Energy Consumption Analysis of Practical Programming Languages

Radboud University Nijmegen

Master Thesis

Author:
Stein Keijzers

Supervisors:
Prof.dr. Marko C.J.D. van Eekelen
Drs. Bernard van Gastel

August 13, 2014

Abstract

Energy consumption of software has received significantly more attention in the past decade due to the advent of increasingly small, mobile technology; embedded systems or smart-phones are common enough to make energy and battery consumption a real issue, even aside from the general power consumption of server farms and other such large-scale hardware. Care is being given to optimizing energy usage on the level of the software running on the hardware, rather than just hardware optimizations.

Two avenues for measuring and reasoning about software energy consumption exist: static and dynamic analysis. Most work focuses on dynamic analysis by executing software on specific hardware and measuring the resulting power draw, but the practice of static analysis draws attention by virtue of the lack of requirement for measuring equipment and its relative ease. Based on a set of energy analysis rules[1], one tool performing such static analysis on a small, While-like[2] language is EcaLogic[3]. The analysis uses hardware models coupled with user-defined bounds to analyse the energy consumption of a program.

A limited static energy analysis is also limited in its application. This thesis focuses on extending the energy analysis and its implementation in EcaLogic to include elements from real, practical programming languages: by adding additional data structures, support for recursion and an overall extension of the grammar, usability of the analysis can be significantly increased. The end result is an extension of the static energy analysis method, culminating into an implementation applying it to the C programming language, chosen due to its prevalence mainly in embedded system applications; the extended tool is referred to as EcaLogic-C.

Contents

1 Energy Consumption Analysis

With the ever-increasing usage of embedded systems and hand-held electronic devices, more consideration has been given to the energy efficiency of these objects. Large-scale independent sensor networks are an example of embedded systems where power is a major concern, both due to the difficulty of reaching the devices and the need for them to operate for extended periods of time. Research into improving the energy consumption of these devices often focuses on the hardware, or programming techniques to improve energy efficiency. For instance, Alippi et al. published research on more efficiently retaining energy in large low-power sensor networks, particularly buoy networks. They found that, by adding either a twin-battery or more advanced capacitors, they were able to increase the uptime of individual sensors substantially[4]. They later combined this research with an improved internal structure for the control units[5], stating that more direct internal bus systems were more efficient for transmitting the sensory information.

Tackling the energy efficiency of hardware, while important, only concerns one side of the problem. Research into the energy efficiency of software has thereby attained more attention: after all, efficient hardware operating under inefficient software can decrease the potential efficiency of the hardware substantially, or even negate it in extreme cases. Research into this area commonly features practices and programming tactics to improve the energy efficiency of the final system. Kothari et al. reason that, by moving from individual-level programming to reasoning about an entire system, greater efficiency in the software can be attained by grouping common operations into a more hardware-friendly manner[6], effectively creating highly specialized low-level libraries. Poletti et al. instead researched parallel-computing models, focussing on energy efficiency by minimizing overhead and downtime, to better exploit newer embedded system structures[7].

A different approach to analysing software for energy efficiency deals with existing systems by analysing and reasoning about the energy consumption as a result of the code, or by careful measurement of the power draw while the system is running. Real-time measurements are referred to as Dynamic Analysis, whereas Static Analysis means analysing the code without necessarily having to run it.

1.1 Dynamic Energy Analysis

Analysing a system dynamically refers to measuring the power draw of its hardware during operation. This method poses several problems: the equipment and time required to perform the measurement are expensive, it takes a significant amount of time to run a sufficient number of tests, and the tests themselves have to be accurate and cover at least regular usage, preferably more.

Commonly, this form of analysis considers 'average' usage-cases: behaviour is defined as normal or exceptional, and the normal cases receive the focus. This does not always hold for safety-critical applications: for instance, in aviation electronics and software, the worst-case scenario may pose too great a risk in practice to skip for the analysis. Nevertheless, a proper analysis meeting these requirements can be used to identify bottlenecks in both the code and actual hardware, or satisfy specific energy or time constraints. This option is attractive for businesses, those working with potentially dangerous equipment, and other users in charge of maintaining hardware on a large scale; case studies have been performed on varied hardware such as networking devices[8], mobile consumer electronics[9] and CPU hardware[10]. The results that may be gathered by this type of analysis are, in many cases, worth the required effort, though it may not always classify as an exhaustive or proper analysis, so it can be prone to missing uncovered or exceptional errors.

1.2 Static Energy Analysis

The investment required to perform a dynamic analysis, and its susceptibility for errors, opens up demand for a faster analysis lacking many of the requirements posed by actual power measurement. This form of analysis, which focuses on analysing code or systems without running it or activating them, is referred to as static analysis. While it does not share the disadvantages of a dynamic analysis, it also does not provide the same advantages: without real measurement, the result of a static analysis will always be approximate. Accuracy instead depends on how accurate the models and rules for reasoning about the energy consumption are.

The main draw of this form of analysis is the ease of application. Rather than having to acquire measuring equipment and perform all the required steps for a dynamic analysis, it may be executed on any system, independent of the actual hardware it is running on. However, one problem not present with dynamic analysis is the fact that, in general, it is very difficult, if not impossible without intervention[11], to completely predict a program's behaviour. Concessions have to be made to accuracy in order to be able to perform the analysis as best as possible. As a result, research in this area focuses more on eliminating or alleviating these restrictions, which can improve both the accuracy and usability of the analysis.

Several projects exist that focus on static analysis of code, some of which are described in greater detail below. In particular, this thesis focuses on extending an existing analysis[1], and applying these extensions to the tool EcaLogic[3], itself an implementation of the existing analysis. Both the analysis and tool are described in greater detail in Section 2.2.

1.2.1 Spec#

Microsoft provides an analysis extension for their C# language systems capable of static analysis (as well as run-time analysis for some aspects of the code, such as exceptions), based on 'contracts' for functions and objects, called Spec#[12][13][1]. Run-time checks and assumptions may also be specified, and can be checked statically by a semi-automatic theorem prover. The contracts for functions and objects specify what a function may or may not do, and within which bounds this should fall. This option extends to loop invariants. No specific analysis options exist for termination or bounding, although it is possible to emulate them by adding always-decreasing and equivalent assertions to contracts or invariants.

1.2.2 Frama-C

Similar to Spec# is Frama-C[14][15][2]: a collection of tools providing static and (minor) dynamic analysis of a language, in this case C. The focus of Frama-C lies on conformance to a specification and the analysis or prevention of run-time errors. However, some consideration is given for bounding and termination analysis: loop variants may be specified in an ever-decreasing shorthand notation to denote a bound, and functions may be checked for termination. It also offers significantly more custom assertions and elements through the ACSL language[3], which allows the user to define entire logical structures, new predicates, and other elements that can extend the applicable area.

Furthermore, Frama-C offers two plug-ins by default for automatic verification and proving assertions: WP and Jessie. These may be used in conjunction with any other plug-in to add formal verification and (partial) correctness checks to a program, which can improve the confidence in aspects of the code. An example is a plug-in that performs an analysis on the variables, and can generate assertions to prove that they are not modified outside of specific sections in the code.

1.2.3 KITTeL

An academic tool, KITTeL[16][17][4] aims to analyse the termination behaviour of C programs out of the box: that is to say, it does not use any existing annotations for its analysis, unlike both Spec# and Frama-C, and runs directly on unedited C. It uses the LLVM compiler back-end[5] to transform C programs into a more basic form, then performs an analysis based on term-rewriting systems. The

[1] Spec#, at Microsoft: `http://research.microsoft.com/en-us/projects/specsharp/`

[2] FramaC may be found here: `http://frama-c.com/`

[3] ACSL Specification: `http://frama-c.com/download/acsl-implementation-Fluorine-20130601.pdf`

[4] KITTeL may be found at: `http://baldur.iti.kit.edu/~falke/kittel/`

[5] LLVM is based here: `http://llvm.org/`

result is a simple verdict on the termination behaviour, with a select few of the reasoning steps given. While the tool is able to analyse programs fairly accurately, it has severe limitations on the set of programs it's able to analyse, and uses a naive method to determine non-termination: if the analysis times out, the verdict is that it does not terminate.

2 A Hoare Logic For Energy Analysis & EcaLogic

The focus of this thesis lies on building upon an existing method of static energy analysis: the analysis rules detailed in A Hoare Logic For Energy Analysis[1], which were proven to be correct[18], and implemented in EcaLogic[3]. The first part of this chapter summarizes the original method of analysis, the second part discusses the original implemented tool.

2.1 Energy-Aware Hoare Logic

The original energy analysis makes use of an energy-aware Hoare Logic describing the analysis steps for each expression or statement. This section describes the basis of the analysis, and offers explanation for the elements used in both the semantic and analysis rules. The rules given in this section have been modified from the original to a more common format by applying additional typing restrictions, as well as a syntax more corresponding to Java or C. The completely unaltered rules may be found in the original paper[1]. Extensions to these rulesets are described in their respective chapters where appropriate. A full listing of the final rules, both for semantics and the analysis, can be found in Appendix A.

2.1.1 Energy-Aware Rules

To reason about the energy consumption of a program, the analysis applies energy-aware rules based on Hoare logic[1]. Semantics of the program are described with separate rules. The energy analysis rules describe, for the set of possible statements and expressions, what effect each operation or program step has on the energy consumption of the (modelled) hardware running the program. The original analysis is proven to give sound bounds on the energy consumption of the analysed program[18]. The analysis results in a lower and upper bound on energy consumption of the program, parametrized by both regular function parameters and specific hardware models. The hardware components are described in the next section.

A very simple program to switch a connected radio on performs actions concerning the hardware, and as such must consume some amount of energy. Assuming that switching the radio on consumes 50 J of energy would mean the following program also consumes 50 J of energy:

```
void main() {
  //Switching the radio on consumes 50 energy
  radio_on();
}
```

Loops in a program are analysed by relying on user-specified lower and upper bounds on the number of iterations. Loop iterations are evaluated on how they affect the hardware, and the maximum and minimum values of the energy consumption for each hardware model are taken as the upper and lower bound respectively. Results of the analysis therefore strictly depend on both the hardware components and the bounds defined by the user. The program is never executed or evaluated, so the analysis is static.

Branching paths in the code are treated similar to loops: each path is analysed separately, and the maximum and minimum values are taken as the upper and lower bounds respectively. Lastly, arithmetic and other basic operations also consume energy: a CPU hardware component may be defined to describe the energy and time consumed by these basic operations.

Continuing the simple radio example, assuming that there is another function to switch the radio off consuming 20 J of energy, a conditional branch that switches the radio off or on would be evaluated entirely. Taking the minimum for the lower bound, the branch that switches the radio off would be chosen, whereas the opposite would be true of the on-switching branch.

```
void main() {
  //Branch on some condition
  if (cond) {
    radio_on();
  } else {
    radio_off();
  }
}
```

Analysing this example would result in 20 J of energy as the lower bound, and 50 J of energy as the upper bound. The situation gets more complicated if a loop is introduced: consider a loop that runs for 10 iterations. At the first and sixth iterations the radio is switched off, but at the fifth and tenth iterations it is switched back on.

```
void main() {
  //Loop running 10 iterations
  int x = 0;
  while(x < 10) {
    //The radio is switched on for only two iterations, spaced out
    if (x == 4 || x == 10) {
      radio_on();
    } else {
      radio_off();
    }
    x++;
  }
}
```

The problem with analysing a loop is that each iteration may have a vastly different energy consumption. In the example, the radio is on for two of the ten

iterations, which consequently consume more energy than the iterations where it is switched off. To deal with this problem, and to guarantee the soundness of the resulting bounds on energy usage, the energy analysis relies on fixpoints of the energy consumption: at some iteration, the energy consumption of each hardware component is at a minimum, and the same holds for a maximum. By multiplying these values with the number of iterations, a sound lower and upper bound is found, though these are not guaranteed to be very tight: if a loop consumes a massive amount of energy in one iteration, but almost nothing in the others, both the lower and upper bounds will be vastly under- and over-estimated, respectively.

2.1.2 Hardware Modelling

The energy analysis depends on hardware component models (HCMs) for information about energy and time consumption of the hardware a program will run on, which allows the user of the analysis to abstract the hardware to any desired level, rather than it being fixed at either end of the technical-theoretical spectrum. The more the models correspond to reality, the more accurate the analysis will be, though also the more specialized. Each model describes one component consisting of a state, functions which may alter this state, and a ϕ-function. Functions consume a constant amount of energy and time when they are called, whereas the ϕ-function describes the power draw of the component per time unit, which can depend on the state.

The analysis aggregates the energy consumed by each individual hardware component model, parametrized by variables used in loop bounds and hardware states. Time consumption analysis is used to determine how long operations take. When one hardware component is performing some action, the others are still active in some state and passively consuming energy, which is referred to as time-dependent energy consumption in the original analysis paper[1]. The distinction is made explicit between active and passive energy usage: functions called in the hardware actively consume energy, whereas the component itself passively consumes energy (which may be based on its state). To determine bounds on the energy consumption of the hardware models, minima and maxima are taken over branching states: an ordering must be present.

In the context of the radio example, instead of treating the radio as only consisting of two functions, a ϕ-function and a state can be added. Taking the state as a single boolean, on or off, it becomes possible to define ϕ as a function that consumes 20 additional joules when the radio is on, and just 1 when it is off. An example of how such a hardware component object might look in C++:

```
class radio {
  //State
  int on = 0;

  //Modifying functions
  void on() { on = 1; }
  void off() { on = 0; }

  //Phi-function
  int phi() { return 1 + on * 20; }
};
```

In this case, if the loop-based example were using this component instead of just the functions *radio_on* and $radio\_off$, the energy consumption for each subsequent iteration would be slightly higher: assuming the functions take 1 millisecond unit each, every call would have resulted in the passive energy consumption by the component to increase as well. Thanks to the overestimation, however, this only matters in the fixpoint states.

2.1.3 Analysis Limitations

The energy analysis is restricted, which limits its direct applicability. Most prominently, the analysis rules are limited to a set of default language operations lacking several features common to programming languages. Missing features are data values other than positive integers, extended or structured data of any kind, recursion, manipulation of data beyond local variables and function parameters, and a notion of scoping. This restriction is shared with EcaLogic and its own language[3], which implement the energy analysis rules.

The analysis is strictly parametrized in the hardware component models and function parameters, with no other variables directly allowed unless they have a definite relation to function parameters. This limitation means the analysis cannot depend on values read from a file, received from connections, or otherwise unrelated to the direct usage of the program or a function call.

In the examples above, a variable x was used for the loop, but it was not explicitly given as a parameter for the containing function. In the cases where x is not outright known, some relation to parameters would have to be established: for instance, x could be taken from the command-line arguments for the main function, or regular parameters for any other.

The hardware component models used by the analysis also have several restrictions. In order for the analysis to be able to handle the bounds on a loop, it is required that a 'bigger' state also corresponds to an increase in power draw: this is because determining the maximum (and minimum) consumption rates also depends on finding the maximum state of the component (referred to as the fixpoint of the loop). Energy consumed by hardware component functions has to be constant and may not depend upon function arguments, because parame-

ter dependency would make it significantly harder to determine maximum and minimum values: it is infeasible to simulate every possible argument or evaluate them in order to get a concrete value. Lastly, hardware states themselves may not use data values beyond simple integers.

What these restrictions mean for the radio component from the example is that each function may not consume energy based on any form of parameter: it may not consume more or less energy if it is switched off while it was already off to begin with. The ϕ-function would also not be allowed to return a lower value when the radio is on compared to when it is off.

2.1.4 Typing

The original energy analysis has only one type: natural numbers. When extending the analysis to include the types used by other, practical languages, a general type system has to be taken into consideration. Some languages are not, as a whole, type-safe, but many individual operations are, or they may demand other restrictions on the type itself. As a basic example: a function call used as an expression in a conditional only makes sense if the return type of this function can be evaluated as a boolean. Some operations can invalidate or ignore the type entirely in languages such as C or Python, which feature direct memory manipulation and dynamic typing respectively.

While the examples above only deal with integer values, this can hardly be assumed for every program. Using a floating point value instead would make the analysis impossible to run:

```
void main() {
  //Loop running 10 iterations
  float x = 0.0f;
  while(x < 1.0f) {
    x += 0.1f;
  }
}
```

In the case of energy analysis, some leeway may be granted to the type system, depending on the level of abstraction: when dealing with the more abstract hardware components for their energy usage, intrinsic arithmetic conversions or type casting would not need to be precisely modelled. Ergo, most of the type verification needed concerns enforcing some constraints upon the usage of different types to ensure the validity of the analysis itself, rather than during runtime of the program. For example, when using C-like unions, only the last-modified value is accurately depicted as such in memory: accessing other values may therefore result in undefined behaviour, which in turn may foul the analysis.

As an example, consider a union of an integer value x, and a floating-point value y:

```
union u {
  int x;
  float y;
};
```

Floating point values are generally represented in a vastly different manner than integer values. So, the operation in the following example would result in some poorly defined value for x:

```
union u test;

//Test now contains 2.61 as a floating point value
test.y = 2.61f;

//This interprets the data of y as an integer value
print(test.x);
```

For the Operational Semantic rules in this section and other rules in their individual sections, some identifiers and functions are used to formalize the type restrictions necessary for the energy analysis if more types were to be allowed. These are:

- *type* represents a general type. This may be any type: primitive, structured, pointer-to, defined, .., with the exception of the void type.
- *rtype* refers to the return type of a function. This may be any type, including the void type.
- ***typeof***(e) is used to retrieve the type of the specified expression or value e. This function is defined in greater detail below.
- ***subtype***(a) retrieves the subtype of an array, as it was declared.
- ***conv***(e, t) checks if expression/value e may be implicitly (without explicit casts) converted to the type t. If $typeof(e)$ equals t then this is true, otherwise it depends on the implicit conversions available to the compiler: for instance, converting an integer to a float is usually implicitly possible, but the reverse may not be the case, depending on rounding and availability.
- ***last***(u) denotes the last element modified of the union u. This is used to ensure that accessing union elements is only verifiable if the element is guaranteed to be of the proper format.

The ***typeof***(e) function has several possible return values, depending on the input:

- If e is an arithmetic operation or constant, the result is the type of the result of the operation.
- If e is a variable, the result is the type of the variable as it was declared.
- If e is an element of a struct or union, the result is the type of the element as it was declared.

- If e is a function call, the result is the return type of the function as it was declared.

Boolean values, used for conditionals in loops and branching statements, may not have a basic definition in some languages: the values can be interpreted in a certain manner to define boolean behaviour. In particular, zero-values are often false, while anything else counts as true; this holds for number-values and pointers, though precise restrictions may depend on the specific language or compiler. As such, in the rules, the identifier *boolean* is used to specify when a boolean-interpretable value is needed.

2.1.5 Rule Elements

The operational and energy-aware semantics, as well as the actual analysis rules, are described in proof-tree format, where each rule can be applied to specific statements or expressions. The analysis rules include required elements for a Hoare-based ruleset. Each set of rules is deterministic, but not collectively exhaustive: at any statement or expression, there will always be exactly one rule to apply if the analysis can proceed, or exactly zero if the operation is not supported. The energy analysis is performed, in general, by finding the analysis rule to apply, applying it, and repeating this operation until there is nothing left to analyse. The semantic rules are used to validate the program itself, though the energy-aware rulesets also describe their effects on energy consumption. It should be noted that an implementation of the analysis would primarily be based on the energy analysis rules, with the semantic rules used optionally for verification.

The precise notation of the energy analysis rules is similar to that of the operational semantics. Each rule has a set of pre-conditions and initialization, which are updated after applying the rule in the post-conditions. The elements in the upper part of each rule describe both restrictions that have to hold, and assignments of values used within the rule. The values, variables and special characters used are the following:

- Δ refers to the environment, which includes function definitions, as well as data-structure definitions such as structs or unions.
- σ is the variable state: variables and their values are stored here.
- Γ is the component state of the analysis. This contains instances of the hardware component models, which are updated in certain rules to reflect energy consumption.
- $\mathfrak{t}$ is a timestamp. This is used to compute the continuous energy consumption of the hardware components, and is updated for all components whenever a time-consuming action is performed (which, in practice, should be all actions: nothing is free).

- $C_{imp} :: \mathfrak{e}$ refers to the $\mathfrak{e}$ function of the CPU component C_{imp}. Other components are referred to by C as well, though with the component state as superscript and an index/identifier as subscript.
- $C_{imp} :: \mathfrak{E}_e$ refers to the energy consumption of e, in this case for the CPU component.
- $C_{imp} :: \mathfrak{T}_e$ refers to the time consumption of e, in this case for the CPU component.
- e, e_1 and e_2 refer to expressions in the program.
- a, u, e, s and f refer to identifiers for arrays, unions, enumerations, structs and functions respectively.
- S_1 and S_2 are statements in the program.
- n and m represent intermediate results of expressions, and therefore values.

For updates in the various states a specific notation is used. The variable state, individual hardware component states, and the function environment are updated like this:

$$\sigma[x \leftarrow n]$$

Whereas the energy consumption of an individual hardware component is updated as follows:

$$[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_e]$$

2.1.6 Analysis Operational Semantics

This section lists the operational semantic rules of the original EcaLogic: the rules governing what effect each statement or expression has, without considering energy consumption. To avoid unnecessarily cluttering the energy analysis rules, type-system restrictions are confined to this section. These rules have been slightly modified from the original paper[1] to be less language-specific: several minor typing checks have been added, and the syntax was made to look more like Java or C.

The operation semantics for semantics are in Figure 1, whereas the expression semantics are in Figure 2.

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle}{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma', \Gamma'\rangle}\ (\text{sExprAsStmt}) \qquad \frac{}{\Delta \vdash \langle \textbf{skip}, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma, \Gamma\rangle}\ (\text{sSkip})$$

$$\frac{\Delta \vdash \langle S_1, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle S_2, \sigma', \Gamma'\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle S_1;\ S_2, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}\ (\text{sStmtConcat})$$

$$\frac{n = 0 \qquad \textbf{\textit{conv}}(n, \textit{boolean}) \qquad \Delta \vdash \langle e, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle S_2, \sigma', \Gamma'\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle \textbf{if(}\ e\ \textbf{)}\ S_1\ \textbf{else}\ \ S_2, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-False})$$

$$\frac{n \neq 0 \qquad \textbf{\textit{conv}}(n, \textit{boolean}) \qquad \Delta \vdash \langle e, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle S_1, \sigma', \Gamma'\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle \textbf{if(}\ e\ \textbf{)}\ S_1\ \textbf{else}\ \ S_2, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}\ (\text{sIf-True})$$

$$\frac{n = 0 \qquad \textbf{\textit{conv}}(n, \textit{boolean}) \qquad \Delta \vdash \langle e, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle}{\Delta \vdash \langle \textbf{while(}\ e\ \textbf{)}\ S_1, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma', \Gamma'\rangle}\ (\text{sWhile-False})$$

$$\frac{n \neq 0 \qquad \textbf{\textit{conv}}(n, \textit{boolean}) \qquad \Delta \vdash \langle e, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle S_1;\ \textbf{while(}\ e\ \textbf{)}\ S_1, \sigma', \Gamma'\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}{\Delta \vdash \langle \textbf{while(}\ e\ \textbf{)}\ S_1, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma'', \Gamma''\rangle}\ (\text{sWhile-True})$$

$$\frac{\Delta[f \leftarrow (e, \Delta, x)] \vdash \langle S, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma', \Gamma'\rangle \qquad \textbf{\textit{conv}}(e, \textit{rtype})}{\Delta \vdash \langle \textit{rtype}\ f(x)\ \{\ e\ \}\ S, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma', \Gamma'\rangle}\ (\text{sFuncDef})$$

Figure 1: Statement semantics.

$$\frac{}{\Delta \vdash \langle c, \sigma, \Gamma\rangle \Downarrow^e \langle c, \sigma, \Gamma\rangle}\ (\text{sConst}) \qquad \frac{}{\Delta \vdash \langle x, \sigma, \Gamma\rangle \Downarrow^e \langle \sigma(x), \sigma, \Gamma\rangle}\ (\text{sVar})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle e_2, \sigma', \Gamma'\rangle \Downarrow^e \langle m, \sigma'', \Gamma''\rangle \qquad C_{imp} :: \boxdot(n, m) = p}{\Delta \vdash \langle e_1 \boxdot e_2, \sigma, \Gamma\rangle \Downarrow^e \langle p, \sigma'', \Gamma''\rangle}\ (\text{sBinOp})$$

$$\frac{\Delta \vdash \langle e_1, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle \qquad C_{imp} :: \boxdot(n) = m}{\Delta \vdash \langle \boxdot e_1, \sigma, \Gamma\rangle \Downarrow^e \langle m, \sigma', \Gamma'\rangle}\ (\text{sUnOp})$$

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma'\rangle \qquad \textbf{\textit{conv}}(n, \textbf{\textit{typeof}}(x))}{\Delta \vdash \langle x = e, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma'[x \leftarrow n], \Gamma'\rangle}\ (\text{sAssign})$$

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

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma\rangle \Downarrow^e \langle a, \sigma', \Gamma'\rangle \qquad \Delta(f) = (e_1, \Delta', x) \qquad \Delta' \vdash \langle e_1, [x \leftarrow a], \Gamma'\rangle \Downarrow^e \langle n, \sigma'', \Gamma''\rangle}{\Delta \vdash \langle f(e), \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma', \Gamma''\rangle}\ (\text{sCallF})$$

$$\frac{\Delta \vdash \langle S, \sigma, \Gamma\rangle \Downarrow^s \langle \sigma', \Gamma'\rangle \qquad \Delta \vdash \langle e, \sigma', \Gamma'\rangle \Downarrow^e \langle n, \sigma'', \Gamma''\rangle}{\Delta \vdash \langle S, e, \sigma, \Gamma\rangle \Downarrow^e \langle n, \sigma'', \Gamma''\rangle}\ (\text{sExprConcat})$$

Figure 2: Expression semantics

2.1.7 Analysis Energy-Aware Semantics

The original, slightly modified energy-aware semantics are as in Figure 3 for statements and Figure 4 for expressions. In this case, only the syntax has been modified to match that of the operational semantics. To avoid cluttering the image further, elements from the rules for the regular semantics are omitted: this includes the typing restrictions and union safety restriction. It may be assumed that both these rules and the equivalent regular semantic rules need to apply.

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

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

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

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

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

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

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

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

Figure 3: Energy-aware semantics for statements, adapted from [1]

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

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

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

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

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

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

Figure 4: Energy-aware semantics for expressions, adapted from [1]

2.1.8 Analysis Hoare Logic Rules

While the semantics describe what each statement can and must do, the actual energy analysis is described as a Hoare logic with a set of rules. Energy consumption information is aggregated in the Hardware Component Models, and each rule describes how this information is updated for a specific statement or expression. Because the analysis considers all possible paths of the control flow of the program, there are no seperate rules for branching paths such as if/else statements. Instead, each path is evaluated, and the appropiate minimum or maximum is chosen for the lower or upper bounds respectively.

The set of analysis rules is written in a different notation from the semantic rules, though several items from Section 2.1.5 are still used. The pre- and post-conditions are given as Hoare-logic conditions instead of tuples, and reduction is not explicitly indicated. While the pre- and post-conditions only list the elements applicable to the analysis, other items corresponding to program correctness and semantics are implicitly considered also. That is to say, a program has to fully fit the semantic rules in this chapter, before the analysis may be applied.

Additional symbols and functions used in the analysis rules are formally defined in the original paper[1], but superficially they are:

- ρ, ρ_1 and ρ refer to the variable environment. This is used to store variables and their values, but also for substituting variables used in specified bounds with their parameter-related values.
- $\mathbf{lub}(\Gamma_1, \Gamma_2)$ is a function used to calculate the least upper bound of two sets of hardware component model states. A similar function is used to calculate the highest lower bound.
- $\mathbf{process\text{-}td}(\Gamma, t)$ applies a difference in time to a set of hardware component states. This effectively means calling the ϕ-function for each component for each time unit in t.
- $\mathbf{ci}_i(S)$ denotes the component iteration function over component C_i: the result is the new C_i after evaluating the statement S.
- $\mathbf{fix}_i(S)$ refers to the fixpoint of component C_i in statement S. During evaluation of the statement, the state of the component may change; if the state reaches the same state as it was at an earlier point in the statement, this state is the fixpoint of the component. Minimizing or maximizing such a state is used to determine the energy consumption of an entire loop.
- $\mathbf{wci}_i(S)$ is the worst-case iteration function. This function applies both the **ci** and **fix** functions to determine the worst possible iteration in the loop: the iteration where either the minimum or maximum fixpoint occurs.
- $\mathbf{oe}(..)$ applies the **wci** function to the component states and number of iterations as given in the lower and upper bounds, to determine the energy consumption of a loop. The result is an over-estimation of the worst case: specifically, the case where each loop iteration consumed the same amount of energy as the minimum and maximum fixpoints in the loop.

The original ruleset can be found in Figure 5. Additions to these rules are described in their own chapters where applicable. A full listing of the final set of rules, after any additions and changes, can be found in Appendix A.

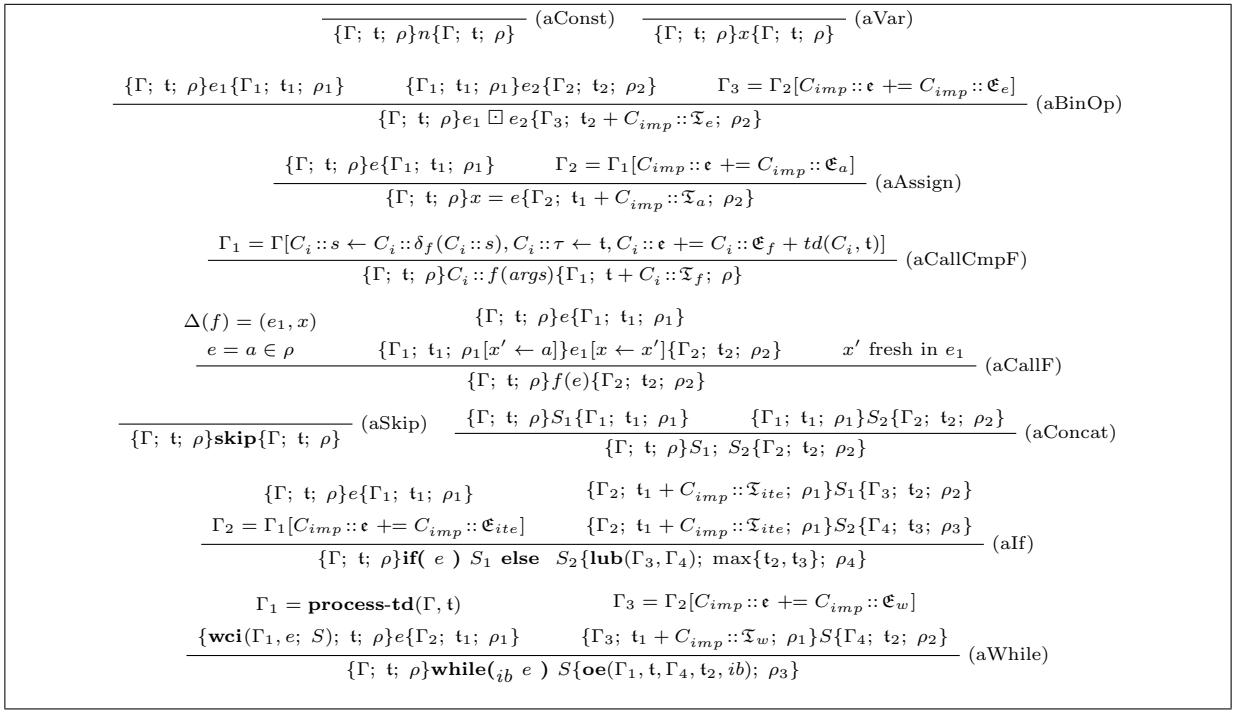

Figure 5: Original energy analysis rules, adapted from [1]

2.1.9 Application Example

Application of the analysis rules is deterministic, so there will be one rule at most to apply to a given statement. Consider the example from earlier in the chapter, extended with a function parameter:

```
void main(int n) {
  //Loop running n iterations
  int x = 0;
  //Bound in (lower, upper) is: (n, n)
  while(x < n) {
    //The radio is switched on for only two iterations, spaced out
    if (x == n/2 || x == n-1) {
      radio_on();
    } else {
      radio_off();
    }
    x++;
  }
}
```

The loop runs a guaranteed n iterations, defining the upper and lower bound as

this number; it may be assumed that this information is communicated to the analysis in some way. If the analysis is performed on this function, it would go through the statements sequentially.

The function has at its highest level only two statements: the assignment of x, setting it to zero, and the while loop. The first statement is handled with the application of just the assignment rule aAssign followed by aConst for the zero value. The while-loop, on the other hand, applies the aWhile rule, which looks at the body of the while-loop in full. In order to estimate the energy consumption of the loop, the analysis determines the fix-point of the component states. The highest state of the radio component is when it is turned on, so the iteration where the $x == n/2$ and $x == n - 1$ branch is entered would naturally apply. However, as a branch, the energy analysis has to look at both sides of the if-else statement conform the aIf rule, which results in the higher-consuming branch (where the radio is turned on) to be considered the result of the branch. This means that each iteration where the radio would be turned off is over-estimated into turning the radio on instead, which makes the fix-point easy to find, as each iteration is considered pretty much equal.

The result of the analysis is, after applying the rules, the energy consumption from the radio being turned on multiplied by the bound of the loop, added to the consumption from the initial assignment to x. Assuming the radio being turned on and this action consuming 20 joules, the passive power usage being zero, and the assignment costing 10 joules, energy consumption of the *main* function would then be $10 + 20nJ$.

2.2 EcaLogic

EcaLogic[3] is a tool implementing the original energy analysis in Scala, and extending this tool to implement the extended analysis for the C language is the second objective of the thesis. With the 'energy analysis' the theoretical aspect of the analysis is meant, 'EcaLogic' refers to the original tool, and the extended implementation will be referred to as 'EcaLogic-C'. Both the energy analysis and EcaLogic have two inputs: a program, and hardware models. The models are used to ascribe energy and time consumption information to hardware components, which is in turn used to analyse the consumption of the program. More details for the original analysis are in Section 2.1, as well as the original paper[1].

The overall structure of the original EcaLogic tool looks as in Figure 6.

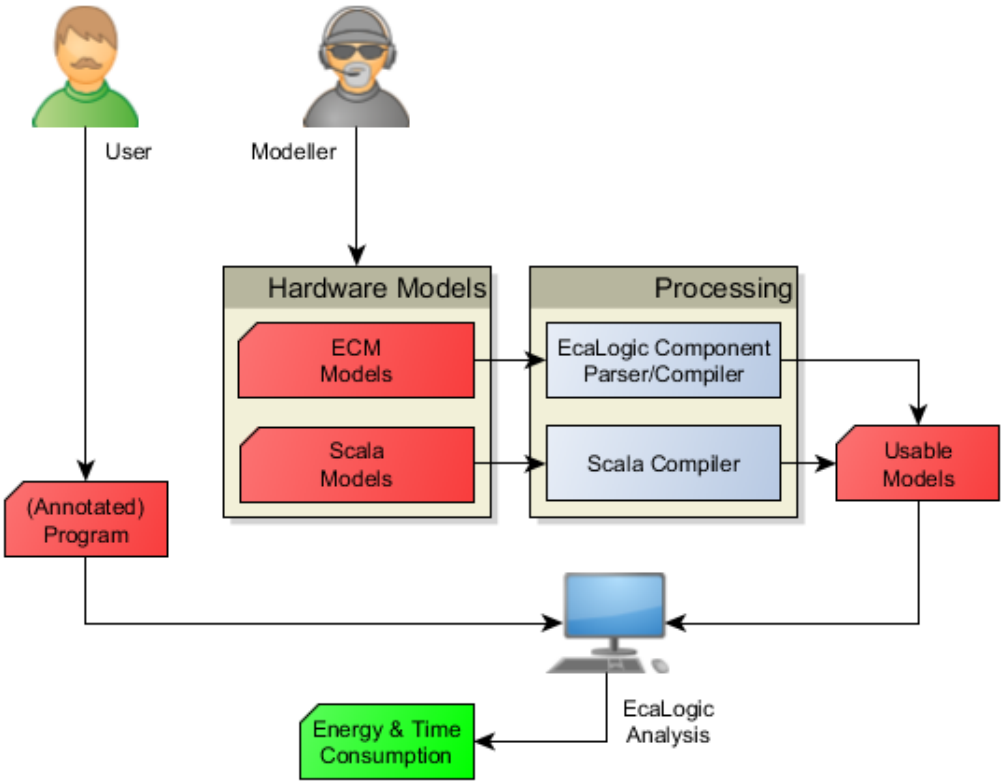

Figure 6: Structure of EcaLogic operation

2.2.1 Original Language

EcaLogic operates on a basic, While-like language with little more than the standard functionality of programming languages: it lacks support for recursion, global variables, scoping, data values beyond natural numbers, a notion of pointers or arrays, or structured data of any kind. One unique feature of the language is calling functions of the hardware components, which are generally treated as simple objects on the surface. For reference, a partial grammar of the language is shown in Figure 7.

EcaLogic applies the energy analysis rules to gain an upper bound on the energy consumption of the program, parametrized in variables used and the hardware components modelled. The tool was later extended to also provide a lower bound. In order to handle loops in the code, it is required for the programmer to define lower and upper bounds on the number of iterations for each loop.

```
<program> ::= {<comp-imp> <sep>} {<fun-def> <sep>}
<comp-imp> ::= ‘import’ ‘component’ id {‘.’ id} [‘as’ id]
<fun-def> ::= ‘function’ id [‘(’ [id {‘,’ id} ]‘)’] <fun-body>
<fun-body> ::= ‘:=’ <expr>
    | <stat-list> ‘end’ ‘function’
    | <empty>
<stat-list> ::= {<statement> <sep>}
<statement> ::= ‘skip’
    | id ‘:=’ <expr>
    | <fun-call>
    | ‘if’ <expr> ‘then’ <stat-list> ‘else’ <stat-list> ‘end’ ‘if’
    | ‘while’ <expr> ‘bound’ ‘(’ <expr> ‘,’ <expr> ‘)’ ‘do’ <stat-list> ‘end’ ‘while’
    | ‘{’ <annot-elem> {‘,’ <annot-elem>} ‘}’ [<statement>]
<fun-call> ::= [id ‘::’] id ‘(’ [<expr> {‘,’ <expr>}] ‘)’
<annot-elem> ::= id ‘<-’ <expr>
<expr> ::= <expr> <bin-op> <expr>
    | <expr> ‘^’ numeral
    | id
    | <fun-call>
    | ‘(’ <expr> ‘)’
<bin-op> ::= ‘or’ | ‘and’ | ‘=’ | ‘<>’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’ | ‘+’ | ‘-’ | ‘*’ | ‘/’
<sep> ::= ‘;’ | end-of-line
```

Figure 7: Original EcaLogic language grammar, taken from [3].

The hardware components can be written in a language based on the simple grammar in Figure 7, the original partial grammar of which can be viewed in Figure 8. Files for this language are referred to as ‘ECM’ files in Figure 6.

```
<component> ::= {<class-imp> <sep>} ‘component’ id [‘(’ [<var-def> {‘,’ <var-def>}] ‘)’]
{<member> <sep>} ‘end’ ‘component’
<class-imp> ::= ‘import’ ‘class’ id {‘.’ id} [‘as’ id]
<var-def> ::= id ‘:’ numeral ‘..’ numeral
<member> ::= ‘initial’ id ‘:=’ numeral
| <fun-def>
| <comp-fun-def>
<comp-fun-def> ::= ‘component’ ‘function’ id [‘(’ [id {‘,’ id}] ‘)’] [<uses-clause>] <fun-body>
<uses-clause> ::= ‘uses’ numeral ‘energy’ [numeral ‘time’]
| ‘uses’ numeral ‘time’ [numeral ‘energy’]
```

Figure 8: Original EcaLogic hardware component language grammar, taken from [3].

In addition to ECM files, the models may also be written in Scala, though this requires a direct implementation of the interface and recompilation of the tool.

2.2.2 EcaLogic Limitations

Certain elements of EcaLogic limit its applicability to ‘real-world’ programs and systems. Foremost among them is its restriction to the custom language. This restriction also holds for the analysis, but it is more general and therefore less pronounced in that case. The language only accepts natural numbers as values and lacks any form of type system, which severely limits using it for any complex system, or even a program relying on arrays. Also lacking are unary operators, bit-wise binary operations, the ability to define groups of values (eg.

structs), control structures such as for-loops or case-switch statements, explicit return values for functions, recursion, any form of memory management, global variables, explicit or implicit jumps in the code, and scoping. The need to rewrite a program to this language in order to be able to analyse it decreases the usability of the tool substantially.

Other, lesser, restrictions are angled towards ease of implementation or time constraints. Only one file may be analysed at a time, inclusion of files into programs is unsupported, there exists no true compiler for the language so it is only superficially checked and is computationally expensive to interpret, and it is not possible to model existing functions or attach energy consumption information to them. Lack of reuse of analysis results means that an entire function definition is re-analysed on every call to this function, which greatly increases the time required for the analysis when going beyond a small amount of functions.

3 Data Structures

The original energy analysis[1] and its implementation[3] only support natural numbers as data, without provisions for negative numbers, floating-point values, or any other data structures. Limiting the tool to just natural numbers makes analysing a practical program much harder, so this chapter discusses adding additional data types and structures to the system: starting at extending the primitive data types available, and going on towards structured data types such as arrays.

3.1 Primitive Data Types

The original analysis used by EcaLogic is restricted to natural numbers for all values. No analysis elements depend directly on the values of variables; the restriction was imposed out of consideration for the implementation rather than the analysis. Indeed, the ordering of the values matters much more for the analysis than the type of the data. Extending computations from natural numbers to integer numbers would not break any element of the analysis in and of itself. However, removing the restriction introduces the issue of overflow: if states or other values on which the analysis depends would encounter overflow, it would violate the ordering restrictions put in place for them. In that regard, then, leaving the responsibility of maintaining these restrictions into the hands of the user simplifies matters. Hardware Component states are the only element of the analysis strictly susceptible to overflow, and since their restriction of ever-increasing energy usage with a larger states can be automatically checked, problems arising from overflow are noticed in time and can be reported back to the user.

The same reasoning applies to the addition of floating point values to the list of usable data types. The ordering and other constraints are still preserved when allowing non-integer values, as long as they are finitely representable: the main issue in this case comes from the actual computation of the analysis, since non-integer values are less convenient to work with. Disregarding this, another issue are the inherit limitations of floating point values in their hardware representation: the number of bits is always limited, so not all values are representable. Since the actual limitations are heavily dependent on factors like the specific system or compiler, abstracting these values to the more accurate but less realistic domain of large double-values would be preferable.

Character-string constants are a special case. While characters themselves are usually treated as very limited integer values, a string constant is implicitly an array of characters in languages such as C, and can be a full object in object-oriented languages, such as Java. In the C case, treating them as a plain array with its supported operations is feasible, and generalizes them more easily for any of the elements of the analysis utilizing them.

3.2 Arrays, Structures, Unions & Enumerations

The most commonly used basic data storage type, arrays are strictly defined regions in memory for storing a list of another data element. Since they have to be explicitly sized before usage, their capacity is always known, even when dynamically allocated, although the ability to use variables for sizing them in this manner might complicate their size.

With the size known as a constant at the time of initialization, treating them as a set of variables of the specific subtype greatly simplifies handling the individual values. For example, having a variable consisting of an array with two elements, a simple method of handling the data would be to consider each element as its own variable. Treating them as a set keeps the methods used for the analysis applicable, provided that they also hold for the data type contained in the array.

Local stack-based arrays initialized with a length defined by variables are a special case: since they are designated as optional in some languages (such as C) and outright unsupported in others (like Pascal), precise semantics may differ. As long as the final size is determinable at a specific point, however, their values are still usable in steps of the analysis without risking access to undefined sections of memory.

In the C programming language, array operations include the following:

```
//Declarations
int list1[10];
int* list2 = alloca(10 * sizeof(int));
int* list3 = malloc(10 * sizeof(int));

//Assignments
list1[1] = 5;
list2[3] = 6;
```

The declaration methods have subtle differences, but can be seen as pretty much equal for the purpose of the analysis. The first declaration type for *list*1 allocates an array on the stack (local scope), with size defined by a constant value. The second declaration for *list*2 also creates an array on the stack, but the allocation size may contain variable references, since it is a call to the *alloca* function. The third and last allocation for *list*3 may also contain variable references, but allocates the array on the heap instead of the stack, which means it persists beyond the current scope. Access to elements of the array happens the same in all these cases, and the syntax is used for the new rules defined in this chapter.

Similar to arrays, in their basic form, structures and unions also represent a solid set of variables. Precise semantics notwithstanding, they may therefore be considered under the same clause as arrays in the basic case. For the non-basic cases, such as recursive and pointer-based manipulation of arrays or structures, separate changes may have to be made, which are described in more detail in

the next two chapters. One part of the semantics of unions is given special consideration: since the fields occupy the same space in memory, changing one value also changes the other values to however they would interpret the new memory. Dealing with values that are ill-defined poses too many risks for the analysis, so unsafe operations are treated as undefined behaviour.

Structures and unions are defined in almost the same way in the C programming language:

```
struct s {
  int x;
  int y;
};

union u {
  int x;
  int y;
};
```

The only difference between definitions is the keyword. Access to structures and unions is done via a plain dot operator: $s.x$ would refer to the x element of struct s. The syntax is also used for the rules introduced in this chapter.

Enumerations in most languages are, in essence, nothing more than a set of predefined constant values. In object-oriented languages they may also represent distinct objects or namespaces. In all cases they come with the added benefit of being able to define a type to the specific set of constants, which helps in preventing errors or just for ease of use. Since these restrictions are fully enforced during compilation, simply treating enumeration values as their respective constants makes them usable for the energy analysis without additional provisions.

3.3 Objects

Object-oriented languages offer the possibility of defining structures with their own variables, functions, sub-classes, etc referred to as objects. The main problem objects pose for the energy consumption analysis is also the feature they have in advantage of basic structures, namely their functions. Unlike globally defined functions, object-local functions have their own instance of the object and its variables to work with. As a result, the restriction of the energy analysis requiring parameter-variables as bounding values poses more problems here than it did for global variables.

Usage of the variables of an object for parts of the analysis is easy in and of itself: with their similar container-like status, treating them as structures with respect to their variables offers plain usage of them. In some languages (for example, C++), structures are treated as simple objects, so using objects and structures for the analysis would have many similarities. The same considerations regarding pointers and recursive objects then apply.

Solving the issue of using object-instantiated functions can be achieved by exploiting a common element in languages supporting them: object-based functions are treated as having an additional invisible parameter pointing to the instance of the object it is operating from, commonly referred to as the *this*-value. Any references to variables instantiating in the object are then implicitly translated to access statements to this pointer, which preserves the restriction on defining bounds based on variables. Coupled with the treatment of objects as elevated structures, they may then be used properly for analysis in all aspects.

3.4 Rule Additions

In order to be able to add extended data structures to the supported statements and expressions for the energy analysis, additions to both the semantic and the analysis rulesets have to be made. In particular, rules to define structures and unions are added to the semantic operations, whereas access and assignment rules are defined in all rulesets for arrays, structures and unions. Array declarations, while technically also a definition, are assumed to also consume energy, and as such they are also added to each set of rules.

The additions to the semantic rules describe what the operations imply and require in terms of the program (as well as energy consumption for the energy-aware semantics), whereas the added analysis rules exclusively concern their energy usage. For an explanation on the functions and symbols used, see Section 2.1; in particular, see 2.1.4 for the typing elements, 2.1.5 for the symbols present in all sets of rules, and 2.1.8 for functions and symbols unique to the energy analysis rules.

3.4.1 Semantic Additions

Adding arrays, unions, structures and enumerations to the semantic rules is necessary to allow analysis. The added rules, both for statements and expressions, are given for operational semantics in Figure 9, and for energy-aware semantics in Figure 10.

These rules are fairly straight-forward. With the exception of unions, each data structure is treated as a plain collection of variables. This means that assigning or accessing values takes the energy cost of plain assignment or access. Additional energy cost associated with accessing memory is ascribed in Chapter 4 instead.

Structure and union definitions store their identifiers and accompanying types in the program environment, whereas arrays store their length in the variable state. Enumerations do not have specific rules, because they are treated as collections of constants with some restraints, which are enforced not by the analysis but by the program compiler.

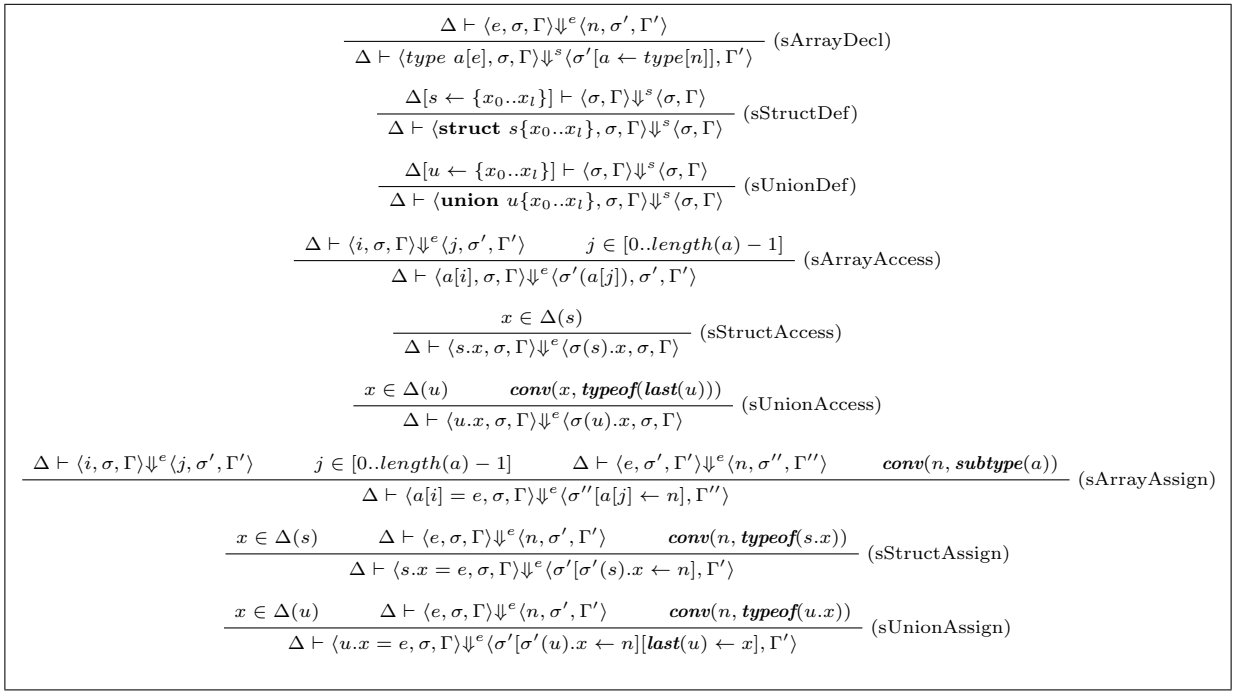

Figure 9: Operational data structure semantics

The union rules for accessing and assigning values are a special case because of the nature of unions. On an assignment, the last-modified value of the union in the analysis is set. Then, on access of any element, the type of the attempted access operation is checked against the last-modified value of the union. This is to ensure that the value will not be undefined, which would invalidate the analysis.

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle}{\Delta \vdash \langle type\ a[e], \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma'[a \leftarrow type[n]], \Gamma', \mathfrak{t}'\rangle}\ \text{(eArrayDecl)}$$

$$\frac{\Delta[s \leftarrow \{x_0..x_l\}] \vdash \langle \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle}{\Delta \vdash \langle \textbf{struct}\ s\{x_0..x_l\}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle}\ \text{(eStructDef)}$$

$$\frac{\Delta[u \leftarrow \{x_0..x_l\}] \vdash \langle \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle}{\Delta \vdash \langle \textbf{union}\ u\{x_0..x_l\}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle}\ \text{(eUnionDef)}$$

$$\frac{\Delta \vdash \langle i, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle j, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad j \in [0..length(a) - 1]}{\Delta \vdash \langle a[i], \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma'(a[j]), \sigma', \Gamma', \mathfrak{t}'\rangle}\ \text{(eArrayAccess)}$$

$$\frac{x \in \Delta(s)}{\Delta \vdash \langle s.x, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma(s).x, \sigma, \Gamma, \mathfrak{t}\rangle}\ \text{(eStructAccess)}$$

$$\frac{x \in \Delta(u)}{\Delta \vdash \langle u.x, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma(u).x, \sigma, \Gamma, \mathfrak{t}\rangle}\ \text{(eUnionAccess)}$$

$$\frac{\begin{array}{c}\Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e}\ \mathrel{+}= +C_{imp} :: \mathfrak{E}_a] \\ \Delta \vdash \langle i, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle j, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad j \in [0..length(a) - 1] \qquad \Delta \vdash \langle e, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^e \langle n, \sigma'', \Gamma'', \mathfrak{t}''\rangle\end{array}}{\Delta \vdash \langle a[i] = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma''[a[j] \leftarrow n], \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_a\rangle}\ \text{(eArrayAssign)}$$

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

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

Figure 10: Energy-aware data structure semantics

3.4.2 Energy Analysis Rule Modifications

In this section the additional energy analysis rules needed for extended data structures are described. This concerns only those energy-aware semantic rules that consume energy, which are the three assignment rules. Chapter 4 details the additional energy consumption caused by memory manipulation, and both adds to and improves this set of additional rules given in Figure 11.

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}i\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\}e\{\Gamma_2;\ \mathfrak{t}_2;\ \rho_2\} \qquad \Gamma_3 = \Gamma_2[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\{\Gamma;\ \mathfrak{t};\ \rho\}a[i] = e\{\Gamma_3;\ \mathfrak{t}_2 + C_{imp} :: \mathfrak{T}_a;\ \rho_2\}}\ \text{(aArrayAssign)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}e\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\{\Gamma;\ \mathfrak{t};\ \rho\}s.x = e\{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_a;\ \rho_1\}}\ \text{(aStructAssign)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}e\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\{\Gamma;\ \mathfrak{t};\ \rho\}u.x = e\{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_a;\ \rho_1\}}\ \text{(aUnionAssign)}$$

Figure 11: Additional analysis rules for data structures

4 Pointers

Explicit memory manipulation through the use of pointers is possible in languages such as C, Pascal and Perl. In object-oriented languages, pointers can often be used, albeit implicitly, so consideration can be given to modelling their energy usage, even in languages where direct manipulation is not supported. In general, pointers allow programmers to use memory locations as values in their computations, with added language-dependent functionality, such as pointer arithmetic in C. As an example, Java implicitly uses pointers for its objects but does not offer direct manipulation, whereas C and derivatives allow for explicit and possibly unsafe direct manipulation of memory addresses, and Fortran features type-safe pointer operations.

4.1 Safe Operations

Using and modifying pointers can be unsafe, but several operations may depend on basic pointer arithmetic without inherit risk. Under regular, safe, usage, the following operations are understood:

- Dereferencing a pointer to a variable and using this new variable instead of the actual variable. For example, a pointer to an integer x can be dereferenced and used safely, provided no modifications to the pointer occur.
- Accessing structure or union elements can be accomplished by modifying the pointer to the start of the structure in memory. For example, if a structure contains three integer values, accessing the second one can be achieved by incrementing the pointer to said structure by the size of one integer value in memory. This usage is safe as long as the pointer does not exceed the bounds set by the total size of the structure.
- Accessing a specific index in an array can, similar to structures, be done by incrementing the pointer to the start of the array to the resolved index. Precise semantics may depend on the compiler or target system: additional dereference operations may be required to access dynamically allocated or multi-dimensional arrays. As with structures, this usage is safe as long as accessed indices remain within the scope of the memory allocated for the array.
- Using a pointer to some value instead of the actual value is a common method to improve efficiency: rather than storing or passing along the entire batch of data, a pointer to the location of this data is used to manipulate or read it. Provided this data is initialized and access operations are safe, usage of it does not differ much from simple variable manipulation.

- Dereferencing a pointer to a variable with a different type is needed for some applications: in particular, void-pointers are often used to transmit information of any type as a function argument or otherwise. Provided the dereference occurs to a variable with the same type as the original data or a guaranteed castable type, this usage is safe.

For the safe operations, energy analysis can be applied in a similar manner to their equivalent operations without pointers: assigning a value to a pointer-location classifies as a variable assignment, and so on. Modelling specific memory operations to require some measure of energy is preferred, however, since the explicit operations do differ on the hardware-level. As such, two operations would be added to the CPU component:

- $C_{imp} :: \mathfrak{E}_{mdec}$ to denote the cost of allocating memory, and
- $C_{imp} :: \mathfrak{E}_{macc}$ to denote the cost of accessing a location in memory, possibly after pointer arithmetic operations.

As an example of allowed pointer usage, consider a case where a struct in C is passed as a pointer to a function, and its second integer value is modified. Since these pointers fall within strictly defined memory ranges, this behaviour would be allowed.

```
struct s {
  int x;
  int y;
};

void f(struct s* t) {
  //This operation would be allowed
  int z = *(t+sizeof(int));
}
```

4.2 Unsafe & Undefined Behaviour

Unsafe pointer operations are those that do not have the same guarantees that the safe operations have, or those that may lead to unpredictable behaviour. Such behaviour may fall under direct usage of pointers as values in operations, but also the usage of functions that depend on some quality of the pointer, which may not always hold. Unsafe operations are the following:

- Attempting to access memory outside of declared ranges. This includes accessing an invalid array index and accessing pointers to structures outside of their size, but also accessing a simple incremented pointer that was not declared as any sort of data structure. This operation can either result in a runtime error or return whatever is in the specific memory location, which is unpredictable.
- Dereferencing a pointer to use as a value with an incompatible type. For

example, attempting to dereference a pointer to a floating-point value into an integer variable. The same holds for functions that implicitly demand a specific pointer type, but also accept others, such as some generic functions utilizing void-pointers. The end result of these operations is undefined.

- Using bounded values in an unbounded manner. This can occur when a function takes a pointer argument to an array, but performs no checks on the actual bounds of the array itself. If the bound of such a function depends on the contents of array, which is the case in some string manipulation functions, the assumption that this condition ever holds has to be made explicit; otherwise it may cause an invalid memory access error similar to the first point in this list. Some compilers (such as the Microsoft .NET C compiler) can provide compilation warnings for this case.

These cases of unsafe pointer usage can easily invalidate the analysis, so they are not supported.

4.3 Rule Additions

By treating simple pointer-based operations as no different from their regular variations, no additional rules would need to be added to the semantic rules. However, since it is desired to attach energy consumption to memory manipulation operations via the CPU component, some rules will need to consume an additional amount of energy. These rules are the ones dealing with arrays, unions and structures; explicit pointer arithmetic is treated as a regular binary operation where applicable.

Furthermore, the additional semantic rules for data structures in Section 3.4 that formerly consumed no energy now do. Extra energy analysis rules have to be added for that reason, in particular for access to structured data types, and declaration of an array.

Two new CPU operations are introduced for these additional rules: memory declaration, and memory access. Memory declaration is denoted with $C_{imp} :: \mathfrak{E}_{mdec}$ and $C_{imp} :: \mathfrak{T}_{mdec}$ for energy and time consumption respectively, whereas memory access uses $C_{imp} :: \mathfrak{E}_{macc}$ and $C_{imp} :: \mathfrak{T}_{macc}$.

4.3.1 Semantic Additions

Extending the semantic rules so that they consume the additional energy results in the energy-aware semantic rules in Figure 12. The operational semantic rules are unaffected by the extension, so they are not listed here: they may be assumed to be the same as the ones given in Section 3.4.

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{mdec}]}{\Delta \vdash \langle type\ a[e], \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{s} \langle \sigma'[a \leftarrow type[n]], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{mdec}\rangle} \text{ (eArrayDecl)}$$

$$\frac{\Delta \vdash \langle i, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle j, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad j \in [0..length(a) - 1] \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\Delta \vdash \langle a[i], \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle \sigma'(a[j]), \sigma', \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{macc}\rangle} \text{ (eArrayAccess)}$$

$$\frac{x \in \Delta(s) \qquad \Gamma' = \Gamma[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\Delta \vdash \langle s.x, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle \sigma(s).x, \sigma, \Gamma', \mathfrak{t} + C_{imp} :: \mathfrak{T}_{macc}\rangle} \text{ (eStructAccess)}$$

$$\frac{x \in \Delta(u) \qquad \Gamma' = \Gamma[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\Delta \vdash \langle u.x, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle \sigma(u).x, \sigma, \Gamma', \mathfrak{t} + C_{imp} :: \mathfrak{T}_{macc}\rangle} \text{ (eUnionAccess)}$$

$$\frac{\Delta \vdash \langle i, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle j, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad j \in [0..length(a) - 1] \qquad \Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc} + C_{imp} :: \mathfrak{E}_{a}] \qquad \Delta \vdash \langle e, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^{e} \langle n, \sigma'', \Gamma'', \mathfrak{t}''\rangle}{\Delta \vdash \langle a[i] = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle \sigma''[a[j] \leftarrow n], \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_{macc} + C_{imp} :: \mathfrak{T}_{a}\rangle} \text{ (eArrayAssign)}$$

$$\frac{x \in \Delta(s) \qquad \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc} + C_{imp} :: \mathfrak{E}_{a}]}{\Delta \vdash \langle s.x = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle \sigma'[\sigma'(s).x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{macc} + C_{imp} :: \mathfrak{T}_{a}\rangle} \text{ (eStructAssign)}$$

$$\frac{x \in \Delta(u) \qquad \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc} + C_{imp} :: \mathfrak{E}_{a}]}{\Delta \vdash \langle u.x = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^{e} \langle \sigma'[\sigma'(u).x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{macc} + C_{imp} :: \mathfrak{T}_{a}\rangle} \text{ (eUnionAssign)}$$

Figure 12: Energy-aware pointer-based operation semantics

4.3.2 Energy Analysis Rule Modifications

The energy analysis rules in Figure 12 are extended with the new CPU operations, and the semantic rules in that section not previously attached to an energy analysis rule now receive one. These new and modified rules are given in Figure 13.

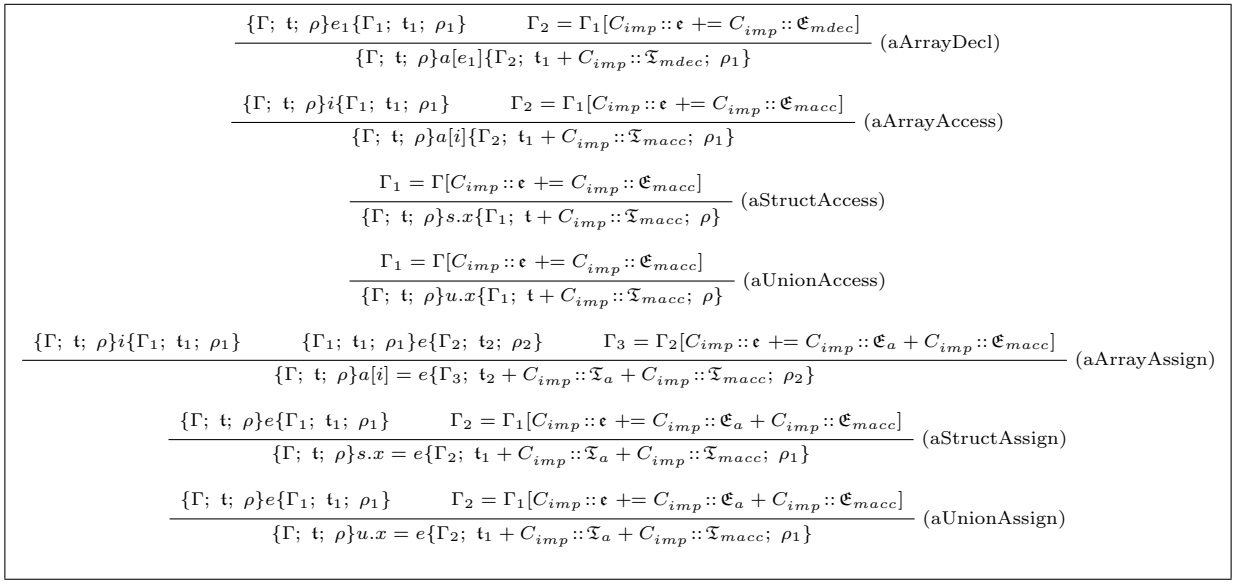

Figure 13: Additional analysis rules for data structures

5 Recursion

One of the common programming language elements not supported by the original energy analysis[1] and EcaLogic[3] is recursion: a function calling itself at some point during its execution. This can be seen as a loop in that the function executes for a certain amount of iterations, before some terminating condition is reached and the original calling function (at the 'top' of the execution tree) terminates.

It is possible to discern two distinct forms of recursion: single and multiple. Single recursion refers to a function that may call itself either zero or one time, whereas multiple recursion means the function can call itself any number of times, including zero. For the purpose of analysis, it can also be beneficial to look at a recursive function without its recursive part; to consider its energy consumption assuming that it never enters recursion.

A method of recursion that may apply to both forms is mutual exclusion (also known as indirect recursion): two or more functions that apply each other. Analysing either one of these functions requires the energy consumption of the other to be known, which in turn requires the other, leading to a loop. Due to the problems caused by this, it is assumed that mutual recursion is rewritten to direct recursion instead. A basic method of accomplishing a rewrite is by inlining the definition of one function into the other: this causes the function to effectively disappear, though it may cause no small amount of code duplication.

Other possible methods are those discussed by Kaser et al., and may be applied before the energy analysis takes place[19].

5.1 Function Signatures

While reasoning about the energy consumption of functions, the concept of function signatures can be brought to attention. The analysis, upon encountering a call to a function, analyses the entire body of this function to determine the energy consumption for the specific call. This leads to a significant amount of redundant analysis: every statement in the function body that does not have an impact on hardware component states would always result in the same active energy consumption, so analysing it more than once would not be necessary. Capturing, storing and applying the energy consumption of a function into a function signature would therefore provide a marked increase in efficiency. In a way, a full application of the analysis results in a signature for the entry-point function that was analysed, but this cannot be applied directly to other functions: for the entry-point function, the initial hardware states are always the same, whereas in a function call they may vary wildly.

A method to capture the effect of a function on the set of hardware component states is to collect operations on the component states, instead of their plain values. By recording each operation on the variables of the component state of each component, it becomes possible to 'replay' these operations whenever the function is called. Operations include direct assignment, such as setting a variable to zero, but also modifications like incrementation, division, etc. When dealing with branching paths or other control flow structures in the function, the lower or upper bound are computed in the same way, but the operations are stored separately. Then, when the function is called, stored operations can be applied on the current hardware states to arrive at a result, without redundantly analysing any part of the body of the function.

5.2 Recursion in Signatures

In addition to efficiency concerns, function signatures may also be applied to the problem of recursion for the energy analysis. A function signature for a non-recursive function consists of a series of operations for each hardware component. Recursive functions, on the other hand, call themselves a number of times, where each call consumes energy as well. However, since the signature of the function is not yet known, instead of a set of operations in the signature, a symbol can be inserted to represent the recursive energy consumption. For instance, consider the faculty function: $fac(n)$ consumes some energy, then calls $fac(n-1)$, until n equals zero. Assuming the energy consumption of a call to fac without the recursive statement is a constant c, the consumption of $fac(n)$ may be described in the following way:

$fac(n) = c + fac(n-1)$
$fac(0) = c$

Such a set of cases with a recursive element is known as a recurrence relation. These relations are generally solvable to a non-recursive format, which is a thoroughly explored topic in mathematics: methods range from basic[20] to advanced[21], and several automated solvers have been implemented, for instance, QEPCAD[6] and PURRS[7]. The prevalence of solving methods makes it feasible to reduce recurrence relations in function signatures to a non-recursive version, which may in turn be applied as the function signature for the whole recursive function.

5.3 Recursion Fixpoints

Function signatures offer possibilities to solve the energy analysis limitation for recursive functions, but they are not the only possible solution. Another possibility is to apply the same reasoning method used for analysing loops, which searches for a fixpoint in the iterations of the loop, and uses this to overestimate the energy consumption. Instead of searching for the fixpoint of the loop iterations, in a recursive function the fixpoint would instead be one call to the function, minus the recursive elements.

Consider, as an example, a simple faculty function:

```
int fac(int n) {
  if (n <= 1) return 1;
  else return n * fac(n-1);
}
```

Assuming that the true-branch of the if-statement consumes 20J of energy while the false-branch consumes 50J, the total energy consumption of a call to fac can be overestimated by multiplying this worst-case with the total number of calls that would result from a single call to the recursive function. In the case of the faculty function, recursion is entered at most $n-1$ times if n is greater than one, and always exactly once otherwise. If hardware component state changes occurred in the function, a fixpoint could be determined to ensure a suitable upper bound on the energy consumption of individual calls would be determined. The total energy consumption for fac, knowing the total number of recursive calls and the worst-case consumption, may finally be set at $50(n-1)J$.

One problem with this fixpoint-based method of handling recursion is also present with the loop-bound energy analysis: the worst case is always taken, so the bounds may not be very tight, especially if there is a large discrepancy between energy consumption of separate calls.

[6] QEPCAD is based here: `http://www.usna.edu/CS/~qepcad/B/QEPCAD.html`
[7] PURRS may be found here: `http://www.cs.unipr.it/purrs/`

6 Analysing C

The biggest impediment to practical usage of the original analysis[1] and the EcaLogic tool[3] is their restriction to a custom language with severe limitations. The elements discussed in the previous chapters focus on overcoming some of these limitations in general; this chapter details applying them to C in particular, by implementing the extensions applicable into the tool. The idea is to modify the tool so that it can be used to perform a static energy analysis of C-programs, ideally with as little modifications to the source code of said programs as possible. The new version of the tool is referred to as EcaLogic-C, and as an extension to EcaLogic, is written in Scala.

Broadly speaking, the following changes have been made to accomplish analysis of (a subset of) C: altering the actual input language to match C, adding the option to model hardware components in a real language or add energy and time information to existing functions, and changing the annotations for bounding and variable relations into a more easily verifiable format. This is on top of the changes to the analysis required by the extension of the tool to deal with different data types, recursion, and other elements described in previous chapters.

The sections in this chapter discuss how the required changes could be and were accomplished. Ultimately, the overall structure of the extended tool looks as in Figure 14.

In order to implement the additions to the energy analysis, changes to the internal representation of a program in EcaLogic had to be made for EcaLogic-C. The original abstract syntax tree was general enough to expand instead of replace, so that analysing EcaLogic's own language would still be possible in the new version. This can be viewed as in Figure 15: the AST of EcaLogic is a subset of the new AST (referred to as 'AST+') of EcaLogic-C, and the Frama-C plugin was developed to transform C into AST+.

6.1 Parsing

In order to analyse C, it is at the very least required to update the language. Rewriting the language parser of EcaLogic to instead parse C would be impractical, because the grammar for C is significantly larger than the grammar for the old simple language. Instead, an existing parser is used to read all program files, whereupon the Abstract Syntax Tree is converted into a format readable by EcaLogic-C.

The ideal choice for the parser is the one used by Frama-C, since it also has structures for dealing with any annotations present in ACSL, which in turn aids formal verification of the analysis. Thanks to the plug-in based nature of Frama-C, any plug-ins have easy access to the full C abstract syntax tree in CIL format, with some additions for the ACSL structures. Unfortunately,

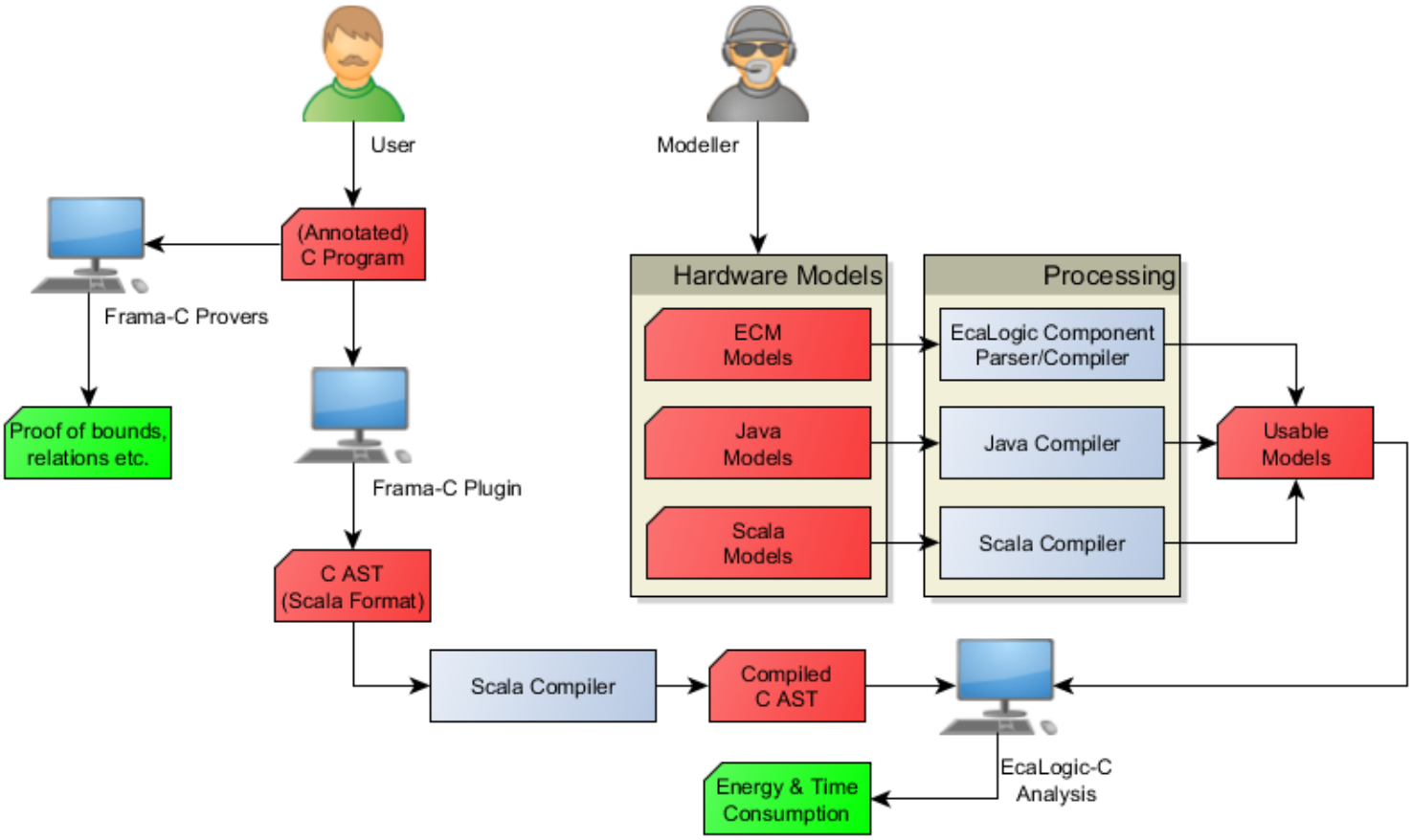

Figure 14: Structure of EcaLogic-C operation

due to the proprietary origins of the CIL format, it is mostly used for .NET framework applications, and no libraries exist for manipulating it in the Scala language.

To transfer the parsed C programs and annotations to EcaLogic-C, a plug-in script for Frama-C was developed. The plug-in works by transforming the C abstract syntax tree into Scala code that, when executed, constructs the tree in its entirety. The plug-in is entirely based on the internal representation of a program in EcaLogic-C, and as such, usage of it by other tools is not possible. In additional to conversion, the plug-in also looks for and handles custom annotations specific to the energy analysis, which are described in the next section. Annotations not using these custom predicates or logical functions are

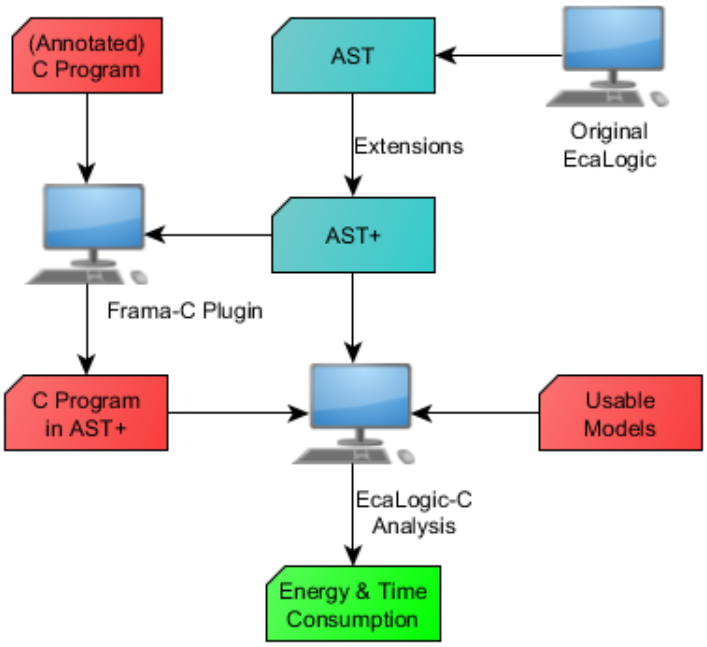

Figure 15: AST extension usage

ignored by the plug-in, so any other ACSL specifications in the input programs are untouched.

6.2 Annotations

Frama-C was selected as the annotation and verification tool to be used for annotating the C programs intended for analysis. The two main annotations used for the energy analysis are relations between non-parameter and parameter variables, as well as upper- and lower bounds on the number of iterations for all loops. Several custom predicates have also been added to help specify some elements of C useful for the analysis that would be difficult to specify in plain ACSL.

6.2.1 Variable Relations

Loop bounds specified for the energy analysis have to be based strictly on function parameter variables. However, usage of non-parameter variables is still possible, provided they are directly related to parameter variables. In order to

use non-parameter variables when specifying loop bounds, a relation needs to be established. In the original EcaLogic, this looked as follows:

```
{ X <- Y/2 }
```

This indicates that X is half of Y, as a relation. It is used as a direct substitution when defined bounds are applied, which means that a bound of X is simply substituted to a bound of $Y/2$. As such, this defined relation implies an equality of both sides. In order to define this bound in Frama-C, then, it would be sufficient to assert equality on both sides of the annotation. In ACSL, it can therefore look as follows:

```
//@ assert X == Y/2;
```

This assertion can then be verified using any of the Frama-C reasoning plug-ins, such as Jessie or WP. Additional lemmas or annotations required to prove it would then fall to the responsibility of the programmer: EcaLogic-C still assumes they are correct, but they can be formally verified as-is, rather than having to rewrite them to a more suitable format.

It is still needed for the analysis to differentiate between these custom assertions and regular equality assertions, so a predicate is defined for easy recognition by the tools. So, instead of the above, a relation is specified as follows:

```
//@ assert ecarelation(X, Y/2);
```

Worth noting is that the Frama-C plug-in for EcaLogic-C only looks at the name and arguments of the predicate, not its application. As a result, the actual definition of the predicate does not matter, allowing the user to provide their own definition to the predicates, which may aid them in formal verification.

6.2.2 Loop Bounds

For the energy analysis, it is required to know the maximum and minimum number of iterations for each while loop in the program. Previously, EcaLogic required the bounds to be defined in the following manner:

```
while (expression) bound (upper, lower) ...
```

Which indicates that the number of iterations for the while loop is always lower than or equal to *upper*, and always higher than or equal to *lower*. Unfortunately, Frama-C or ACSL has no convenient shorthand to reason about the number of iterations for a loop, but it is possible to define the notion of bounds by using ghost-annotations and assertions. Ghost-annotations are code statements that may not affect the actual program, but may be used for analysis elements; they act as a form of read-only code for ACSL and the automated reasoning plug-ins. A possible way of formalizing the upper and lower bounds on the number of iterations would then look as follows:

```
int k = 10;
//@ ghost int c = 0;
//@ ghost int lower = k;
//@ ghost int upper = k;
while (k > 0) {
  --k;
  //@ ghost c += 1;
}
//@ assert c <= upper;
//@ assert c >= lower;
```

The actual code in this example performs a while-loop that runs for 10 iterations, defined by the variable k. Using ghost annotations the values c, *lower* and *upper* are defined, where c represents the iteration counter, and *lower* and *upper* are defined as k, since k is both the lower and upper bound in this example. At each iteration c is incremented, and at the end assertions are made to ensure c lies within the defined lower and upper bounds.

With these assertions and statements in place, one can attempt to verify that they always hold using one of the automated reasoning plug-ins for Frama-C. Additional annotations may be required to prove this. In this particular example, it is sufficient for the WP plug-in to prove that they always hold by defining a loop invariant stating that the iteration counter inevitably stays below the initial value of k. Proving the given bounds on iterations gives a greater amount of confidence in the final verdict of the energy analysis.

Like with the variable relations, a custom predicate for easy differentiation is defined. It is possible to note the presence of this predicate, and then insert the appropriate assertions into the code for the loop, as an additional pre-processing step to make validation easier. However, the important part is that the bounds are read properly for use by the analysis. For example, defining the bounds for the code above looks as follows:

```
int k = 10;
//@ loop invariant ecaloopbound(k, k);
while (k > 0) {
  --k;
}
```

In this case, the predicate is located in the loop invariant, rather than an assertion. The reason for this is that loop invariants are guaranteed to be tied to their respective loop, whereas assertions are not attached to any particular statement and instead viewed as being in a certain location in the code. The difference means that the ecaloopbound predicate, whatever its definition, is checked for every iteration of the loop, rather than just once. Just like the variable relation predicate, defining the loop bound predicate can be done by the user; the Frama-C plug-in for EcaLogic-C only looks at the name and arguments.

6.2.3 Function Signatures

The function signatures described in Chapter 5 can also be considered for inclusion into annotation possibilities. Annotating a full energy consumption signature to a function opens the possibility to reuse energy analysis results in-between executions of the analysis. For instance, it becomes possible for a library to be annotated entirely, so that any programs using this library do not have to apply any specific measures to ensure it is analysed properly; the analysis can simply use the already present signatures.

Adding the functionality to both read and write function signatures into annotations would require, at the very least, a suitable and recognizable format in ACSL. Furthermore, it will operate under the assumption that hardware components referred to in already present function signatures are either exactly the same as those used by the program, or delivered separately. After all, EcaLogic-C can make no distinction between two components with the same name, so if existing function signatures apply hardware component models unknown or different from those used by the current analysis, conflicts may arise.

In ACSL the concept of function contracts is thoroughly defined. The logical location for energy consumption information on the function would then be in the contract. Since it is possible in ACSL to define predicates and logical functions that always apply to the implemented function, placing the energy consumption information there would be the most straight-forward method. Defining the format, then, becomes the main issue: a notation would need to be defined to contain information on both the energy consumption of the function, and any operations it performs on the states of hardware components. Once these are defined, the plug-in developed for Frama-C can easily be extended to search for and parse these custom predicates, and deliver the information to EcaLogic-C.

6.2.4 Recursion

One of the options described in Chapter 5 on how to add the ability to analyse recursion to the energy analysis deals with a fixpoint in a manner similar to how loops are analysed; in particular, Section 5.3. However, since loops require annotations giving a lower and upper bound on their number of iterations, a similar annotation would be necessary to apply this method to recursion. For recursion, it would need to provide a lower and upper bound to the number of executions for the function that would occur if it is called once. This bound would then only apply to the initial call, and not recursive calls.

The loop annotation is scanned for in the loop invariant as used by ACSL, but for functions, it would need to be present in the function contract. Since a contract in ACSL has a section that always applies to the function, an annotation similar to *ecaloopbound* present in this part of the contract would be

the most convenient. The plug-in for Frama-C would then be able to recognize the annotation and pass it along together with the information on the function itself.

6.2.5 Utility

The annotations described in the above sections are all, in some way, essential to the energy analysis. However, there are some elements of C that are difficult to formalize in ACSL, yet may be useful for the analysis, especially for the user. The following predicates and logical functions may, as usual, be more elaborately defined by the user, but their basic usage and meaning as interpreted by EcaLogic-C are as follows:

- $ecalength(< array >)$ - This function may be used to refer to the length of an array in an annotation. During the analysis, it is substituted with the length of the array, as it was defined. If the array was defined with a variable length, this will likely be a variable generated by the GCC compiler. $< array >$ may refer to any array, which includes sub-arrays of multi-dimensional arrays.
- $ecapow(< base >, < exponent >)$ - Exponentiation may be achieved by applying this function. This is most useful when dealing with bounds that are exponents, such as a bound x^2, but may be used for any integer exponent, including exponential ones like 2^x. Unless x is constant in these examples, the polynomial result of the energy analysis will likely be in the order of the exponent.
- $ecacston(< cstring >)$ - In many C programs, arguments to the executable are used in some form or another. To facilitate using these arguments for the energy analysis, this logical function represents a conversion from c-string to an integer value, and is transformed into $numeric(< cstring >)$ in the analysis results. It is assumed $< cstring >$ is a valid target for conversion.

6.3 Hardware Component Models

In the energy analysis, Hardware Component Models (or HCMs for short) are models describing the actual power-consuming hardware elements. They consist of a state, a function *phi*, and a set of component functions. A state contains zero or more variables, each with a defined range. The *phi* function describes the constant power draw associated with a state, which has to always increase along with the state: higher states have to correspond to a larger power draw while in that state. This is needed for the analysis to determine proper upper and lower bounds in a loop.

The component functions in a HCM represent actions that may be performed with the associated hardware. Each has an associated power and time consumption, which have to be constant values, because argument-dependent values might invalidate the restriction that a higher state also means more power consumption. State changes may occur in the body of the function.

The HCMs in the original EcaLogic tool were defined in their own specific language. In order to use them in C programs, and make future work easier, they can be transformed into a more suitable format while retaining the properties needed for analysis. Three options to accomplish integration into the new EcaLogic-C tool are considered in this section: translation to executable C-code, using ACSL function contracts to describe effects, and using an interface compatible with the implementation language directly.

Ultimately, the third option, modelling them in the implementation language, was the most convenient. It offers usage of all elements of the existing language, without having to parse and interpret a specific set of elements just for the components. Furthermore, Scala and Java are object-oriented languages and offer generalization functionality that can be used to create a suitable interface for the components, which would make producing them at increasing complexity substantially easier. Lastly, since they can be compiled beforehand, performance would be increased at subsequent usages of the same components.

In addition to the new Java-based hardware component models, the original options from EcaLogic remain: in EcaLogic-C, it is still possible to define models in the custom language, as well as Scala itself. Java was introduced by virtue of being more well-known, and easier to manipulate in executable form.

6.3.1 Concrete State

This option most closely resembles method used by the original EcaLogic tool. The state consists of some actual variables in the code, which can be modified in component functions. Since these functions have the same expressive power as regular programs, conditional branches and other control structures are allowed. Ergo, to determine the state after execution of a component function, EcaLogic simply evaluates the body of the function.

A direct translation of this state modelling method to C would consist of a number of global variables (possibly encapsulated in a struct), which could then be modified by functions. However, there are several problems with this approach:

- It requires significant additions to source code to deal with the new state, especially when converting existing code to a hardware component model.
- Enforcing restrictions requires global data invariants, and is a lot harder to verify.

- Functions would need to be evaluated to pinpoint any state changes. Evaluating C is significantly more complicated than with the basic, original EcaLogic language, and invalidates the principle of static rather than dynamic analysis.
- Verification of state changes using annotations requires more elaborate assertions throughout the function, if execution is not feasible.

Ideally, hardware component modelling should require as little change to existing code as possible, and should be able to be easily subjected to verification. As such, this option is not preferred, though it is possible with significant effort.

6.3.2 Function Contracts

This method of transforming the Hardware Component Models to the C/Frama-C structures focusses on function contracts as specified in ACSL, without adding any actual C code. ACSL function contracts are series of annotations associated with a function that specify how the function should act, what it modifies, possible return values, termination constraints, etc.

The problems with the first option of modelling the hardware in C were mostly related to having to deal with state changes in a function, and the subsequent need for evaluation. The energy analysis needs to know the state of a hardware component after evaluating a function, no matter the contents. In particular, if the function contains a branching structure where each branch results in a different state, the result still needs to be determined.

ACSL function contracts allow the user to specify behaviours of a function: provided some assumptions hold, the function will act in a certain way. This can be used to specify branching paths of state changes in hardware components based solely on annotations, thus requiring no changes to the actual code, and removing the need to execute the entire function. For example, take a function *toggle-power* of some hardware component, which switches the active state on or off depending on the current state. Specifying this behaviour in a contract might look as follows:

```
/*@ behavior switch_on:
  @  assumes state == off;
  @  ensures state == on;
  @ behavior switch_off:
  @  assumes state == on;
  @  ensures state == off;
  @ complete behaviors;
  @ disjoint behaviors;
  @*/
void toggle-power() {
  //etc
```

In this example, the resulting state is solely based on the current state, but this need not be the case. It may depend on any other ACSL-supported features available, such as parameters, global variables, etc, which are not entirely supported by the energy analysis itself. Furthermore, while the two behaviours are mutually exclusive and complete in the above case, this is not mandatory, however desirable for state-change related cases it may be. If complete determinism is required, it may be enforced in ACSL by specifying *complete* and *disjoint* behaviours, mandating total coverage and mutual exclusion respectively.

Considering state changes of hardware components in this way does not fully eliminate the need for evaluation: Frama-C can offer no guarantees as to the function behaviour selected for a given function call, so it is not trivial to determine what case applies. However, moving the conditions into the world of annotations makes verification significantly easier, and removes the need for entire-function evaluation to ascertain the next state, instead replacing it with the need to evaluate the function arguments and reason about them.

The other elements of hardware component models, namely the *phi* function and energy/time consumption by functions, pose less issues when modelling them via annotations. The *phi* function can be modelled as an ACSL logic function, and the constant energy/time consumption of a function may be specified in a global ensures clause not associated with a behaviour. However, due to the remaining need for potentially difficult, partial evaluation, this option is still not preferred when compared to the next option.

6.3.3 Implementation Language

EcaLogic, and by extension EcaLogic-C, are implemented in Scala, a Java-based language with functional as well as imperative elements. Scala is compiled to Java-based bytecode, so that their executable forms may be interchanged. The hardware component models have a structure similar to a class in many object-oriented languages: an encapsulated set of local variables, functions and inherited elements. As such, it would be possible to define a class interface or similar in Java, to which all components naturally have to adhere, after which the hardware can be modelled.

One main advantage of this method over the others is that, since the languages match, parsing and evaluating each component separately is no longer necessary. Originally, EcaLogic interpreted its input language to infer state changes from component function calls. By compiling the components into the same language as the analysis, they can instead be called through their interface instead, which is faster than evaluation. Since compiling only has to occur once, and executing byte-code through a virtual machine is generally faster than dynamic interpretation, this would improve efficiency substantially. Furthermore, the language shift would make further changes to the inner workings of hardware components and the analysis thereof easier to implement, and offers users more

options: Java is generally more widely known than Scala, and especially more widely known than EcaLogic's own language.

A disadvantage of shifting the language is that the languages for the hardware components and the actual program to be analysed no longer match. Java is common enough that this may not be a problem for the end user, but since the C programs that are analysed still have to call the component functions, a calling convention needed to be established to accomplish Java-based components for EcaLogic-C. While it would be possible to define an ACSL construct to handle this in a non-interfering method, it would also be possible to associate actual C functions with hardware component models. This way, energy consumption can be added to existing functions as in the other two options, without having to add to existing code. A mixture of these constructions would offer the greatest flexibility to the end user.

Having the Hardware Components modelled as a Java class for EcaLogic-C can make writing them easier, whilst retaining the above advantages. By using Java Reflections, most elements required for the hardware model can be inferred automatically, which makes the resulting code requirement not very different from the old language used for hardware components. As an example, the simple on-off radio can be defined in Java as follows:

```
public class JavaRadio {
    public int active;

    public JavaRadio() {
        active = 0;
    }

    @Consumes(energy = 200, time = 20)
    public void on() {
        active = 1;
    }

    @Consumes(energy = 10, time = 10)
    public void off() {
        active = 0;
    }

    public int phi() {
        return 20 + active * 200;
    }
}
```

Using Reflections, each required hardware component element can be inferred: the state contains the declared fields, methods can be invoked during run-time as compiled byte-code, and the phi function can be detected and executed easily on a specific instance of the object, without having to separately maintain the state. This leaves the implementation of a HCM as close to a regular program as possible whilst also decreasing the execution time of EcaLogic-C. As such, this option was the most preferred one, and is implemented exactly as described

in this section.

The possibility to attach a component function to a real C function was implemented as a so-called alias: if the C function is called, it is substituted for the hardware component functions that have it as their alias, and any actual implementation of the C function is ignored. As a result, an existing C library dealing with some aspect of the hardware can be modelled, making it act effectively as a hardware component with whatever level of abstraction is desired by the user. More than one component function can be aliased as one C-function; the calls are simply appended.

Adding an alias to a hardware component function looks as follows:

```
@ConsumesAlias(energy = 200, time = 20, alias = "socket")
public void CreateSocket() {
  sockets += 1;
}
```

This example creates an alias for the function *socket*, which creates a new hardware socket for listening to internet connections. It abstracts the call to consume a certain amount of energy and time, and increases the *sockets* state-variable by one, which in turn is an abstraction of the number of open connections. By adding more aliased functions, it becomes possible to model network interaction to whatever abstraction level is desired.

6.4 Control Flow

Some elements of C that are lacking direct translations in the original EcaLogic involve the control flow of a program. This includes for-loops, break and continue statements, case-switch sections, and go-to instructions. Since these may affect the flow of the analysis itself in a significant way, they have to be analysed properly for EcaLogic-C and the energy analysis. This section lists the various control structures and how they can be dealt with.

6.4.1 For-loops

A for-loop consists of three parts in the conditional: the initialisation, update and check steps. Frama-C uses this information to transform for-loops into a while loop by first executing the initialisation step. The check is executed at each iteration, and the update occurs at the end of it. Since while loops are supported by the energy analysis, this makes sure for-loops are treated properly automatically, though loop bounds still have to be provided.

With for-loops, the possibility exists to infer some information about the bounds of the loop automatically. If the conditional for the loop is of some specific, easily recognized format with a solid loop bound, this may be used to simplify

the work required by the user of the tool somewhat. For instance, in a for-loop with a simple counter from 0 to some value x, a higher bound for this loop could be automatically inferred to be x, leaving only the lower bound to be decided.

6.4.2 Break & Continue

Break and continue statements offer some control to the operation of a loop beyond the given conditional. A break interrupts the loop, whereas a continue causes it to immediately proceed to the next iteration. It may be safe to assume that these are only used as conditional statements: if one is always executed, any code after the statement in the loop will never be reached, also known as 'dead code'.

For the upper-bound analysis, the assumption that they are always conditional means the analysis should follow the control flow to the branch where the break or continue was not executed, so that any energy-consuming elements beyond the statement count towards the upper bound. In terms of the lower-bound analysis, operating under the reverse modus means the break or continue will always be executed: in this case, the specified lower bound for the loop may be relied upon. Put differently: a $while(true)$ loop will (usually) have some internal break statements, and it is expected that the lower and upper bounds specified for this loop account for this, rather than providing infinite bounds. As such, break and continue statements are not treated in a special manner.

6.4.3 Go-to Statements

A go-to statement represents a direct jump to a different location in the code, usually denoted with a label. Usage of go-to statements has been widely discussed, perhaps most famously by Dijkstra[22], where the conclusions are far from universally agreed upon. Stigma notwithstanding, it is a form of control flow in programming languages, including C, and as such must be either analysed or disallowed. One disadvantage of the go-to statement is that it has no consideration of scope beyond the current function: it is possible to jump outside of a loop or other structure. Break- and continue-statements can be seen as special use cases of a go-to statement, though they are more restricted, as described in the previous section.

The problem go-to statements present to the energy analysis is that it can easily create loops in a program, despite not explicitly being listed as such. Consider the following example:

```
entry:
  if (x > 5) return;
  else goto entry;
```

In this case, the program will loop until x becomes greater than five. As such, it can be rewritten to:

```
while (x <= 5) {}
return;
```

While rewriting a go-to statement was easy in this case, the same cannot be assumed to hold for all others. Due to the lack of defined bounds or usage restrictions on go-to statements, they can make the energy analysis significantly more difficult, both for the analysis itself and the user of the analysis tool. In order to prevent these issues, then, it is assumed that go-to statements are rewritten to eliminate them. Some methods have been researched to accomplish such a rewrite, for instance, by altering the control flow of the program directly[23].

6.5 Adding Data Types

C includes various data elements of differing complexity. These specific elements concern the C11 standard[8], and consist of the following, overlapping with those detailed in Chapter 3:

Element	Details
(Un)signed integer types	Unlike the natural numbers, C has integer values of differing size, ie. char, short, int, etc. Problems may be caused by overflow: if the number progresses beyond its defined limits, it loops back around to its minimal value, and vice versa. However, since the automated reasoning tools of Frama-C offer the possibility to automatically check for overflow errors, leaving this as a choice for the user is preferable; numbers are therefore abstracted to a larger capacity.
Floating point and double values	These values deal with non-whole numbers, and are usually heavily affected by specific hardware and library implementations. Furthermore, since Frama-C already abstracts them to general real numbers, EcaLogic-C can use them as such. Though, since actual real numbers have infinite precision, some concession has to be made: double values are used to store values in the actual implementation.
Arrays	Arrays in C are defined as areas in memory containing a set number of elements. In general, this region cannot be extended without reallocation; so the length of the array is known when it is declared, possibly parametrized. As such, they are treated as a fixed-size collection of individual variables for the purpose of analysis.

[8] C11 is described as an ISO standard: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`

C-strings	Character-strings in C refer implicitly to arrays of the basic number-type of characters. Frama-C makes a special case of constant string literals to store them as-is, with the possibility to check their length at the time of analysis automatically. The storage type is still treated as a pointer to an array of characters, so any operations using this form are still applicable: the special storage case can be translated back without issues. It should be said that while Frama-C and ACSL do recognize character strings, the automated reasoning plug-ins do not yet support reasoning about them, which might make automatic verification more difficult if they are not treated as a simple array.
Enumerations	These consist of a set of numbered identifiers denoting constant values. While these may be considered as an integer type, the possibility to assign specific values to enumeration elements might interfere with such an assumed order. As such, they are modelled as a sequence of constants; their separation from general constants is limited to plain assignment, so this does not affect the analysis.
Structures	Defined as a set of identifiers of some type, these offer grouped storage. In most cases these would not offer much more complexity than ordinary values, but there is the possibility of recursion, including pointer usage. Therefore, in the base case without allowing recursion, they are treated as a fixed collection of variables. The case where recursion is used is discussed in more detail in the respective section of this chapter.
Unions	Similar to structures, these elements contain various typed identifiers. The unique part, however, is that each identifier modifies the same location in memory. As such, data corruption of each identifier is possible. Ultimately, in order to prevent problems arising from this element, attempting to access a union element not of equal type to the last modified element is considered undefined.
Complex numbers, thread locals, atomic types	These types either concern concurrency in the case of thread locals and atomic types, or may be (similar to floats) heavily dependent on specific implementations. In the cases where they are implemented as base operations they are treated and analysed as such; in other cases they are not used for the analysis.
Type casts	C has both explicit and implicit type casts. The implicit kind is guaranteed to be reliable, so modelling this would not pose a large problem. However, modelling explicit type casts may pose a problem if these intersect with other necessary elements, such as bounding annotations. For the tool, type casts are assumed to be correct and the final value is considered as-is; any correctness verification is left to either the compiler or the user.

With the exception of complex numbers, thread locals and atomic types, all of the data types in the list have been implemented in EcaLogic-C by following the new rules defined in Section 3.4. The extension for floating-point values does not change the rules, but instead modifies their implicit basic values from positive integers to double values. C-strings are treated as arrays where applicable, and enumerations are processed by using them as a set of constants. Lastly, type-casts are assumed to have taken place correctly, and thus are not treated in any special fashion.

Frama-C offers some features in its specification language to aid in formally verifying any of the above elements as they are used:

- For structs, it is possible to define inductive properties about termination and length, which aids in providing a suitable (and, more importantly, checkable) bound on their usage.
- The possibility exists to define 'type invariants' for structures and other defined types, which have to hold at all points in the program. This may be used to further verify specific usage cases.
- Using either of the previous options (or a combination) it is possible to define logical functions in ACSL for bounding a struct, which can be checked in Frama-C and be used as an alternative by the analysis.
- Arrays are already implicitly considered with their size, so reasoning about them does not require many additional elements.
- Floating point numbers and other non-whole numbers are generalized to the Real domain automatically by Frama-C, so further abstraction would not be required.
- When modelling unions, Frama-C takes into account the order in which the specific elements are changed, treating different orders as distinct resultant states. This provides some additional security, though it may also complicate the verification.

6.5.1 Variable-Length Arrays

For the GCC compiler, used by Frama-C and by extension EcaLogic-C, variable-sized arrays (referring to arrays initialized with variables, not arrays with variable length) are handled via dynamic allocation: upon declaration of a variable-length array, a new temporary local variable is created keeping track of the size of the upcoming array declaration. At the statement containing the declaration, a call to the built-in alloca function is placed, which allocates a section of memory that is automatically freed at the end of the scope. The actual size allocated is a multiplication of the size variable with a call to the sizeof of the actual contents of the declaration. As an example:

```
int x = 24;
int list[x];
```

Is transformed to the following:

```
int x = 24;
size_t __lengthoflist;
int* list;
__lengthoflist = x;
list = __builtin_alloca(__lengthoflist * sizeof(x));
```

Treating this as a special case, the Frama-C plug-in for EcaLogic-C looks for applications of the built-in alloca function, and uses its arguments to define the size of the array at that point. This is mainly to allow these arrays to be used for bounding operations as-is, without the user having to define anything in addition; the effect of this special case on the actual energy analysis is non-existent.

6.5.2 Additions to the AST

In order to implement the energy analysis for extended data structures, additions had to be made to the internal representation of the program abstract syntax tree. Most importantly, this meant moving all values to the domain of doubles instead of integers, with the added bonus of allowing for non-integer values when determining the energy consumption and giving the result.

To facilitate some form of validation, as well as extend the available information, basic type information was also added to the AST. This is used to store the types used by arrays, structures and other datatypes, but also other variables to be able to discern same-named elements. Types defined by the user are unrolled when encountered.

Finally, to implement the new analysis rules, new supported operations directly corresponding to the rules were added. This includes expressions for access to arrays and other data types, but also declarations and modification statements. While these statements are usually similar to their counterparts for other data types, the distinction between them is kept explicit to avoid overly generalizing the operations.

6.6 Allowing Pointer Usage

Direct pointer manipulation is a staple of C, so safe operations (as detailed in Chapter 4) have been implemented into the supported operations of EcaLogic-C. Treating safe pointer operations as not much different from their respective operations in Section 6.5 means the extent of the additions to the implementation is rather small. Instead, most implicit memory operations are handled

by the Frama-C plugin: where possible, implicit (and a small amount of explicit) pointer operations are directly converted to their already-implemented counterparts.

The remaining pointer operations that were not covered by the Frama-C plugin are those that concern explicit pointer arithmetic, as well as explicit dereferencing or referencing of pointers or variables. In order to reason about these operations, they were added as binary or unary operators to the internal AST of EcaLogic-C, but other than this case they are not treated as a special operation: an assignment to a dereferenced pointer is treated the same as an assignment to a regular variable, for instance. This allowed for pointer operations with only minor additions to the implementation of EcaLogic-C being necessary.

6.7 Function Signatures & Recursion

Capturing the precise energy, time and component changes induced by a function can be used to determine that function's signature. An extension of such a signature to allow for recursive function calls and their analysis becomes possible as well, and both of these elements are discussed in Chapter 5. Adding these elements to EcaLogic-C would both extend the possible language elements that could be analysed, as well as increase the efficiency of the tool by making sure that functions would only need to be analysed once.

Unfortunately, the inclusion of function signatures (and by extension, recursive function calls) required significant changes to the inner structures and reasoning methods of the tool. Coupled with time constraints, implementation was not feasible. This does not imply it is impossible; on the contrary, with the required changes, implementing the function signatures would be very feasible, though time-consuming. A basic implementation, which does not look at state changes in hardware component models, was added to illustrate the purpose of the signatures.

7 Energy Analysis Examples

To test the extended analysis and its implementation in the EcaLogic-C tool, some examples have been worked out. For each of the following problems, two algorithms are compared with each other: sorting an array of values, and clock synchronization in a network. Array sorting is a common example for algorithmic studies, with only the implicit CPU component consuming any energy, whereas the network synchronization example deals with networking hardware, so the focus thereof lies on the hardware utilization.

Specifically, the array-sorting example serves to demonstrate how the extended analysis would operate on elements not supported by the original analysis, whereas the focus of the clock synchronization lies on demonstrating the applicability of the energy analysis and EcaLogic-C to a practical program, and modelling the hardware associated with its function.

Another verification option for the analysis is to compare the static energy analysis results with dynamic analysis case studies performed on real hardware. For example, the case study performed by Chang et al. on the ARM7TDMI processor[10] provides energy consumption information for individual instructions executed on the processor. A comparison to the static energy analysis can be made by looking at the byte-code of the compiled program, aggregating the energy consumption for each instruction based on the dynamic analysis results, and comparing this to a concrete result.

For embedded systems, Šimunić et al. performed a hardware and software analysis on the SmartBadge, a small personal system[24]. They proposed an energy consumption model that matched the actual hardware consumption, based on individual instructions. Later, the research was expanded to include an implementation of an encoding algorithm running on the system[25]. A comparison to the static analysis and EcaLogic-C could be made, and would consist of a suitably elaborate hardware model of the SmartBadge. Having this, the implementations tested in the research could be analysed and compared.

Unfortunately, due to time constraints, comparison to a dynamic analysis case study or a different static analysis could not be performed.

7.1 Array Sorting

Array sorting is a common target for algorithmic studies, both due to its practical usability and the many subtly different solutions that exist. Many of the sorting algorithms incorporate recursion, which is the focus of this example. Since recursion was ultimately not added to the EcaLogic-C implementation due to time constraints, this example remains theoretical.

For this particular example, QuickSort and MergeSort are considered, both recursive algorithms. QuickSort sorts a list by first selecting a pivot, then re-

arranging the list so that elements smaller than the pivot occur before it in the list, and larger elements are moved to beyond the pivot. This step is then recursively repeated for the smaller and larger lists. Runtime complexity of QuickSort is, on average, $O(n*log(n))$, but the worst-case is $O(n^2)$[26]. MergeSort starts by considering each element as an ordered list of length 1, then starts recursively merging adjacent lists. In contrast to QuickSort, MergeSort has a runtime complexity of at worst $O(n*log(n))$[27], but has a larger constant overhead.

Comparing QuickSort and MergeSort is interesting because of their theoretical bounds. QuickSort performs worse in a worst-case scenario than MergeSort, but QuickSort is considered more efficient in practice; it has been included in the C standard library and used by Java for a number of years[28]. Example implementations for both QuickSort and MergeSort in C are given in Appendix B.

Due to the practical preference leaning towards QuickSort, it may be expected that MergeSort has a larger energy consumption caused by basic operations, unrelated to the loops. This is collaborated by the loops in the MergeSort implementation only passing over most of the list once: elements are merged once, then considered sorted for that iteration, though they can be compared several times in one merge operation. QuickSort, on the other hand, can rearrange the same element more than once, but only performs a single comparison each iteration.

Both functions enter recursion on a part of the original list twice to perform the sorting. As a result, their recurrence relations featuring recursion would be similar, if not equal, resulting in comparable results. The difference between the algorithms would then lie strictly in the loops, the constant energy usage from operations, or both. Since loops are over-estimated into their worst-case, it is conceivable that the energy analysis and EcaLogic-C would consider QuickSort more consuming as parametrized in the function arguments, though less consuming in the constant part of the polynomial.

7.2 Clock Synchronization

The clock synchronization problem deals with individual system clocks in a network drifting apart. Two existing algorithms have been tested: the Berkeley algorithm[29] assigns one network machine to be the master, which takes timestamps from each other machine, averages them, and sends a correction back out. The second algorithm is Cristian's algorithm[30], which works via a central time server that is assumed to have the correct time, and responds to each request with the direct correction required for the client to match its own time.

For this example, only the master or server of each algorithm is considered. Cristian's algorithm is simpler than Berkeley's, but it does require a central server that has to be correct, and is less accurate when dealing with higher-latency

or more widely distributed systems. As such, it can be expected that Cristian's algorithm will consume less power, especially since Berkeley's algorithm has to keep multiple connections open. Both examples were written with the same libraries, following the same conventions, and using the same hardware components. Since both of them deal with network connections, a hardware component concerning Sockets was created, and a more basic model to deal with standard I/O and other calls was added to cover the functions not related to networking. The Sockets hardware component has a single state variable, the number of sockets open, to represent additional energy cost from keeping track of more than one connection. In the end, the only code analysed is that of the programs itself, and all the library functions used are covered by the hardware models.

The code of each example, as well as the network hardware model, may be found in Appendix C. It should be noted that the code is functional: that is to say, it will compile properly, can be linked with the appropriate libraries, and can then be executed on a computer to perform its function. This helps to illustrate the applicability of the analysis to 'real', practical programs, instead of just a While-like language.

Analysis Results Berkeley's algorithm performs a total of three loops: accepting connections, computing the average timestamp, and transmitting the results. The averaging-loop does not use any Sockets-based hardware functions, so it will only affect energy consumption by the CPU component. The first loop, accepting connections, opens N active sockets, where N is taken from the command line. These sockets remain open until the last loop, where the result of the timestamp computation is transmitted, before the socket is closed. Because the energy consumption of the Sockets hardware component depends on the number of active sockets open (in this case, N at maximum), and this holds through at least one iteration of the first and last loops, it will be over-estimated into a large energy consumption for the Sockets component, as if all N sockets were open during most of the program. The result given by EcaLogic-C on the energy consumption of the Sockets component when used by Berkeley's algorithm is the following (*numeric* is used to denote char-string to integer value conversion):

$$17805.0 + 4825.0 * numeric(argv[1])$$

Cristian's algorithm only performs one loop to receive, compute and send the updated time, but performs most actions the same as Berkeley's algorithm otherwise. However, this does mean that each socket is considered to be open longer for each iteration in the analysis, which is counted heavily towards the passive, time-dependent energy usage by the Sockets component. As such, it may be expected that the analysis returns a larger consumption in the order of N, though perhaps with lesser constant usage. The result given by EcaLogic-C for the Sockets component when used by Cristian's algorithm is the following:

$15905.0 + 6725.0 * numeric(argv[1])$

In practice, the difference between both results would not be significant enough to favour one algorithm over the other, so the analysis can aid in eliminating the concern regarding unequal energy consumption if a choice was considered.

8 Limitations & Opportunities

The extended energy consumption analysis and its implementation, EcaLogic-C, alleviate several of the limitations present in the original work, but there are still avenues of improvement feasible. This chapter discusses some limitations that were not solved, or not solved entirely, in the extended versions, as well as new problems. Furthermore, several suggestions for future work are presented, angled towards either improving the usability of the analysis and tool, or extending their abilities to further improve their applicability to practical programming languages.

8.1 Unsupported Language Elements

Certain elements of programming languages remain out of the scope of the energy analysis, and by extension EcaLogic-C for C-specific structures. Explicit code jumps are as of yet unsupported: analysing a go-to statement, especially if it is conditional, can not be solved by simply continuing the analysis at the new location, because the analysis explicitly considers all branching paths. Avoiding an infinite loop in such a case would then require more annotations or special rules to determine proper bounds, so it is preferable to disallow explicit code jumps and instead rely on the user to partially rewrite the program. However, since it does not technically invalidate the analysis itself, adding support is a possibility for future work.

Another, increasingly common, element of programming languages that is wholly unsupported is the notion of concurrency. While abstraction of this concept might be a possibility, the analysis itself will still always consider only one thread of execution. Concurrency is considered out of the scope of this thesis, as it was in the original paper, and as such the possibility of inclusion cannot be dismissed. A solution to analysing multiple threads of execution might be conceived, with appropriate abstraction, though such an extension would feature extensive alterations to the operation of the analysis as a whole.

Other unsupported elements are objects, higher-order functions, and language features such as generics, reflection, direct file interaction, anonymous functions, etc. Recursion, while discussed in Chapter 5, was not implemented due to time constraints. As with the other unsupported elements, the listed ones are either considered outside of the scope of the thesis, or require significant alterations to the entire analysis. That is not to say these elements may never be introduced, but only that it requires a non-trivial amount of additional research. Possibilities exist afterwards, then, to extend the analysis to languages using any of the previously unsupported elements.

8.2 Semantic Analysis

The original energy analysis implementation, EcaLogic, also performed a semantic analysis on the program to be analysed first, in order to detect any errors or unsupported operations. Due to the language shift towards C in EcaLogic-C, this part of the original implementation became incompatible with the extended internal data structures. Instead, Frama-C calls a C compiler before any transformations, and this is taken as a basic check on the semantics: if the program compiles, it is at least properly coded. However, this check does not extend towards the unsupported operations, and does not verify that every requirement of the analysis is met. As a result, errors in the program or analysis will be harder to notice and solve, which decreases the usability of the tool somewhat.

Specifically, EcaLogic performed checks on the usage of variables and parameters in loop bounds, recursive function calls, and function calling conventions. Of those, only the function calls are checked by the compiler in EcaLogic-C, leaving the other checks up to the user. Additional checks that can be performed are the presence of mutual recursion, go-to statements, and unsafe pointer usage in the case of manipulation of a pointer to a non-structured variable. The addition of these checks would make it easier for the user to prevent implicit or explicit errors during the analysis.

8.3 Accuracy of the Analysis

Due to the problem of programs being ultimately undecidable, the energy analysis always produces a lower and an upper bound on the energy consumption of a program. Accuracy then depends on the tightness of these bounds: the closer the bounds are to the real energy consumption, the better. Ideally, the lower and upper bounds are exactly equal; though in the case of programs more complex than a single control flow route, this may only be the case if every path is exactly equal, and as such may not be relied on. The main cause of looser bounds in the energy analysis implemented by EcaLogic-C is the overestimation used in determining the energy consumption of loops.

The potential solution to solving analysis of recursive functions, by determining the difference between component states after calls of the function, may be used to improve the analysis of loops as well. By considering the difference between each iteration in a loop, rather than a function call, an iteration signature might be conceived. Maximizing such a signature, or even using it directly, might then be used to tighten the bounds on a loop considerably.

8.4 Soundness Proof of the Analysis

The original energy analysis[1] was proven to be sound with respect to the semantics of the simple While-like language[18], showing that the resulting bounds

are guaranteed to be overestimations of the real power usage. The proof is based on the individual rules and how they would affect the energy bounds, as well as the restrictions on the energy consumption and state ordering of the hardware component models.

To offer a greater measure of certainty in the accuracy of the extended analysis (and its implementation in EcaLogic-C), the soundness proof can be continued to cover the added features. The restrictions on the hardware component models are still present in the extended analysis, so this section of the proof could remain largely the same. The changed rules for function signatures and added rules for the various data structures and pointers, however, would need to be newly considered for every step in the proof that depends on the application of rules, which is a large amount. In the case of the data-structure and recursion rules, they are similar enough to assignment- and expression-based rules that this should not pose a significant chance in the proof tactic. The function signatures add more operations to reason about relations between hardware component states, and thus would require more significant additions to the proof.

8.5 Usability Improvements

Usability was one of the main concerns of the original EcaLogic implementation; in particular, there was a significant amount of semantic analysis and suitable error messages for the user to identify problems with. Due to the extensions of EcaLogic-C, not every usability feature survived: in addition to the lack of semantic checking as mentioned earlier in this chapter, by moving the parsing and compilation step to Frama-C, any errors in this step are heavily generalized, and specific compiler error messages are no longer automatically provided.

Due to the strict separation of compilation and analysis, a user would have to compile the program to be analysed separately, before handing it off to the analysis. In these cases, it helps that the annotations required for the analysis are in a non-disruptive format, so that compilation should be able to be performed without additional requirements. However, in cases where the program has to be structurally modified before analysis becomes possible, separate compilation for error detection might become harder to perform. A deeper integration of the language compiler into the analysis tool would then help users in producing correct results.

8.6 Application of ACSL

ACSL, or 'ANSI/ISO C Specification Language', is the specification language used by Frama-C to reason about and prove elements of C programs. EcaLogic-C already uses this language to parse information on bounds and relations in programs meant for energy analysis. It may be convenient to specify some of the requirements and checks for the limitations of the energy analysis in ACSL,

so that the end user of the analysis can apply these to prove that their program satisfies all the requirements.

As an example, the array-based operations in the energy analysis have to be within the bounds of the allocated array, because otherwise the program's behaviour becomes unpredictable. An assertion to prove that array access to a variable-size declared array (variable only on the point of declaration) is always within bounds can be defined as follows:

```
//Declaration of the array to some variable n
int array[n];
//@ ghost int array_length = n;

//Application
//@ assert x >= 0 && x < array_length;
int y = array[x];
```

This example stores the length of *array* in a ghost variable *array_length*, which can later be used in other annotations only. On access of the array, x can be checked against the bounds of the array: indices may vary between 0 and $array\_length - 1$.

Using a similar method, the access restrictions on unions and pointer-based structure manipulation can also be verified. Furthermore, Frama-C offers a command-line option to generate assertions regarding overflow and some floating-point operations automatically, though these may be less vital to the energy analysis.

8.7 Web-Based Interface

The original implementation of the energy analysis, EcaLogic, can be run on a webserver to allow visitors to use the tool without having it or any of the prerequisites. While the same might be true for the Scala-based part of EcaLogic-C, it does not necessarily hold for Frama-C or the Scala and Java compilers. Due to the number of tools involved and their significantly higher overhead, running a webserver that offers the new analysis would be more costly, though still possible. Allowing executable Java or ACSL would, however, pose a security risk, so that plainly running the tools on the server might not be preferable. It would then only be feasible with sufficient precautions taken, for instance, with a virtual sandbox without internet access.

8.8 Other CIL-Based Languages

Frama-C uses the Common Intermediate Language originally developed by Microsoft[9] internally to store processed C files. Since EcaLogic-C operates on this internal representation, it can be possible to use the existing infrastructure to analyse other languages that translate to CIL. Notably, this includes other .NET framework languages such as C++ and C#, but other translations exist: LLVM's VMKit project focuses on transforming LLVM into CIL. Such translations can significantly expand the applicability of the tool.

It needs to be mentioned, however, that Frama-C uses its own library for manipulation of the CIL abstract syntax tree. As such, it may not be fully compatible with CIL files generated from, for example, C# compilation. For one, the CIL Frama-C dialect has no direct capacity for objects, though they are officially supported by CIL in the specification. Furthermore, the ACSL extension in use by Frama-C is not officially supported, and as such might need to be altered before application to other languages becomes possible.

[9] CIL is part of the Common Language Infrastructure, defined here: `http://www.ecma-international.org/publications/standards/Ecma-335.htm`

9 Conclusion

The goal of this thesis was to extend a static energy analysis method[1], implemented in EcaLogic[3], so that it could analyse more practical programming languages. Extended data structures, memory manipulation, and recursion were the language elements considered. These extensions, with the exception of recursion, were implemented in EcaLogic-C: an extended version of the tool, allowing it to analyse a subset of the programming language C by using the suggested analysis additions.

The practical applicability of the energy analysis and EcaLogic-C has increased compared to the original EcaLogic, though a C program may still not be analysed unaltered: loop bounds and variable relations still have to be specified, though this can now be done in an non-obstructive fashion. Furthermore, straight jumps in the code or any form of concurrency remains unsupported for the analysis, and were considered out of the scope of this thesis.

While applicable practically, the energy analysis and EcaLogic-C are open to expansion: in particular, adding support for concurrent program elements would further increase its analysis capabilities. Usability may also be a concern for future work; since some user-interaction elements of the original EcaLogic had to be disabled, the extended version is more opaque.

References

[1] R. Kersten, P. P. Toldin, B. van Gastel, and M. van Eekelen, "A hoare logic for energy consumption analysis," 2014.

[2] N. D. Jones, *Computability and complexity: from a programming perspective*, vol. 21. MIT press, 1997.

[3] M. Schoolderman, J. Neutelings, R. Kersten, and M. van Eekelen, "Ecalogic: hardware-parametric energy-consumption analysis of algorithms," in *Proceedings of the 13th workshop on Foundations of aspect-oriented languages*, pp. 19–22, ACM, 2014.

[4] C. Alippi and C. Galperti, "Energy storage mechanisms in low power embedded systems: Twin batteries and supercapacitors," in *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology, 2009. Wireless VITAE 2009. 1st International Conference on*, pp. 31–35, IEEE, 2009.

[5] C. Alippi, R. Camplani, C. Galperti, and M. Roveri, "A robust, adaptive, solar-powered wsn framework for aquatic environmental monitoring," *Sensors Journal, IEEE*, vol. 11, no. 1, pp. 45–55, 2011.

[6] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *ACM SIGPLAN Notices*, vol. 42, pp. 200–210, ACM, 2007.

[7] F. Poletti, A. Poggiali, D. Bertozzi, L. Benini, P. Marchal, M. Loghi, and M. Poncino, "Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support," *Computers, IEEE Transactions on*, vol. 56, no. 5, pp. 606–621, 2007.

[8] L. M. Feeney and M. Nilsson, "Investigating the energy consumption of a wireless network interface in an ad hoc networking environment," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, pp. 1548–1557, IEEE, 2001.

[9] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pp. 280–293, ACM, 2009.

[10] N. Chang, K. Kim, and H. G. Lee, "Cycle-accurate energy consumption measurement and analysis: Case study of arm7tdmi," in *Low Power Electronics and Design, 2000. ISLPED'00. Proceedings of the 2000 International Symposium on*, pp. 185–190, IEEE, 2000.

[11] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, pp. 345–363, 1936.

[12] M. Barnett, K. R. M. Leino, and W. Schulte, "The spec# programming system: An overview," in *Construction and analysis of safe, secure, and interoperable smart devices*, pp. 49–69, Springer, 2005.

[13] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter, "The spec# programming system: Challenges and directions," in *Verified Software: Theories, Tools, Experiments*, pp. 144–152, Springer, 2008.

[14] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *Software Engineering and Formal Methods*, pp. 233–247, Springer, 2012.

[15] L. Correnson and J. Signoles, "Combining analyses for c program verification," in *Formal Methods for Industrial Critical Systems*, pp. 108–130, Springer, 2012.

[16] S. Falke, D. Kapur, and C. Sinz, "Termination analysis of imperative programs using bitvector arithmetic," in *Verified Software: Theories, Tools, Experiments*, pp. 261–277, Springer, 2012.

[17] S. Falke, D. Kapur, and C. Sinz, "Termination analysis of c programs using compiler intermediate languages," in *RTA*, vol. 11, pp. 41–50, 2011.

[18] P. Parisen Toldin, R. Kersten, B. v. Gastel, and M. v. Eekelen, "Soundness Proof for a Hoare Logic for Energy Consumption Analysis," October 2013.

[19] O. Kaser, C. Ramakrishnan, and S. Pawagi, "On the conversion of indirect to direct recursion," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 2, no. 1-4, pp. 151–164, 1993.

[20] D. H. Greene and D. E. Knuth, *Mathematics for the Analysis of Algorithms*. Springer, 2007.

[21] V. G. Papanicolaou, "On the asymptotic stability of a class of linear difference equations," *Mathematics Magazine*, pp. 34–43, 1996.

[22] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[23] A. M. Erosa and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements," in *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pp. 229–240, IEEE, 1994.

[24] T. Šimunić, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pp. 867–872, ACM, 1999.

[25] T. Simunic, L. Benini, and G. De Micheli, "Energy-efficient design of battery-powered embedded systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 1, pp. 15–28, 2001.

[26] C. A. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[27] J. Katajainen, T. Pasanen, and J. Teuhola, "Practical in-place mergesort," *Nord. J. Comput.*, vol. 3, no. 1, pp. 27–40, 1996.

[28] J. L. Bentley and M. D. McIlroy, "Engineering a sort function," *Software: Practice and Experience*, vol. 23, no. 11, pp. 1249–1265, 1993.

[29] R. Gusella and S. Zatti, "The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3 bsd," *Software Engineering, IEEE Transactions on*, vol. 15, no. 7, pp. 847–853, 1989.

[30] F. Cristian, "Probabilistic clock synchronization," *Distributed computing*, vol. 3, no. 3, pp. 146–158, 1989.

[31] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C programming language*, vol. 2. prentice-Hall Englewood Cliffs, 1988.

Appendices

A Semantics Reference

This appendix gives the full set of semantic rules for the energy consumption analysis, both for the operational and energy-aware semantics, as well as the energy analysis rules. The general structure, special characters, and other elements are explained in greater detail in Section 2.1.

A.1 Operational Semantics

These are the operational semantic rules: the rules governing what effect each statement has, without considering energy consumption. To avoid unnecessarily cluttering the energy analysis rules, type-system restrictions are confined to this part of the semantics.

The semantics for statements are in Figure 16, whereas the semantics for expressions can be found in Figure 17.

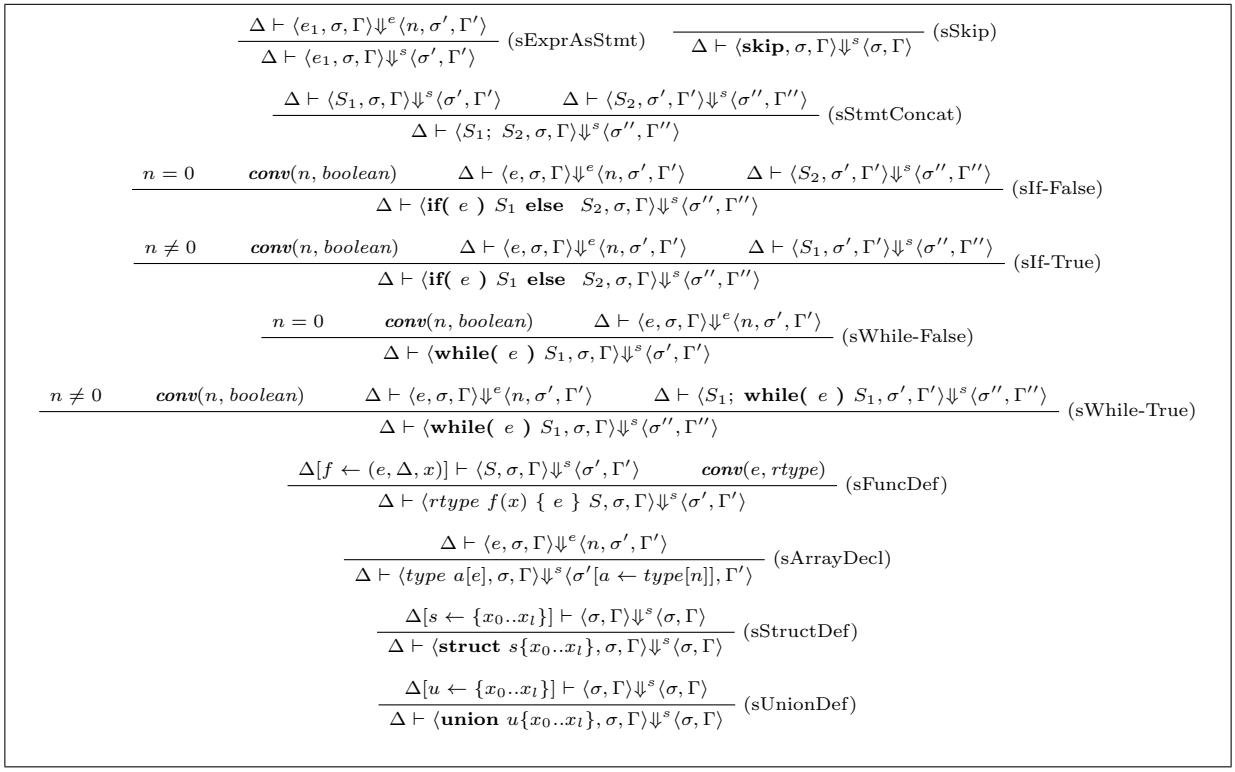

Figure 16: Statement semantics.

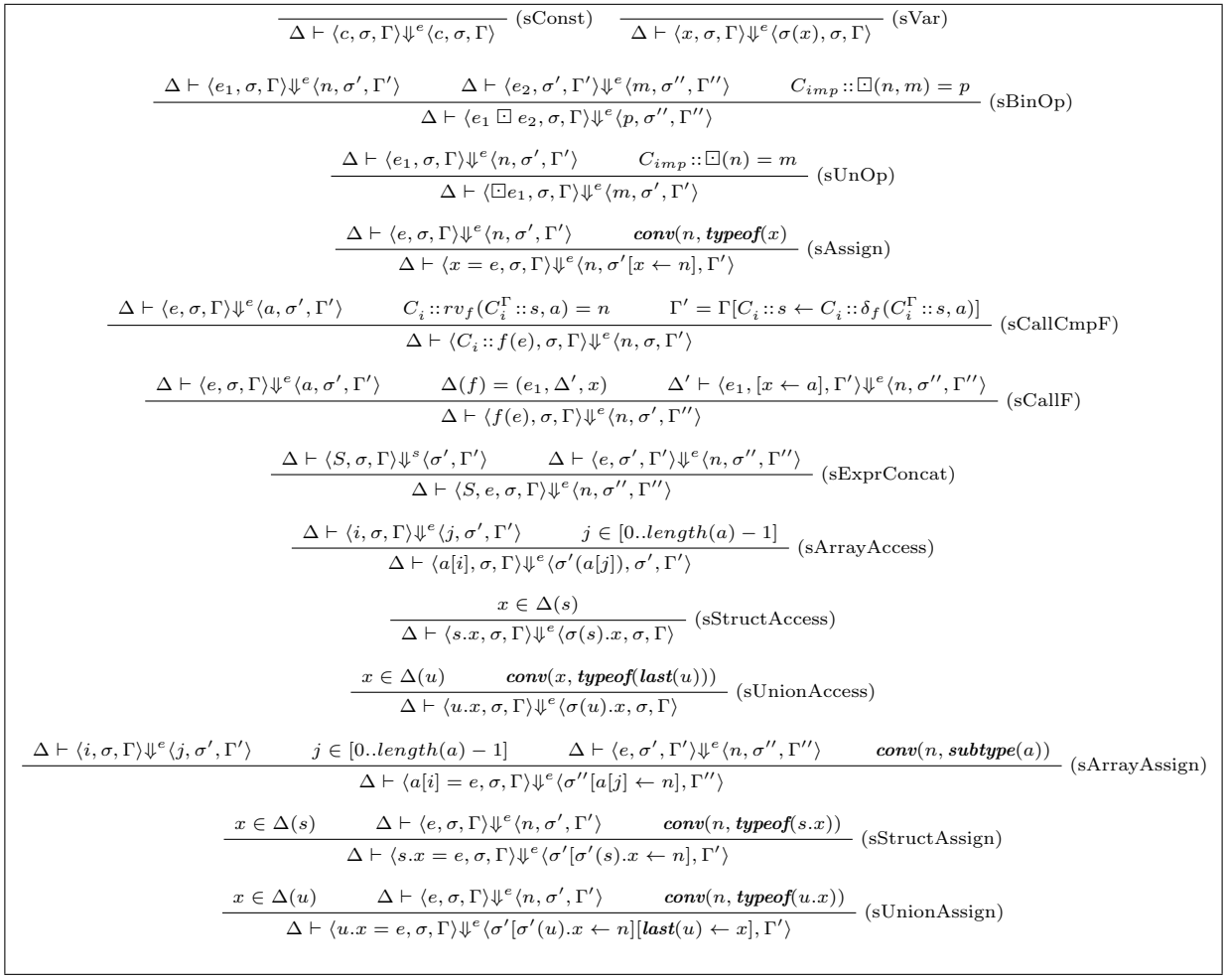

Figure 17: Expression semantics

A.2 Energy-Aware Semantics

The full energy-aware semantics are as in Figure 18 for statements, and Figure 19 for expressions. To avoid cluttering the image further, elements from the rules for the regular semantics are omitted: this includes the typing restrictions and union safety restriction. It may be assumed that both these rules and the equivalent regular semantic rules need apply.

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

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

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

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

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

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

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

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

$$\frac{\Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{mdec}]}{\Delta \vdash \langle type\ a[e], \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma'[a \leftarrow type[n]], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{mdec}\rangle} \text{ (eArrayDecl)}$$

$$\frac{\Delta[s \leftarrow \{x_0..x_l\}] \vdash \langle \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle}{\Delta \vdash \langle \textbf{struct}\ s\{x_0..x_l\}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle} \text{ (eStructDef)}$$

$$\frac{\Delta[u \leftarrow \{x_0..x_l\}] \vdash \langle \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle}{\Delta \vdash \langle \textbf{union}\ u\{x_0..x_l\}, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^s \langle \sigma, \Gamma, \mathfrak{t}\rangle} \text{ (eUnionDef)}$$

Figure 18: Energy-aware semantics for statements.

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

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

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

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

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

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

$$\frac{\Delta \vdash \langle i, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle j, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad j \in [0..length(a) - 1] \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\Delta \vdash \langle a[i], \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma'(a[j]), \sigma', \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{macc}\rangle}\ (\text{eArrayAccess})$$

$$\frac{x \in \Delta(s) \qquad \Gamma' = \Gamma[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\Delta \vdash \langle s.x, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma(s).x, \sigma, \Gamma', \mathfrak{t} + C_{imp} :: \mathfrak{T}_{macc}\rangle}\ (\text{eStructAccess})$$

$$\frac{x \in \Delta(u) \qquad \Gamma' = \Gamma[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\Delta \vdash \langle u.x, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma(u).x, \sigma, \Gamma', \mathfrak{t} + C_{imp} :: \mathfrak{T}_{macc}\rangle}\ (\text{eUnionAccess})$$

$$\frac{\begin{array}{c}\Gamma''' = \Gamma''[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc} + C_{imp} :: \mathfrak{E}_a] \\ \Delta \vdash \langle i, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle j, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad j \in [0..length(a) - 1] \qquad \Delta \vdash \langle e, \sigma', \Gamma', \mathfrak{t}'\rangle \Downarrow^e \langle n, \sigma'', \Gamma'', \mathfrak{t}''\rangle\end{array}}{\Delta \vdash \langle a[i] = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma''[a[j] \leftarrow n], \Gamma''', \mathfrak{t}'' + C_{imp} :: \mathfrak{T}_{macc} + C_{imp} :: \mathfrak{T}_a\rangle}\ (\text{eArrayAssign})$$

$$\frac{x \in \Delta(s) \qquad \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc} + C_{imp} :: \mathfrak{E}_a]}{\Delta \vdash \langle s.x = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma'[\sigma'(s).x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{macc} + C_{imp} :: \mathfrak{T}_a\rangle}\ (\text{eStructAssign})$$

$$\frac{x \in \Delta(u) \qquad \Delta \vdash \langle e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle n, \sigma', \Gamma', \mathfrak{t}'\rangle \qquad \Gamma'' = \Gamma'[C_{imp} :: \mathfrak{e} \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc} + C_{imp} :: \mathfrak{E}_a]}{\Delta \vdash \langle u.x = e, \sigma, \Gamma, \mathfrak{t}\rangle \Downarrow^e \langle \sigma'[\sigma'(u).x \leftarrow n], \Gamma'', \mathfrak{t}' + C_{imp} :: \mathfrak{T}_{macc} + C_{imp} :: \mathfrak{T}_a\rangle}\ (\text{eUnionAssign})$$

Figure 19: Energy-aware semantics for expressions.

A.3 Energy Analysis Rules

The full energy analysis rules as implemented by EcaLogic-C are those in Figure 20.

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

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}e_1\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\}e_2\{\Gamma_2;\ \mathfrak{t}_2;\ \rho_2\} \qquad \Gamma_3 = \Gamma_2[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_e]}{\{\Gamma;\ \mathfrak{t};\ \rho\}e_1 \boxdot e_2\{\Gamma_3;\ \mathfrak{t}_2 + C_{imp} :: \mathfrak{T}_e;\ \rho_2\}}\ \text{(aBinOp)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}e\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_a]}{\{\Gamma;\ \mathfrak{t};\ \rho\}x = e\{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_a;\ \rho_2\}}\ \text{(aAssign)}$$

$$\frac{\Gamma_1 = \Gamma[C_i :: s \leftarrow C_i :: \delta_f(C_i :: s), C_i :: \tau \leftarrow \mathfrak{t}, C_i :: \mathfrak{e}\ \mathrel{+}= C_i :: \mathfrak{E}_f + td(C_i, \mathfrak{t})]}{\{\Gamma;\ \mathfrak{t};\ \rho\}C_i :: f(args)\{\Gamma_1;\ \mathfrak{t} + C_i :: \mathfrak{T}_f;\ \rho\}}\ \text{(aCallCmpF)}$$

$$\frac{\begin{array}{c}\Delta(f) = (e_1, x) \qquad \{\Gamma;\ \mathfrak{t};\ \rho\}e\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \\ e = a \in \rho \qquad \{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1[x' \leftarrow a]\}e_1[x \leftarrow x']\{\Gamma_2;\ \mathfrak{t}_2;\ \rho_2\} \qquad x' \text{ fresh in } e_1\end{array}}{\{\Gamma;\ \mathfrak{t};\ \rho\}f(e)\{\Gamma_2;\ \mathfrak{t}_2;\ \rho_2\}}\ \text{(aCallF)}$$

$$\frac{}{\{\Gamma;\ \mathfrak{t};\ \rho\}\mathbf{skip}\{\Gamma;\ \mathfrak{t};\ \rho\}}\ \text{(aSkip)} \qquad \frac{\{\Gamma;\ \mathfrak{t};\ \rho\}S_1\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\}S_2\{\Gamma_2;\ \mathfrak{t}_2;\ \rho_2\}}{\{\Gamma;\ \mathfrak{t};\ \rho\}S_1;\ S_2\{\Gamma_2;\ \mathfrak{t}_2;\ \rho_2\}}\ \text{(aConcat)}$$

$$\frac{\begin{array}{c}\{\Gamma;\ \mathfrak{t};\ \rho\}e\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_{ite};\ \rho_1\}S_1\{\Gamma_3;\ \mathfrak{t}_2;\ \rho_2\} \\ \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_{ite}] \qquad \{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_{ite};\ \rho_1\}S_2\{\Gamma_4;\ \mathfrak{t}_3;\ \rho_3\}\end{array}}{\{\Gamma;\ \mathfrak{t};\ \rho\}\mathbf{if(}\ e\ \mathbf{)}\ S_1\ \mathbf{else}\ \ S_2\{\mathbf{lub}(\Gamma_3, \Gamma_4);\ \max\{\mathfrak{t}_2, \mathfrak{t}_3\};\ \rho_4\}}\ \text{(aIf)}$$

$$\frac{\begin{array}{c}\Gamma_1 = \mathbf{process\text{-}td}(\Gamma, \mathfrak{t}) \qquad \Gamma_3 = \Gamma_2[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_w] \\ \{\mathbf{wci}(\Gamma_1, e;\ S);\ \mathfrak{t};\ \rho\}e\{\Gamma_2;\ \mathfrak{t}_1;\ \rho_1\} \qquad \{\Gamma_3;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_w;\ \rho_1\}S\{\Gamma_4;\ \mathfrak{t}_2;\ \rho_2\}\end{array}}{\{\Gamma;\ \mathfrak{t};\ \rho\}\mathbf{while(}_{ib}\ e\ \mathbf{)}\ S\{\mathbf{oe}(\Gamma_1, \mathfrak{t}, \Gamma_4, \mathfrak{t}_2, ib);\ \rho_3\}}\ \text{(aWhile)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}e_1\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_{mdec}]}{\{\Gamma;\ \mathfrak{t};\ \rho\}a[e_1]\{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_{mdec};\ \rho_1\}}\ \text{(aArrayDecl)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}i\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\{\Gamma;\ \mathfrak{t};\ \rho\}a[i]\{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_{macc};\ \rho_1\}}\ \text{(aArrayAccess)}$$

$$\frac{\Gamma_1 = \Gamma[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\{\Gamma;\ \mathfrak{t};\ \rho\}s.x\{\Gamma_1;\ \mathfrak{t} + C_{imp} :: \mathfrak{T}_{macc};\ \rho\}}\ \text{(aStructAccess)}$$

$$\frac{\Gamma_1 = \Gamma[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_{macc}]}{\{\Gamma;\ \mathfrak{t};\ \rho\}u.x\{\Gamma_1;\ \mathfrak{t} + C_{imp} :: \mathfrak{T}_{macc};\ \rho\}}\ \text{(aUnionAccess)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}i\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\}e\{\Gamma_2;\ \mathfrak{t}_2;\ \rho_2\} \qquad \Gamma_3 = \Gamma_2[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_a + C_{imp} :: \mathfrak{E}_{macc}]}{\{\Gamma;\ \mathfrak{t};\ \rho\}a[i] = e\{\Gamma_3;\ \mathfrak{t}_2 + C_{imp} :: \mathfrak{T}_a + C_{imp} :: \mathfrak{T}_{macc};\ \rho_2\}}\ \text{(aArrayAssign)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}e\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_a + C_{imp} :: \mathfrak{E}_{macc}]}{\{\Gamma;\ \mathfrak{t};\ \rho\}s.x = e\{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_a + C_{imp} :: \mathfrak{T}_{macc};\ \rho_1\}}\ \text{(aStructAssign)}$$

$$\frac{\{\Gamma;\ \mathfrak{t};\ \rho\}e\{\Gamma_1;\ \mathfrak{t}_1;\ \rho_1\} \qquad \Gamma_2 = \Gamma_1[C_{imp} :: \mathfrak{e}\ \mathrel{+}= C_{imp} :: \mathfrak{E}_a + C_{imp} :: \mathfrak{E}_{macc}]}{\{\Gamma;\ \mathfrak{t};\ \rho\}u.x = e\{\Gamma_2;\ \mathfrak{t}_1 + C_{imp} :: \mathfrak{T}_a + C_{imp} :: \mathfrak{T}_{macc};\ \rho_1\}}\ \text{(aUnionAssign)}$$

Figure 20: Full energy analysis rules

B Array Sorting Code Listing

This appendix contains the code for the Array Sorting example in Section 7.1. It consists of the two algorithms.

B.1 QuickSort

This implementation is loosely adapted from the implementation given by Kernighan & Ritchie[31] in 1988, for the C language. The function pointer argument is omitted, since function calls are expensive and it might dominate an analysis.

```
/*
Basic QuickSort implementation
Uses leftmost element as the pivot
Only operates on lists of integers
*/

#include "eca.acsl"

void swap(int* v, int a, int b) {
  int c = v[a];
  v[a] = v[b];
  v[b] = c;
}

void qsort_r(int* v, int left, int right) {
    int i, last;

    if (left >= right)
        return;
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);
    qsort_r(v, left, last-1);
    qsort_r(v, last+1, right);
}

void qsort(int* v, int length) {
  qsort_r(v, 0, length-1);
}
```

B.2 MergeSort

This code is a slightly modified version from a public implementation[10].

[10] MergeSort C implementation located here: `http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#C`

```
/*
Basic MergeSort implementation
Only operates on lists of integers
*/

#include "eca.acsl"

void merge(int* v, int left_start, int left_end, int right_start,
int right_end)
{
  int left_length = left_end - left_start;
  int right_length = right_end - right_start;

  int left_half[left_length];
  int right_half[right_length];

  int r = 0;
  int l = 0;
  int i = 0;

  for (i = left_start; i < left_end; i++, l++)
    left_half[l] = v[i];

  for (i = right_start; i < right_end; i++, r++)
    right_half[r] = v[i];

  for (i = left_start, r = 0, l = 0; l < left_length && r <
  right_length; i++)
  {
    if (left_half[l] < right_half[r]) v[i] = left_half[l++];
    else v[i] = right_half[r++];
  }

  for ( ; l < left_length; i++, l++) v[i] = left_half[l];
  for ( ; r < right_length; i++, r++) v[i] = right_half[r];
}

void mergesort_r(int* v, int left, int right)
{
  if (right - left <= 1)
    return;

  int left_start  = left;
  int left_end    = (left+right)/2;
  int right_start = left_end;
  int right_end   = right;

  mergesort_r(v, left_start, left_end);
  mergesort_r(v, right_start, right_end);

  merge(v, left_start, left_end, right_start, right_end);
}

void mergesort(int* v, int length)
{
  mergesort_r(v, 0, length);
}
```

C Clock Synchronization Code Listing

This appendix contains the code for the Clock Synchronization example in Section 7.2. It consists of the two algorithms and the hardware component modelling sockets.

C.1 Berkeley Algorithm

```
/*
Basic Berkeley implementation for a master-server
Usage: <executable> N
Listens for N timestamps sent by clients to its socket, then
culls and averages the result, and sends the modification back
Skips the RTT adjustment and any real outlier detection to keep it
simple
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <strings.h>
#include <arpa/inet.h>

#define SERV_UDP_PORT 6543
#define SERV_TCP_PORT 6543
#define SERV_HOST_ADDR "192.168.2.200"

#define TIMESTAMP_TOLERANCE 2000

#include "eca.acsl"

//Computes the average of a set of timestamps, ignores those
beyond a certain threshold of the master's current time
//Also adds the current server time, just in case everything is an
outlier
time_t avg(time_t* timestamps, int n) {
  time_t now = time(NULL);
  if (now < 0) err_dump("Failed to get current time!");

  int realn = n;

  time_t res = now;
  //@ loop invariant ecaloopbound(n, n);
  for (int i=0; i<n; ++i) {
    if (timestamps[i] - now < TIMESTAMP_TOLERANCE && now -
    timestamps[i] < TIMESTAMP_TOLERANCE) res += timestamps[i];
    else --realn;
  }

  return res/realn;
}

int main (int argc, char ** argv, char ** env) {
  int socket_server;
```

```
struct sockaddr_in address_server;
long int testn;

//Determine the amount of clients
if (argc < 2)
  fprintf(stderr, "First argument should be the test count!");

testn = strtol(argv[1], NULL, 10);

int client_sockets[testn];
time_t client_timestamps[testn];

//Open the server socket to listen on
if ((socket_server = socket(AF_INET, SOCK_STREAM, 0) ) < 0)
  err_dump("Server: Can't open stream socket!");

//Bind to the local adress and start listening
bzero((void *) &address_server, (size_t)sizeof(address_server));

address_server.sin_family = AF_INET;
address_server.sin_addr.s_addr = htonl(INADDR_ANY);
address_server.sin_port = htons(SERV_TCP_PORT);

if (bind(
  socket_server,
  (struct sockaddr *) &address_server,
  sizeof(address_server))
    < 0)
  err_dump("Server: Can't bind local address!");

listen(socket_server, 5);

//Receive N timestamps and remember the sockets
//@ assert ecarelation(testn, ecacston(argv[1]));
//@ loop invariant ecaloopbound(0, testn);
for (int i=0; i<testn; ++i) {
  //Accept a connection
  struct sockaddr_in address_client;
  socklen_t client_address_size = sizeof(address_client);
  client_sockets[i] = accept(
      socket_server,
      (struct sockaddr *) &address_client,
      &client_address_size);

  if (client_sockets[i] < 0)
    err_dump("Server: Accept error!");

  //Read a timestamp
  read(client_sockets[i], &(client_timestamps[i]),
  sizeof(time_t));
}

//Compute the average timestamp
//@ assert ecarelation(testn, ecacston(argv[1]));
time_t avg_timestamp = avg(client_timestamps, testn);

//Send the difference to each client
```

```
  //@ assert ecarelation(testn, ecacston(argv[1]));
  //@ loop invariant ecaloopbound(0, testn);
  for (int i=0; i<testn; ++i) {
    //Compute this difference
    time_t diff = avg_timestamp - client_timestamps[i];

    //Write the timestamp
    write(client_sockets[i], &diff, sizeof(time_t));

    //Close the socket, since we don't need it anymore
    close(client_sockets[i]);
  }

  close(socket_server);
}
```

C.2 Cristian's Algorithm

```
/*
Basic Cristian's algorithm implementation for a master-server
Usage: <executable> N
Listens for N timestamps sent by clients to its socket, and
immediately sends the correction required back
RTT adjustment is left to the client
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <strings.h>
#include <arpa/inet.h>

#define SERV_UDP_PORT 6543
#define SERV_TCP_PORT 6543
#define SERV_HOST_ADDR "192.168.2.200"

#include "eca.acsl"

int main (int argc, char ** argv, char ** env) {
  int socket_server;
  struct sockaddr_in address_server;
  long int testn;

  //Determine the amount of clients
  if (argc < 2)
    fprintf(stderr, "First argument should be the test count!");

  testn = strtol(argv[1], NULL, 10);

  int client_socket;
  time_t client_timestamp;

  //Open the server socket to listen on
  if ((socket_server = socket(AF_INET, SOCK_STREAM, 0) ) < 0)
    err_dump("Server: Can't open stream socket!");

  //Bind to the local adress and start listening
```

```
  bzero((void *) &address_server, (size_t)sizeof(address_server));

  address_server.sin_family = AF_INET;
  address_server.sin_addr.s_addr = htonl(INADDR_ANY);
  address_server.sin_port = htons(SERV_TCP_PORT);

  if (bind(
    socket_server,
    (struct sockaddr *) &address_server,
    sizeof(address_server))
      < 0)
    err_dump("Server: Can't bind local address!");

  listen(socket_server, 5);

  //Receive N timestamps and send back the correction
  //@ assert ecarelation(testn, ecacston(argv[1]));
  //@ loop invariant ecaloopbound(0, testn);
  for (int i=0; i<testn; ++i) {
    //Accept a connection
    struct sockaddr_in address_client;
    socklen_t client_address_size = sizeof(address_client);
    client_socket = accept(
        socket_server,
        (struct sockaddr *) &address_client,
        &client_address_size);

    if (client_socket < 0)
      err_dump("Server: Accept error!");

    //Read a timestamp
    read(client_socket, &client_timestamp, sizeof(time_t));

    //Get the current system time
    time_t now = time(NULL);
    if (now < 0) err_dump("Failed to get current time!");

    //Compute the difference
    time_t diff = now - client_timestamp;

    //Send back the difference
    write(client_socket, &diff, sizeof(time_t));

    //Close the socket
    close(client_socket);
  }

  close(socket_server);
}
```

C.3 Network Model

```
/**
 * Example Hardware Component describing energy consumption for
 some C-Socket functions in Java
 */
public class Sockets {
```

```
  //Energy consumption is dependent on the number of active sockets
  public int active_sockets;

  public Sockets() {
    active_sockets = 0;
  }

  @ConsumesAlias(energy = 40, time = 10, alias = "socket")
  public void CreateSocket() {
    ++active_sockets;
    if (active_sockets < 0) active_sockets = Integer.MAX_VALUE;
  }

  @ConsumesAlias(energy = 20, time = 5, alias = "close")
  public void CloseSocket() {
    --active_sockets;
    if (active_sockets < 0) active_sockets = 0;
  }

  @ConsumesAlias(energy = 200, time = 30, alias = "bind")
  public void BindSocket() {
    //Does not change the state
  }

  @ConsumesAlias(energy = 100, time = 100, alias = "listen")
  public void ListenSocket() {
    //Does not change the state
  }

  @ConsumesAlias(energy = 30, time = 20, alias = "accept")
  public void AcceptSocket() {
    //Does not change the state
  }

  public double phi() {
    return 5 + active_sockets * 90;
  }
}
```