically a constant or symbolic input, and the second is a constraint on that input. For example, where the concrete model of TERM.readInt() asks for input and returns the user's answer, the symbolic model returns a fresh symbolic input any_j, with the constraint true, as this input can be any integer. A temperature sensor in a cold room can return a fresh any_j, together with a constraint $13 \leq any_j \wedge any_j \leq 17$. A symbolic LED returns constant 0, with the constraint true.

4.1 The Energy-Aware Symbolic Execution Semantics

We now study the algorithm that computes all possible executions of SECA programs, together with their corresponding skylines. The algorithm records skylines of each function separately. This results in one skyline for each function call, for each execution path on which the call lies. The algorithm is defined by case distinction on the abstract syntax. We present the whole algorithm in figs. 7 and 8, but provide a detailed description only for a few clauses. A complete description of the algorithm, as well as a formal definition of the semantics in the style common in programming language research, can be found in the technical report [19]. An implementation is available online [3].

Figures 7 and 8 show pseudocode for the functions E and S that compute symbolic skylines for expressions and statements respectively. We elaborate on some of the clauses below. Application of a function to syntactic arguments is denoted with double brackets $[\![-]\!]$, which have no further special meaning. A program state $\sigma \in \Sigma$ is a record with all information needed to execute a statement. It contains the values of local program variables *env* and global program variables *genv*, the current skyline *sky*, the current path constraint φ, the program counter *pc* (a list of statements to be executed after the current statement), the function call stack *stack*, and the CState *cstate*. The helper functions *lookup* and *assign* (not shown here) ensure that the scoping rules are respected, which means they prefer variables in *env* over *genv*.

$$E : \mathbf{Expr} \times \Sigma \to \mathcal{P}(\mathbf{Val} \times \Sigma)$$

$$E[\![x]\!](\sigma) = \{\, \langle lookup(x,\sigma), \sigma\rangle \,\} \quad (1)$$

$$E[\![e_1\ op\ e_2]\!](\sigma) = \{\, \langle v_1\ op\ v_2, \sigma''\rangle \mid \langle v_1, \sigma'\rangle \in E[\![e_1]\!](\sigma), \langle v_2, \sigma''\rangle \in E[\![e_2]\!](\sigma') \,\} \quad (2)$$

$$E[\![un\ e]\!](\sigma) = \{\, \langle un\ v, \sigma'\rangle \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma) \,\} \quad (3)$$

$$E[\![f(\overline{e})]\!](\sigma) = \{\, \langle lookup(\#return, \sigma'''), \sigma'''[pc \mapsto \sigma.pc]\rangle \mid \langle \overline{v}, \sigma'\rangle \in \overline{E}[\![\overline{e}]\!](\sigma), \sigma'' = call[\![f(\overline{v})]\!](\sigma'[pc \mapsto []]), \sigma''' \in X(\sigma'') \,\} \quad (4)$$

$$E[\![c.f()]\!](\sigma) = \{\, \langle v, \sigma'\rangle \,\} \quad (5)$$

where

$\sigma' = \sigma[cstate \mapsto cstate', sky \mapsto sky', \varphi \mapsto \varphi']$

$\langle v, \psi, s_c'\rangle = \delta_c(f, s_c)$

$cstate' = \sigma.cstate[c \mapsto s_c']$

$s_c = \sigma.cstate[c]$

$p = powerDraw(cstate')$

$l = lineOfCompCall$

$sky' = \sigma.sky +\!+ [F(l), E(p)]$

$\varphi' = \sigma.\varphi \wedge \psi$

$$call : \mathbf{Expr} \times \Sigma \to \Sigma$$

$$call[\![f(\overline{v})]\!](\sigma) = \sigma' \quad (6)$$

where

$\sigma' = \sigma[env \mapsto env', sky \mapsto sky', pc \mapsto \overline{s}, stack \mapsto stack']$

$env' = [\overline{x} \mapsto \overline{v}]$

$p = powerDraw(\sigma.cstate)$

$l = lineOfOpeningBrace$

$sky' = [S(l,p)]$

$\overline{s} = functionBody[\![f]\!]$

$stack' = push(\sigma, stack)$

$$X : \Sigma \to \mathcal{P}(\Sigma)$$

$$X(\sigma) = \begin{cases} \{\,\sigma\,\} & \text{if } \sigma.pc = [] \\ \bigcup\{\, X(\sigma') \mid \sigma' \in S[\![s]\!](\sigma[pc \mapsto rest]) \,\} & \text{if } \sigma.pc = [s] +\!+ rest \end{cases}$$

Fig. 7. The function E for expressions and X to execute whole programs

$$S : \mathbf{Stmt} \times \Sigma \to \mathcal{P}(\Sigma)$$

$$S[\![x = e]\!](\sigma) = \{\, assign(x, v, \sigma') \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma[sky \mapsto \sigma.sky +\!+ [F(l)]]) \text{ where } l = lineNumberOfAssignment \,\} \quad (1)$$

$$S[\![\textsf{if}(e)\,\{\,\overline{s}_1\,\}\,\textsf{else}\,\{\,\overline{s}_2\,\}]\!](\sigma) = \bigcup\{ \{\, \sigma'[pc \mapsto pc_1, \varphi \mapsto \varphi_1], \sigma'[pc \mapsto pc_2, \varphi \mapsto \varphi_2] \,\} \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma[sky \mapsto \sigma.sky +\!+ [F(l)]]) \quad (2)$$

where

$l = lineOfIfKeyword$

$\varphi_1 = \sigma'.\varphi \wedge v$

$\varphi_2 = \sigma'.\varphi \wedge \neg v$

$pc_1 = \overline{s}_1 +\!+ \sigma'.pc$

$pc_2 = \overline{s}_2 +\!+ \sigma'.pc\ \}$

$$S[\![\textsf{while}(e)\,\{\,\overline{s}\,\}\,i]\!](\sigma) = \bigcup\{ \{\, \sigma'[pc \mapsto loop, \varphi \mapsto \varphi_1], \sigma'[\varphi \mapsto \varphi_2] \,\} \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma[sky \mapsto sky']) \quad (3)$$

where

$\varphi_1 = \sigma'.\varphi \wedge v$

$\varphi_2 = \sigma'.\varphi \wedge \neg v$

$sky' = \begin{cases} \sigma.sky +\!+ [F(l)] & \text{if } i = 0 \\ \sigma.sky +\!+ [F(m), J(l)] & \text{otherwise} \end{cases}$

$loop = \overline{s} +\!+ [\textsf{while}(e)\{\overline{s}\}(i+1)] +\!+ \sigma'.pc$

$l = lineOfWhileKeyword$

$m = lineOfClosingBrace\ \}$

$$S[\![\textsf{return}\ e]\!](\sigma) = \{\, registerSkyline(f, sky'', \sigma'') \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma[sky \mapsto sky']) \quad (4)$$

where

$sky' = \sigma.sky +\!+ [F(l)]$

$sky'' = \sigma'.sky +\!+ [F(m)]$

$sky_c' = \sigma_c.sky +\!+ [F(n), E(p)]$

$\langle \sigma_c, stack_c\rangle = pop(\sigma.stack)$

$\sigma'' = \sigma'[env \mapsto \sigma_c.env[\#return \mapsto v], pc \mapsto \sigma_c.pc, sky \mapsto sky_c', stack \mapsto stack_c]$

$l = lineOfReturnKeyword$

$m = lineOfClosingBrace$

$n = lineOfCallSite$

$p = powerDraw(\sigma')\ \}$

$$S[\![e]\!](\sigma) = \{\, \sigma' \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma) \,\} \quad (5)$$

Fig. 8. The function S for statements

Each clause of the semantics specifies how a single statement or expression together with a given program state produces the set of all possible immediate successor program states. Hence, a statement can be seen as a state transformer $\Sigma \to \mathcal{P}(\Sigma)$. To compose two functions of this type, we need glue code that applies the second function to every result of the first function. This is implemented by the function X in fig. 7, which executes whole programs.

4.2 Evaluation of Expressions

The evaluation function E (fig. 7) takes an expression e and a program state and returns the set of all possible values that e can evaluate to, together with the updated program states. Clauses **(1)** and **(3)** are not explained here.

Clause **(2)**: To evaluate a binary operator, all possible values v_1 for e_1, and all possible values for e_2 are calculated. The evaluation of e_2 happens in the result state of the evaluation of e_1. The result is the set of the symbolic values $v_1\, op\, v_2$ for all combinations (v_1, v_2). These values are subject to constant folding (e.g. $1 + 2$ becomes 3), which is not shown here.

Clause **(4)**: To evaluate a function call, first all arguments are evaluated. This is done by the sequential extension $\overline{E}$, which chains the state through the evaluation of the argument vector $\overline{e}$ and results in the set of all possible value vectors $\overline{v}$. For each vector $\overline{v}$, the helper function *call*, described below, prepares the function call, and X executes it. This execution will eventually end with a return statement. The return statement restores the program state so that σ''' can be used as the result state at the call site. The resulting value of the function call is the value of the #return register in σ'''.

Clause **(5)**: To evaluate a component call, first the transition function δ_c of the component c is invoked, with the function name f and the component's current state s_c as arguments. This yields a return value v, a constraint ψ on v, and a new component state s'_c. The total power draw p after the call is computed. The skyline is extended with a forward segment $F(l)$ to the location l of the call site, followed by an edge $E(p)$ to the new power draw. The result of evaluating $c.f()$ is the return value v of δ_c, together with the updated program state.

Clause **(6)**: The helper function *call* prepares the program state for execution of the function. It first initializes the environment env' for the function body with the actual arguments. It then starts a new skyline for the call to f with the current power draw p at the location l of the opening brace of the function definition of f. It uses the function body $\overline{s}$ as program counter, and creates a new stack frame for the function call.

4.3 Execution of Statements

The function X (fig. 7) recursively executes all statements of the program counter $\sigma.pc$, and collects the results. Execution of a SECA program starts in a program state that contains the body of the `main` function as program counter.

The function S (fig. 8) executes a single statement in a given program state, and returns all possible successor program states.

Clause **(1)**: Assignments are executed by first extending the current skyline to the line of the assignment. Then e is evaluated to all its possible values, and the final results are all successor states where x has value v. Expressions can have side effects, so the successor states may have different skylines.

Clause **(2)**: Execution of conditionals starts with extending the current skyline with a forward segment $F(l)$ to the line l of the *if* keyword. Then, all possible values v of the condition e are computed. This results in paths into both branches, for each v. The path constraint for the *then* branch is extended with v, for the *else* branch with $\neg v$. The program counter pc_1 specifies that first the statements $\overline{s}_1$ of the *then* branch are executed, and after that the original continuation $\sigma'.pc$. Similarly for the *else* branch. If the SMT solver sees that φ_1 or φ_2 is unsatisfiable, their states are pruned (not shown here).

Clause **(3)**: For the first iteration of while loops, we need to generate a different skyline than for subsequent iterations. The first loop iteration can be recognized by the loop counter i being 0. If this is the case, the current skyline comes from outside the loop body, and we extend it with a forward segment $F(l)$ to the line of the *while* keyword. Otherwise, the current skyline comes from inside the loop body, and is extended with a forward segment $F(m)$ to the line m of the closing brace, followed by a backwards jump $J(l)$ to the beginning of the loop. In both cases, the condition e is evaluated to all possible values v. For every v, we generate two continuations: one for entering the loop with path constraint $\sigma'.\varphi \wedge v$ and one for exiting the loop with constraint $\sigma'.\varphi \wedge \neg v$. The program counter *loop* for entering the loop consists of the loop body $\overline{s}$, followed by the loop itself with incremented loop counter, then by what comes after the loop $\sigma'.pc$. The program counter for exiting needs no change, as $\sigma'.pc$ already contains the instructions following the loop. Our implementation uses the loop counter to bound the number of iterations. This is not shown here.

Clause **(4)**: The clause for return statements is more complicated than the others, as it has to deal with two different skylines: the one from the function that is about to return, and the one from the caller. Let f be the name of the current function. To execute a return statement, the current skyline is first extended with $F(l)$ to the location l of the return keyword. Then, the returned expression is evaluated. Next the skyline is extended with $F(m)$, to the location m of the closing brace of the function body. Then, the program state from before the function call is restored, but updated with all the changes made by f. For this, the topmost element σ_c of the call stack is removed; this is the program state of the caller. A new state σ'' is constructed, which the caller should use to resume execution; σ'' has the caller's original *env*, but with the #return register holding the return value. The program counter and call stack are restored to the ones from before the call. The caller's skyline $\sigma_c.sky$ is extended with a forward line $F(n)$ to the call site, and an edge $E(p)$ to the power draw p. Finally, the skyline of the function call is recorded in the list of all skylines of f. This is done with the function *registerSkyline*, which stores the given skyline in the given state, and returns the thus updated state.

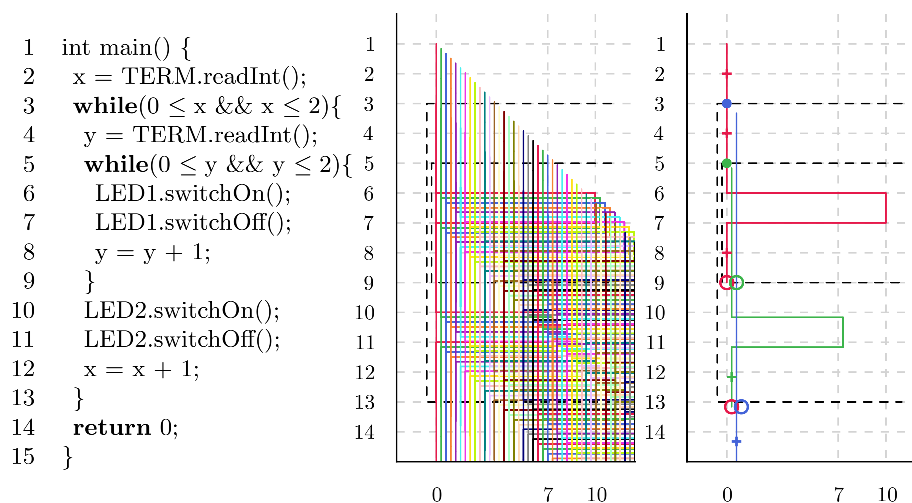


Fig. 9. A program with many execution paths, and its unmerged and merged skylines.

5 Merging Skylines

The skylines of a program often have many identical parts. Take for example the program in fig. 9. Most of its execution paths have identical energy behaviour. To merge skylines, we use a three-phase algorithm: *preparation*, *merging* and *finalization*. It is executed independently for every function.

Preparation. First all skylines are split into *fragments* and stored in an array, giving each a unique index. Fragments represent single horizontal or vertical lines with explicit start and end points. Every fragment has a set of continuations: indexes of the fragments that follow it. Merging deletes explicit jumps $J(l)$: they are kept implicitly as fragments whose start point does not coincide with the end point of their predecessor. Initially each fragment has at most one continuation, but more may be added later. Preparation is shown in fig. 11.

procedure COLOURIZE(*frags*)
  for $i \leftarrow 1$ to *Nfrags* **do**
    if $(i > 1) \wedge (i \in \mathit{frags}[i-1].\mathit{conts}) \wedge (\mathit{frags}[i-1].\mathit{end} = \mathit{frags}[i].\mathit{start})$ **then**
      $\mathit{colour}[i] \leftarrow \mathit{colour}[i-1]$
    else
      $\mathit{colour}[i] \leftarrow$ a fresh colour
    end if
  end for
end procedure

Fig. 10. Assigning colours

Merging. Whenever two fragments indexed i and j are equal, we can merge them by first combining their continuations, and then replacing all occurrences of j in continuations of other fragments by i. This is formally described by fig. 12.

```
procedure Prepare(skies(f))
  // input: all skylines of function f
  // output: frags, an array of fragments,
  // each with at most one continuation
  Nfrags ← 0
  for all Skyline sky ∈ skies(f) do
    // all skylines begin with S(l,p)
    let ⟨l, p⟩ be such that sky[1] = S(l, p)
    for i ← 2 to length(sky) do
      sky[i] is either F(l′) or J(l′) or E(p′)
      in the first two cases, let p′ = p
      in the last case, let l′ = l
      if sky[i] is F(l′) or E(p′) then
        Nfrags ← Nfrags + 1
        frags[Nfrags].start ← ⟨l, p⟩
        frags[Nfrags].end ← ⟨l′, p′⟩
        frags[Nfrags−1].conts ← { Nfrags }
      end if
      ⟨l, p⟩ ← ⟨l′, p′⟩
    end for
    // last fragment has no continuation
    frags[Nfrags].conts ← ∅
  end for
end procedure
```

Fig. 11. Initializing the *frags* array

```
procedure Merge(frags, Nfrags)
  // frags, Nfrags as produced by prepare
  // output: modified frags with equal
  // fragments merged
  for i ← 1 to Nfrags − 1 do
    if frags[i] = null then continue
    for j ← i + 1 to Nfrags do
      if frags[j] = null
      or frags[i].start ≠ frags[j].start
      or frags[i].end ≠ frags[j].end
        then continue
      frags[i].conts ←
          frags[i].conts ∪ frags[j].conts
      frags[j] ← null
      for k ← 1 to Nfrags do
        if frags[k] ≠ null ∧
            j ∈ frags[k].conts then
          frags[k].conts ←
              (frags[k].conts \ { j }) ∪ { i }
        end if
      end for
    end for
  end for
end procedure
```

Fig. 12. Merging fragments

Visualization. Finally, fragments are grouped into skylines, by assigning the same colour to directly connected fragments. Figure 10 implements this. It then assigns a small diagonal offset to each colour group (not shown here), to avoid drawing lines on top of each other.

Statement markers. Between two consecutive horizontal lines, a + indicates that a statement was executed at that point. Continuations are drawn as coloured bullets or circles: if $j \in \mathit{frags}[i].\mathit{conts}$ and $\mathit{colour}[i] \neq \mathit{colour}[j]$, then if $\mathit{frags}[i].\mathit{end} = \mathit{frags}[j].\mathit{start}$ then a bullet in $\mathit{colour}[j]$ is drawn at the end of fragment i. Otherwise, the continuation is a jump backwards; this is indicated by drawing an open circle in $\mathit{colour}[j]$ at the end of fragment i. Dotted lines in the diagram indicate the beginning and end points of loops.

6 A Real-World Example: Line-Following Robot

In this section we demonstrate how to apply our analysis to an existing real-world example, written in C. The program is simple enough that there is no potential for energy consumption optimization, but nonetheless our analysis gives insight into the program's energy behaviour.

We chose a random "simple line follower" project from the Arduino project database [1]. This robot has two motors and two sensors, and uses them to

```
1   int main() {
2     while( true ) {
3       loop();
4     }
5     return 0;
6   }
7   void loop(){
8     if((SensorLeft.read()==LOW) &&
9       (SensorRight.read()==LOW)) {
10      MoveForward();
11    }
12    if((SensorLeft.read()==HIGH) &&
13      (SensorRight.read()==HIGH)) {
14      Stop();
15    }
16    if((SensorLeft.read()==LOW) &&
17      (SensorRight.read()==HIGH)) {
18      TurnLeft();
19    }
20    if((SensorLeft.read()==HIGH) &&
21      (SensorRight.read()==LOW)) {
22      TurnRight();
23    }
24  }
25  void MoveForward() {
26    MotorLeft.Forward();
27    MotorRight.Forward();
28    delay(20);
29  }
```

Fig. 13. Excerpt of line follower program

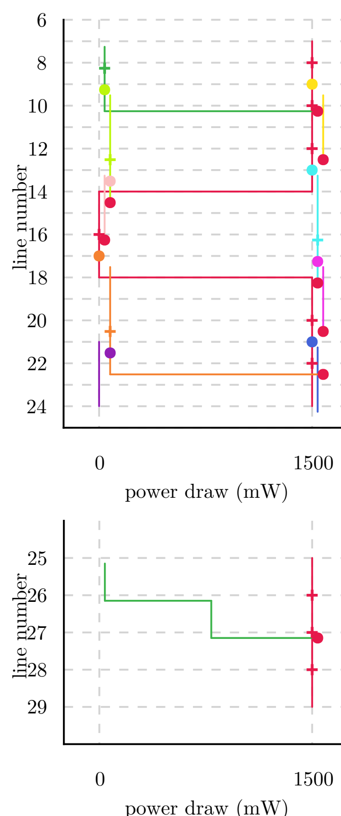


Fig. 14. Diagrams for *loop* (top) and *MoveForward* (bottom)

follow a black line on the floor. It works as follows. The sensors are positioned to the left and right of the line. If only the left sensor sees the line, the robot turns left. Symmetrically for the right sensor. If neither sensor sees the line, the robot moves forward. If both sensors see the line, the robot stops. The code has potential for refactoring, as it contains unnecessary repetition. However our goal was not to find the most elegant line follower robot, but to apply our method to a real-world example.

The original source code, written in C, is almost valid SECA. We made two changes to the code for our parser to accept it. First, we defined the constants `LOW` and `HIGH`, and the function `delay`, which for our purpose is empty. Second, we replaced the statements that write to output pins and read from input pins with component calls. Figure 13 shows an excerpt of the code after these adjustments.

We then created the component models for motors and sensors in the source code of our analysis engine. The simulated motors have three states, *forward*, *backward*, and *stop*, and corresponding component calls. In the forward and

backward states, motors have a power draw of 750mW. The sensors have no power draw, and their `read` component call returns a symbolic value in $\{0, 1\}$.

Analysis results. Figure 14 shows skyline diagrams for the functions *loop* (top) and *MoveForward* (bottom). The diagram for *MoveForward* illustrates that the function has two behaviours. One where the power draw increases in two steps from 0 to 1500 mW, and one where the power draw stays constant at 1500 mW. The functions *TurnLeft* and *TurnRight*, not shown here, look similar. The function *Stop*, also not shown, has the opposite behaviour: the power draw decreases in two steps. The function *loop* has many behaviours, depending on the executed conditional. There are the cases where the power draw stays 0 or 1500 mW, or it can increase in lines 10, 18, or 22, or it can decrease in line 14.

Discussion. The function *loop* has high complexity for symbolic execution. We had to set the iteration limit to 2 for the analysis to terminate within 20 seconds on a ten year old laptop. The merged diagram would not change with more iterations. The high complexity occurs because firstly the four conditionals can be entered independently, and secondly the sensors are read in each condition, making the conditionals not mutually exclusive. This results in 16 possible executions of the function. Refactoring the program either by reading the sensors once at the start of `loop()` or by nesting the conditionals, reduces the number of possible executions to 4, making the analysis terminate in 2.4 seconds with iteration limit 2. An improved version of the robot program together with its skyline diagrams can be found on the project website [2].

7 Related Work

Directly related are the second author's previous works [11,16] describing static energy analyses for the language ECA. The first derives energy bounds for a specific input scenario; the second is a symbolic analysis that over-approximates all possible paths. These works do not use skylines. They do use hardware models that also support incidental one-time energy costs. This incidental energy cost can be useful for approximating energy consumption that varies over time. Also closely related is the first author's previous work [20], which introduces skylines but also an overapproximation since it estimates resource use over time.

Most publications on energy efficiency of software approach the problem on a high level, defining programming and design patterns for writing energy-efficient code; see, e.g., [4,24,26]. In [9] and [25], a program is divided into *phases* describing similar behaviour. Based on the behaviour of the software, design-level optimizations are proposed to achieve lower energy consumption. Petri-net-based energy modelling techniques for embedded systems are proposed in [14,23].

A general analysis for resource consumption is described in [17]. There are generic resource consumption analyses, built on techniques such as solving recurrence relations [5], amortized analysis [12], separation logic [7], and a Vienna Development Method style program logic [6]. Contrary to these approaches, our method has an explicit hardware model and a context in the form of component states. This enables the inclusion of state-dependent energy consumption.

In [13] and [15] energy consumption of the processor running embedded software is analysed for specific architectures (SimpleScalar in [13], and XMOS ISA-level models in [15]), while our approach is hardware-parametric and focuses on external hardware. Several tools perform a static analysis of the energy consumption of the CPU based on per-instruction measurements, such as in [27,8].

8 Discussion and Future Work

This article proposes a new approach for visualizing the energy consumption of a system with external hardware without actually running the software or having a real test setup. The result is presented as skyline diagrams with a direct link to the source code, using line numbers. This visualization is generated by symbolic execution, followed by a merging algorithm to deal with the explosion of possible execution paths. There are few restrictions on the models of hardware components, allowing a user to model a wide variety of hardware components.

We have implemented all techniques of this article in Haskell as a proof of concept, using the Z3 SMT library to prune infeasible paths. Every skyline diagram in this paper and on the project website [2] was computed by this tool in a few seconds on a ten year old laptop. However, since symbolic execution has exponential complexity, it would only take a couple of nested loops containing component calls for the analysis to no longer be computed in feasible time.

The focus of this paper is on a minimal implementation, to explore the idea of drawing graphs over source lines. A larger case study using the implementation could result in useful feedback on how the process can be applied in practice. This case study should evaluate if programmers get feedback they can use, and if there is a practical need to use another technique instead of, or alongside, bounded symbolic execution. In particular, our visualization should lend itself well to abstract interpretation, which can be an alternative to symbolic execution. This would require incorporating parts of the merging algorithm into the abstract interpretation. A combination of symbolic execution and abstract interpretation would be more complex, but could provide powerful tooling.

Editor integration can improve usability, for instance by annotating or overlaying the source code with diagrams. It may also be useful to offer an interactive visualization that allows developers to explore skylines and recover information about individual execution paths, highlighting the relevant code.

Finally, our approach could track other resources. A similar methodology could be used to visualize memory usage, or even time. *Incidental*, one time, energy consumption of hardware component calls could also be relevant to show.

Acknowledgements. We would like to thank Rinus Plasmeijer, Olha Shkaravska, Tim Steenvoorden, and Nico Naus for many hours of fruitful discussion. Special thanks goes to Ralf Hinze, who created an exam question, the grading of which eventually led to the idea of resource skylines. Thanks also to Pieter Koopman who provided funding for this project.

References

1. Arduino project hub. https://create.arduino.cc/projecthub, accessed: 2020-05-01
2. SECA project wiki. https://gitlab.science.ru.nl/mklinik/eca-symbolic-execution/-/wikis/home, accessed: 2020-02-06
3. SECA source code repository. https://gitlab.science.ru.nl/mklinik/eca-symbolic-execution, accessed: 2020-01-29
4. Albers, S.: Energy-efficient algorithms. Communications of the ACM **53**(5), 86–96 (2010)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In: Formal Methods for Components and Objects, vol. 5382, pp. 113–132. Springer (2008)
6. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.W., Momigliano, A.: A program logic for resources. Theoretical Computer Science. **389**(3), 411–445 (Dec 2007). https://doi.org/10.1016/j.tcs.2007.09.003
7. Atkey, R.: Amortised resource analysis with separation logic. In: Programming Languages and Systems. LNCS, vol. 6012, pp. 85–103. Springer (2010). https://doi.org/10.1007/978-3-642-11957-6
8. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations. SIGARCH Computer Architecture News **28**(2), 83–94 (May 2000)
9. Cohen, M., Zhu, H.S., Senem, E.E., Liu, Y.D.: Energy types. SIGPLAN Notices **47**(10), 831–850 (Oct 2012)
10. van Gastel, B.: Assessing sustainability of software - Analysing Correctness, Memory and Energy Consumption. Ph.D. thesis, Open University (2016)
11. van Gastel, B., Kersten, R., van Eekelen, M.: Using dependent types to define energy augmented semantics of programs. In: Proceedings of FOPARA'15. LNCS, vol. 9964, pp. 20–39 (2015)
12. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Ball, T., Sagiv, M. (eds.) POPL'11. pp. 357–370. ACM (2011)
13. Jayaseelan, R., Mitra, T., Li, X.: Estimating the worst-case energy consumption of embedded software. In: Proceedings of RTAS'06. pp. 81–90. IEEE (2006). https://doi.org/10.1109/RTAS.2006.17
14. 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. pp. 261–281 (2006)
15. 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 (Sep 2013)
16. Kersten, R., Parisen Toldin, P., van Gastel, B., van Eekelen, M.: A Hoare logic for energy consumption analysis. In: Proceedings of FOPARA'13. LNCS, vol. 8552, pp. 93–109. Springer (2014)
17. Kersten, R., Shkaravska, O., van Gastel, B., Montenegro, M., van Eekelen, M.: Making resource analysis practical for real-time Java. In: Proceedings of JTRES'12. pp. 135–144. ACM (2012). https://doi.org/10.1145/2388936.2388959
18. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (Jul 1976)
19. Klinik, M., van Gastel, B., Kop, C., van Eekelen, M.: Skylines for symbolic energy consumption analysis – technical report. Tech. rep., Radboud University

(2020), https://gitlab.science.ru.nl/mklinik/eca-symbolic-execution/blob/master/paper/techreport.pdf

20. Klinik, M., Jansen, J.M., Plasmeijer, R.: The sky is the limit: Analysing resource consumption over time using skylines. In: Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages, IFL 2017. ACM (2017)
21. Microchip Technology Inc.: ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet (2018)
22. Nielson, H.R., Nielson, F.: Semantics With Applications: A Formal Introduction. Wiley (1992)
23. 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 Journal on Embedded Systems pp. 2:1–2:12 (Jan 2011). https://doi.org/10.1155/2011/316510
24. Ranganathan, P.: Recipe for efficiency: principles of power-aware computing. Communications of the ACM **53**(4), 60–67 (2010)
25. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: EnerJ: approximate data types for safe and general low-power computation. SIGPLAN Notices **46**(6), 164–174 (Jun 2011)
26. Saxe, E.: Power-efficient software. Communications of the ACM **53**(2), 44–48 (2010). https://doi.org/10.1145/1646353.1646370
27. Sinha, A., Chandrakasan, A.P.: JouleTrack: A web based tool for software energy profiling. In: Proceedings of DAC'01. pp. 220–225. ACM (2001)