Executable Battery Model in SystemC

by

Erning Zhao

A Report submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Engineering

in the Department of Electrical and Computer Engineering

Supervisory Committee

Executable Battery Model in SystemC

by

Erning Zhao

Supervisory Committee

Supervisor
Dr. Daler Rakhmatov, Department of Electrical and Computer Engineering

Departmental Member
Kin Fun Li, Department of Electrical and Computer Engineering

Abstract

Supervisory Committee

Supervisor
Dr. Daler Rakhmatov, Department of Electrical and Computer Engineering

Departmental Member
Kin Fun Li, Department of Electrical and Computer Engineering

As the popularity of battery-powered portable electronic systems continues to grow, an Electronic System Level (ESL) designer may need an executable high-level battery model to make informed battery-aware decisions targeting maximization of the system's online lifetime. This report describes an executable SystemC model with 13 parameters that allows a designer to predict the battery voltage given a sequence of discharge-current loads, thus facilitating ESL design of battery-powered systems. This model also exposes user-controlled trade-offs between numerical accuracy and computational complexity associated with battery voltage calculations.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank:

my Supervisor Dr. Daler Rakhmatov, for mentoring, support, encouragement, and patience. Thank you for being punctilious and rigorous through my graduate study.

my Supervisory committee member Dr. Fun Kin Li, for being on my supervisory committee. Thank you for your time reviewing my project. I also want to express my gratitude to the staff members in the Department of Electrical and Computer Engineering, Janice Closson, Dan Mai, and Amy Rowe.

my friend Feng Zhu, for standing together with me and helping me overcome difficulties. I learned a lot from the discussion and your valuable suggestions on my studies. Thank you for answering my questions patiently and carefully from trivial programming questions to global design problems. The time we spend together is priceless.

my family, my wife Andrea Deng, my son Ethan Zhao and my daughter Flora Zhao, for supporting me, living in Vancouver and allowing me to finish my degree in Victoria.

Dedication

To my families!

Chapter 1

Introduction

1.1 Battery models for electronic systems

Different battery-powered system designs may yield different current loads[1] over time, thus affecting battery voltage behavior. From an Electronic System Level (ESL) design perspective, to prolong a battery-powered electronic system's online lifetime, a designer needs to find a solution that keeps battery voltage above some threshold called "cut-off voltage" as long as possible. This solution requires a model relating design-imposed current loads to the battery voltage behavior. A variety of battery models have been developed to date. Physical models, such as [1] and [2], describe battery's internal physical processes. They are the most accurate models, but are hard to configure and slow to compute. The statistical models, such as [3] and [4], are derived from experimental measurements. They are relatively less precise but fast to implement. Stochastic models, such as [5], model the battery behavior as a stochastic process simulation. Equivalent circuit models, such as [6] and [7], are derived by emulating battery behavior with equivalent electrical circuits. They can be relatively precise but slow to simulate. Finally, analytical models, such as [8] and [9], combine both physical and statistical approaches. They rely on simplified equivalent-battery modeling. Our executable high-level battery model for ESL design described in this report falls into this analytical equivalent-battery model category. Our implementation (using SystemC language) is based on the battery model described in [9]. From a design automation perspective, a high-level analytical model, such as the one from [9], is more suitable than low-level simulation models, such as described in [1], [2], [6], and [7], due to faster computations and analytical insights.

The high-level analytical battery model from [9], combining both physical and statistical approaches, is relatively accurate and robust, as well as relatively fast and compact. It captures battery capacity loss at high discharge rates, charge recovery, and capacity fading over time [9]. The user needs to configure 9 parameters to describe the battery behavior. Additional 4 parameters are introduced to exploit user-controlled trade-offs between accuracy and complexity during battery voltage calculations and battery lifetime estimation.

[1] The term "current load" refers to the discharge current drawn from a battery.

Figure 1.1 shows a generic current load profile that serves as an input to the model, whose output is the battery voltage $V_j(t)$ at time t. In this current load profile, time t is in the interval $[t_j , t_j + d_j)$. Current load $I(t)$ is a staircase function that consists of N steps. Considering the jth step, the current load I_j lasts from t_j to $t_j + d_j$.

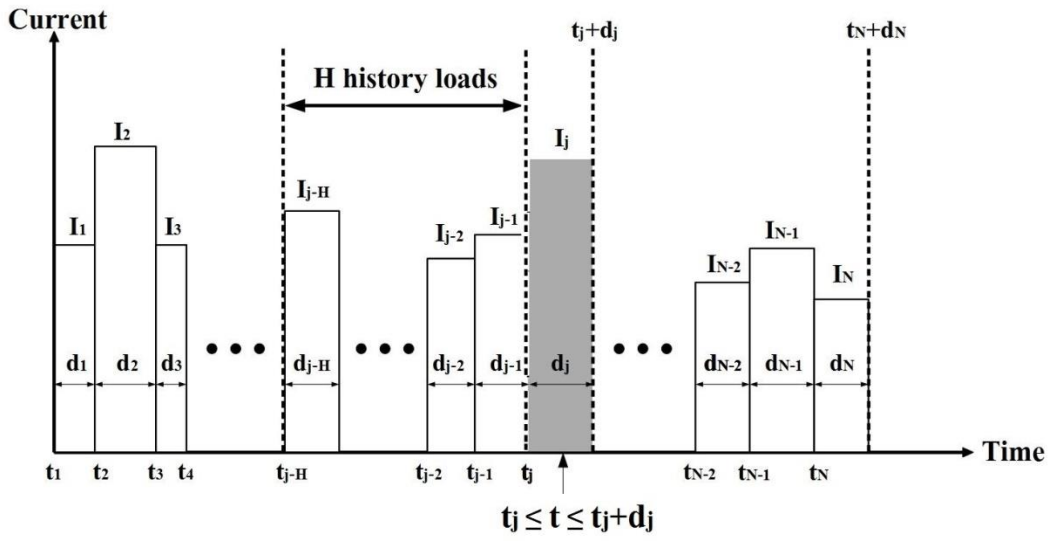


Figure 1.1 Current load profile example

The battery model from [9] is shown below:

$$\mathrm{V}_j(t) = V_0 - rI_j - \varphi\left[\left(\gamma_n + \gamma_p\right)t + ln\frac{\alpha_n + I_j F_{nj} + \sum_{k=1}^{j-1} I_k F_{nk}}{\alpha_p - I_j F_{pj} - \sum_{k=1}^{j-1} I_k F_{pk}}\right] \qquad (1)$$

Using equation (1), given a series of current loads before and at time t (as shown in Figure1.1), we can calculate the battery voltage at t. The F-Factors and S-Series in equation (1) are given in Table 1.1 [9].

Table 1.1 Definitions of F-Factors and S-Series

$$F_{nj} = \frac{e^{-\gamma_n t_j} - e^{-\gamma_n t}}{\gamma_n} + S_{nj}$$ $$S_{nj} = 2e^{-\gamma_n t} \sum_{m=1}^{\infty} \frac{1 - e^{-(\beta_n m^2 - \gamma_n)(t - t_j)}}{\beta_n m^2 - \gamma_n}$$
$$F_{pj} = \frac{e^{\gamma_p t} - e^{\gamma_p t_j}}{\gamma_p} + S_{pj}$$ $$S_{pj} = 2e^{\gamma_p t} \sum_{m=1}^{\infty} \frac{1 - e^{-(\beta_p m^2 + \gamma_p)(t - t_j)}}{\beta_p m^2 + \gamma_p}$$
$$F_{nk} = \frac{e^{-\gamma_n t_k} - e^{-\gamma_n (t_k + d_k)}}{\gamma_n} + S_{nk}$$ $$S_{nk} = 2 \sum_{m=1}^{\infty} \frac{e^{-\gamma_n (t_k + d_k)} \cdot e^{-\beta_n m^2 (t - t_k - d_k)} - e^{-\gamma_n t_k} \cdot e^{-\beta_n m^2 (t - t_k)}}{\beta_n m^2 - \gamma_n}$$
$$F_{pk} = \frac{e^{\gamma_p (t_k + d_k)} - e^{\gamma_p t_k}}{\gamma_p} + S_{pk}$$ $$S_{pk} = 2 \sum_{m=1}^{\infty} \frac{e^{\gamma_p (t_k + d_k)} \cdot e^{-\beta_p m^2 (t - t_k - d_k)} - e^{\gamma_p t_k} \cdot e^{-\beta_p m^2 (t - t_k)}}{\beta_p m^2 + \gamma_p}$$

The proposed model has the following 9 parameters:

- r represents the ohmic resistance (Ohm);
- V_0 represents reference voltage (Volt);
- φ determines the *voltage curve flatness* (Volt);
- α_n and $\alpha_p$determine *initial battery capacity* (Coulomb);
- β_n and $\beta_p$characterize *short-term capacity losses*, usually observed at high discharge rates (1/sec);
- γ_n and γ_p characterize *long-term capacity losses*, usually observed as the battery ages (1/sec).

We have implemented this battery model in SystemC language to ease its integration with existing SystemC code by an ESL designer. SystemC is particularly well-suited for system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis [10]. Our SystemC executable battery model will help ESL designers explore battery-related performance

issues at a higher-level abstraction, and to exploit trade-offs between computational complexity and numerical accuracy.

1.2 Organization of this report

Chapter 2 provides background information to facilitate a better understanding of this project, including a brief description of SystemC language, the battery model in use, and the executable model applications for ESL design.

Chapter 3 introduces our executable battery model implementation in SystemC. We describe modules design in detail, including module classification, module's input and output format, and the data structure in use.

Chapter 4 focuses on application examples and result analysis. First, a trade-off between numerical accuracy and computational complexity is introduced. Then, as a practical application, battery lifetime estimation is described in detail. We present battery lifetime estimation results based on several current load profiles.

Chapter 5 provides a summary of this project and outlines future work directions on this topic.

Chapter 2

Background

2.1 SystemC language

SystemC is a library of C++ classes and macros, which provides an event-driven simulation interface. These facilities enable a designer to simulate concurrent processes, taking the advantages of C++ syntax. SystemC processes can communicate using signals of datatypes offered by C++, and additional ones offered by the SystemC library, as well as the ones defined by a user [10].

As a superset of C++, SystemC inherits all the features of C++. SystemC offers a greater range of expressions, similar to object-oriented C++ design partitions and template classes. Our existing knowledge of C++ can be leveraged when we use SystemC. Source code can be compiled with the SystemC library (which includes a simulation kernel) to produce an executable. SystemC provides a simple form of concurrency, where both parallel execution of statements in different processes and sequential execution of statements within the same process are supported. SystemC includes common hardware-description language features, such as structural hierarchy and connectivity, clock-cycle accuracy, and four-valued logic (0, 1, X, Z). SystemC also contains functions for communication abstraction, transaction-level modeling, and virtual-platform modeling [10]. With SystemC's features, we can use the language as a tool for both hardware and software modeling at an ESL.

2.1.1 Code example in SystemC

The basic elements of SystemC language are module, port, and process. Modules contain processes (C++ methods) and instances of other modules. Ports on modules define their interfaces. Processes are pieces of code that run concurrently with other processes. To illustrate these concepts, we shall use a simple example: hardware adder module with its stimulus module, monitor module, and main module as shown in Figure 2.1.

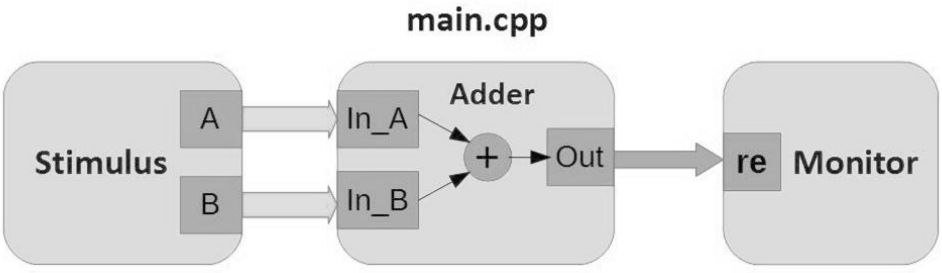
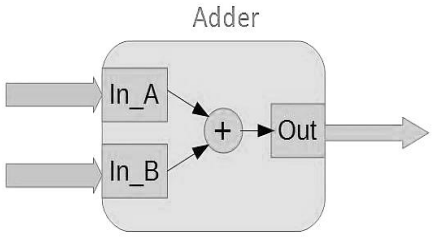


Figure 2.1 Example: "Adder" in SystemC

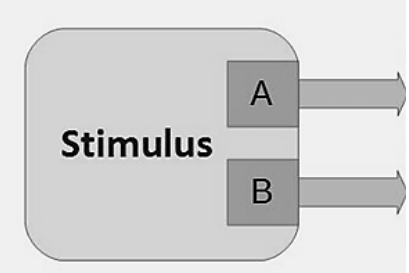
The "Adder" module obtains two integer values from its input ports "In_A" and "In_B", and outputs their sum to output port "Out" (file `adder.h`):

```
#include "systemc.h"
SC_MODULE(Adder)                    // declare Adder as sc_module
{
      sc_in  <int> In_A, In_B;      // input signal ports
      sc_out <int> Out;             // output signal port

      void do_add()                 // a C++ function
      {
            Out.write(In_A.read() + In_B.read());
      }

      SC_CTOR(Adder)
//constructor for Adder
      {
            SC_METHOD(do_add);
// register do_add as SC_METHOD process
            sensitive << In_A << In_B;
// sensitivity list
      }
};
```

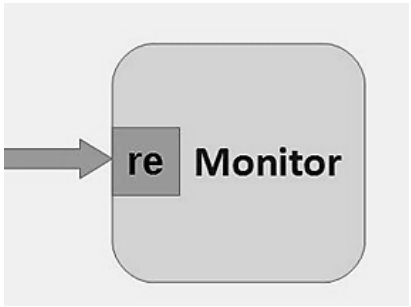


The "Stimulus" module generates stimulation data and outputs the data to its output ports "A" and "B" (file `stimulus.h`):

```
#include "systemc.h"
SC_MODULE(Stimulus)         // declare Stimulus as sc_module
{
      sc_out <int> A, B;    // output signal ports
      void do_write_out()
      {
            // write the ports twice
            A.write(2); B.write(3), wait(10, SC_US);
            A.write(-2); B.write(-3), wait(10, SC_US);
      }
            SC_CTOR(Stimulus)
            //constructor for Stimulus module
      {
            SC_THREAD(do_write_out);
// register do_write_out as SC_THREAD
// process, execute this process only once
      }
};
```



"Monitor" module takes the input data through its "re" input port and prints it out (file `monitor.h`):




```
#include "systemc.h"
#include<iomanip>
using namespace std;
SC_MODULE(Monitor)
{
      sc_in <int> re;
      // print the adder result;
      void monitor() {
            cout << setw(5) << re.read();
            cout << setw(20) << " is the adder result\n";
      }
      SC_CTOR(Monitor) {
            SC_METHOD(monitor);
            dont_initialize(); // don't call this process at time 0;
            sensitive << re;
// register "monitor" as SC_METHOD process
// execute this process when sensitivity list signal "re" changes
      }
};
```

After "Adder", "Stimulus" and "Monitor" modules have been created, we need a main function as a top level module to instantiate and connect these modules. Here is the code for top level module (file `main.cpp`):

```
#include "systemc.h"
#include "adder.h"
#include "stimulus.h"
#include "monitor.h"

int sc_main(int ac, char *av[]){
      // Create signals to connect modules
      sc_signal<int> s1;
      sc_signal<int> s2;
      sc_signal<int> s3;
      // Instantiate modules
      Adder Add("MyAdder");
      Add.In_A(s1);  Add.In_B(s2);  Add.Out(s3);
      Stimulus ST("MyStim");
      ST.A(s1);   ST.B(s2);
      Monitor MON("MyMon");
      MON.re(s3);
      // Simulate for 200 time units
      sc_start(200, SC_US);
      cin.get();//when debugging, waiting keyboard input to keep
window display
      return (0);
}
```

After compiling and running our Adder example, we obtain the following displayed result in Figure 2.2:

```
     SystemC 2.3.1-Accellera --- May 13 2015 20:05:23
     Copyright (c) 1996-2014 by all Contributors,
     ALL RIGHTS RESERVED
 5 is the adder result
-5 is the adder result
```

Figure 2.2 "Adder" example result screenshot

The basic SystemC example above shows us the format of SystemC language elements, which is also summarized below.

A. Modules

Modules are the basic building blocks of a SystemC design hierarchy [10]. A SystemC model usually consists of several modules which communicate via ports. In the SystemC code example above, we introduced an Adder module, coupled with Stimulus module and Monitor module. These three modules communicate via ports as shown in Figure 2.1.

B. Ports

Module ports pass data to and from the processes of a module. We can declare a port direction as "in", "out", or "inout". We can also declare data type of the port as any C++ data type, SystemC data type, or user defined type [11]. Assuming a port's name is p, it can be accessed via signal-type access methods, for example, p->read() and p->write(). Some specialized ports allow using "." instead of "->", i.e., p.read () and p.write (). In the Adder module, we define input ports "sc_in <int> In_A", "sc_in <int > In_B", and output port "sc_out<int> Out". The Adder module reads these two input ports value and writes its output port using the statement "Out.write(In_A.read() + In_B.read())".

C. Processes

SystemC provides processes to support the construction of networks of concurrent pieces of code. There are three types of processes in SystemC:

1 SC_METHOD is a simple C++ method, made sensitive to its inputs [12]. Every time inputs in the sensitivity list change, this method will be triggered. It cannot store control state between two method invocations.

2 SC_THREAD models anything (e.g., testbench), and it is also triggered in response to changes on the input sensitivity list. It can suspend itself and be reactivated.

3 SC_CTHREAD is specifically designed to model synchronous logic. It is triggered in response to a clock edge. It can suspend itself and be reactivated.

In the previous Adder example, we used SC_METHOD process sensitive to any change of signals In_A and In_B.

2.1.2 SystemC testbench

Much like in traditional RTL design, executable SystemC models need a testbench to verify the functionality [13]. In a SystemC testbench, there are a stimulus generator module to generate inputs, a monitor module to analyze results, and a Design under Test (DUT) module to be verified, as shown in Figure 2.3. Our previous Adder example (see Figure 2.1) follows this testbench setup.

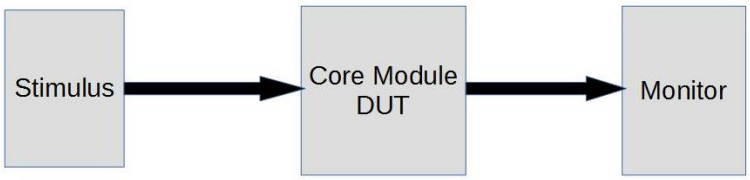


Figure 2.3 Basic testbench setup

Verification of complex systems has become as expensive as design itself. Consequently, verification-oriented languages (such as SystemC) and models are becoming increasingly important [14]. To verify a DUT, there should be a stimulus generator, providing input vector to test the behavior of functions as in core module description. There should also be a monitor module that captures the DUT outputs and analyzes their correctness. By comparing received outputs with the expected ones for the given stimulus, it is possible to determine if the design description is functionally correct [15].

2.2 Battery model description

Equation (1) of the battery model [9] has been introduced in Chapter 1. It has 9 parameters: r, V_0, φ, α_n, α_p, β_n, β_p, γ_n, γ_p. There are also 4 extra parameters for facilitating battery voltage and lifetime computation: *M, H,* $V_{cut-off}$, and *delta.*

In order to calculate battery voltage output, according to equation (1), we need to calculate a pair of S-Series and a pair of F-Factors shown in Table 1.1. Taking S-Series $S_{nj} = 2e^{-\gamma_n t}\sum_{m=1}^{\infty}\frac{1-e^{-(\beta_n m^2-\gamma_n)(t-t_j)}}{\beta_n m^2-\gamma_n}$, for instance, we are faced with the infinite sum. We introduce the *M* parameter that indicates the finite number of terms to be summed during S-Series evaluations. The user can specify any desired value of *M*, thus controlling the computational complexity and accuracy of S-Series calculations.

To further reduce the complexity of voltage calculations, we use *H* previous loads rather than the entire history, i.e. we consider only $I_{j-H}, I_{j-H+1}, I_{j-H+2},\ldots, I_{j-1}$ (see Figure 1.1). In other words, we let:

$$\sum_{k=1}^{j-1} I_k F_{nk} \approx C_{n(j-H-1)} + \sum_{k=j-H}^{j-1} I_k F_{nk} \quad (2)$$

$$\sum_{k=1}^{j-1} I_k F_{pk} \approx C_{p(j-H-1)} + \sum_{k=j-H}^{j-1} I_k F_{pk} \quad (3)$$

Here $C_{n(j-H-1)}$ and $C_{p(j-H-1)}$ are correcting terms representing the current loads missing from the H-load history. The user can specify any desired value of parameter H to control computational complexity and accuracy during battery voltage calculations.

In order to estimate battery lifetime, we need to set the threshold voltage value $V_{cut-off}$, at which the battery is considered to be discharged. We also need parameter $delta$, which is the time increment unit used for searching the point where battery voltage crosses the threshold value $V_{cut-off}$. If $delta$ is large, the amount of computation will decrease, but the precision of a battery lifetime estimates will decrease as well. On the other hand, if $delta$ is small, the precision of battery lifetime estimates will increase accordingly, but the amount of computation will increase as well. This trade-off between efficiency and accuracy, due to the choice of the $delta$ value, is user-controlled.

Thus, we have the total of 13 parameters: $r, V_0, \varphi, \alpha_n, \alpha_p, \beta_n, \beta_p, \gamma_n, \gamma_p, M, H, V_{cut-off}$, and $delta$. These parameters are read from the file ConfigData.dat (see Appendix 1). We shall further discuss parameter settings in Chapter 4.

2.3 Executable battery model application for ESL design

A battery-powered electronic system shuts down when the battery voltage crosses a certain threshold value. Experiments show that a battery charge delivered to a system before its shut-down depends not only on the battery characteristics, but also on system's load profile. From ESL design perspective, battery-powered system's current loads must be carefully managed in order to maximize battery lifetime. Our SystemC executable battery model allows an ESL designer to estimate battery voltage and to predict the battery lifetime, given current loads profile. Having this executable SystemC battery model allows an ESL designer to make battery-aware decisions targeting battery lifetime maximization. Additionally, an ESL designer can exploit a trade-off between numerical accuracy and computational complexity by utilizing different M and H settings in ConfigData.dat file. We shall further discuss battery lifetime estimation in Chapter 4.

Chapter 3

Model implementation in SystemC

3.1 Implementation overview

The main contribution of this project is our SystemC implementation of the battery model proposed in [9]. We have introduced this model briefly in previous chapters. In this chapter, we will focus on the structure of the SystemC code we have developed.

The objective of the executable battery model is to calculate battery voltage at time t, given 13 model parameters and a current load profile. Appendix 2 contains specifications of several current load profiles, designated as Cases 1, 2, 3, 4, 5, and 6, which are based on [16]. Figure 3.1 shows the current profile data of Case 1.

currentProfile_Case1.dat

0.00000	300.0	0.500000
.500000	113.9	25.00000
25.5000	137.9	25.00000
50.5000	234.1	25.00000
75.5000	252.3	25.00000
100.500	494.7	5.00000
105.500	0.000	25.00000
130.500	300.0	0.500000
131.000	628.0	5.00000
136.000	265.6	5.00000

Figure 3.1 Current load profile data format in Case 1

As we can see, every row has three items, namely, starting time t_j, current load I_j, and load duration d_j. Case 1 has 10 rows in total, starting from time stamp 0 to 136 minutes. Note that the unit of starting time is minute, rather than second. Accordingly, the units of β_n, β_p, γ_n, and γ_p mentioned in equation (1) are replaced by 1/min, rather than 1/sec.

We use a doubly-linked list data structure, where each node "Entry" links to the previous node and the next node. As shown in Figure 3.2, every "Entry" has a data set named "Step" consisting of starting time t_j, current load I_j, and load duration d_j. Our doubly-linked list is illustrated in Figure 3.3, where the head is an empty "Entry" linked to the first node and the last node of the list.

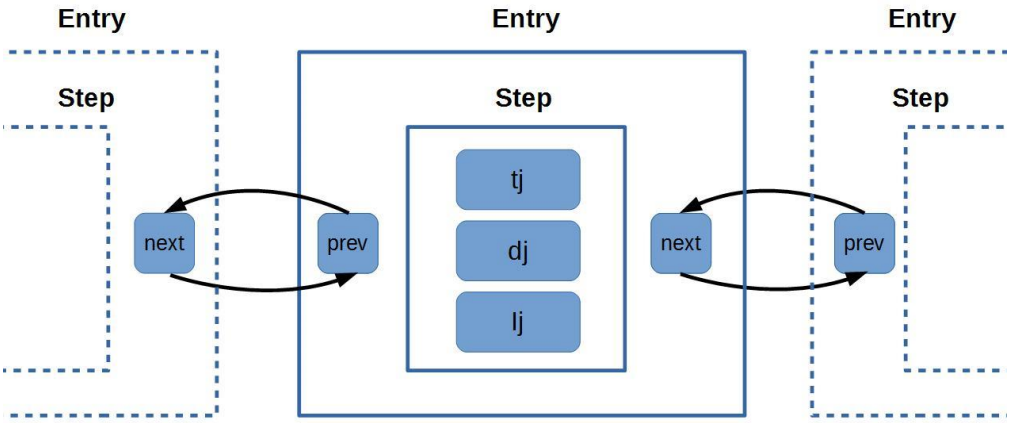


Figure 3.2 "Entry" data structure

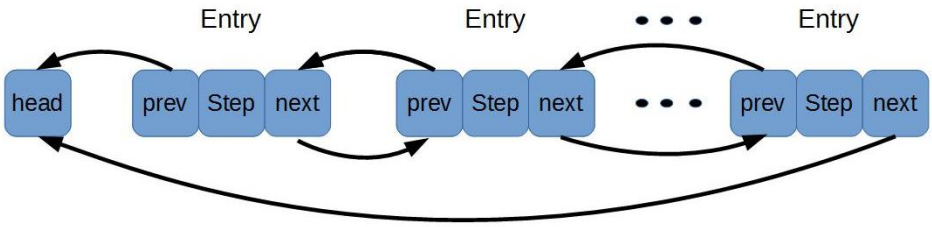


Figure 3.3 Doubly-link list data structure

Our executable battery model also needs 13 configuration parameters. We define them as global variables, read from the file ConfigData.dat (see Appendix 1) by the Computation module. The Stimulus module reads the current load profile data (e.g., currentProfile_Case1.dat), constructs the doubly-linked list data structure shown in Figure 3.3, and provides input data to the Computation module. The Monitor module accepts the calculated battery voltage values from the Computation module, and then prints the result as a display on screen for inspection. These three modules are implemented separately in their corresponding `*.h` and `*.cpp` files. The main function instantiates and connects all these modules together, as shown in Figure 3.4. We shall present further implementation details in later sections.

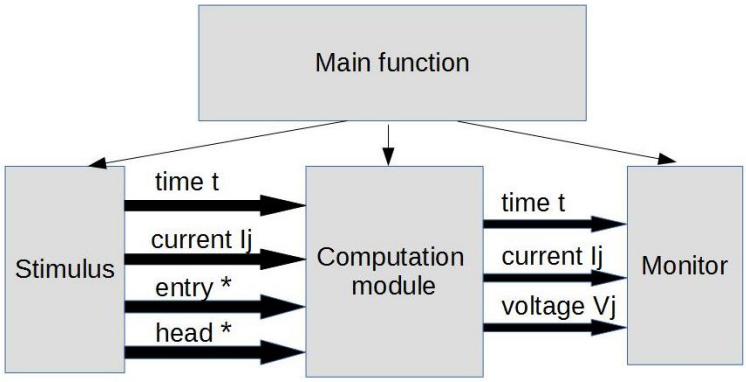


Figure 3.4 Top-level view

3.2 Module implementation in SystemC

3.2.1 Computation module

The computation module has four input ports: the time stamp, the corresponding current load, a pointer pointing to the corresponding entry of the current load profile data, and a pointer pointing to the head of the current load profile. The output port of this module is the calculated battery voltage value at time $t \in [t_j, t_j + d_j)$ (see Figure 1.1).

Recalling equation (1), we need to compute $\ln \frac{\alpha_n + I_j F_{nj} + \sum_{k=1}^{j-1} I_k F_{nk}}{\alpha_p - I_j F_{pj} - \sum_{k=1}^{j-1} I_k F_{pk}}$. We define functions `Fnj(double t, Entry *entry)` and `Fpj(double t, Entry *entry)` for calculating F-Factors, i.e. F_{nj} and F_{pj}, as shown in Table 1.1. These functions take two parameters: the time t and the pointer to the entry of a current load profile under consideration. For example, the function to calculate F_{nj} is shown below:

```
double battery_voltage::Fnj(double t, Entry *entry)
{
      double Fnj_temp = 0;
      double Snj_temp = 0;
      for (int m = 1; m <= M; m++) {
            Snj_temp += (1 - exp(-(beta_n*m*m - gamma_n)*(t - entry
->step->startTime))) / (beta_n*m*m - gamma_n);
      }
      Snj_temp = 2 * exp(-gamma_n*t)*Snj_temp;
      Fnj_temp = (exp(-gamma_n*(entry->step->startTime)) - exp(-
gamma_n*t)) / gamma_n + Snj_temp;
      return Fnj_temp;
}
```

To calculate $\sum_{k=1}^{j-1} I_k F_{nk}$ and $\sum_{k=1}^{j-1} I_k F_{pk}$, we define functions `cal_IkFnk(Entry *head, Entry *last, double t)` and `cal_IkFpk(Entry *head, Entry *last, double t)`. The functions take three parameters: the pointer to the first `Entry` of current load profile, the pointer to the last `Entry` of current load profile, and the time t. Detailed code can be found in Appendix 3.

Finally, we declare the overall computation function `Voltage_Computation()` as an SC_METHOD process, sensitive to the input ports t and I_j, which means the process will be triggered whenever t and/or I_j change.

As we have mentioned earlier, the Computation module is also responsible for reading the battery configuration file specifying 13 parameters. This is accomplished by function `read_configData()`, whose code is also given in Appendix 3.

3.2.2 Stimulus module

The Stimulus module has four output ports, which feed the Computation module with the time stamp, the corresponding current load, the pointer to the entry of current load profile, and the pointer to the head of current load profile. These outputs are written by an SC_THREAD process `do_write_out()`, whose code is shown below.

```
void stimulus::do_write_out()
{
      Entry* entry;
      double start_time, execution_time;

      for (entry = _head->next; entry != _head; entry = entry->next)
      {
            head_out.write(_head);
            entry_out.write(entry);
            start_time = entry->step->startTime;
            execution_time = entry->step->loadDuration;
            Ij.write(entry->step->currentLoad);
            //write t with the starting time of interval
            t.write(start_time);
            //wait for duration time, to write t second time
            wait(execution_time * 60, SC_SEC);
            //keep the values on ports Ij, head_out, and entry_out
            //write t with the end time of interval
            t.write(start_time + execution_time);
            //wait 0 second, be ready to write t with the starting
time of next interval
            wait(0, SC_SEC);
      }
}
```

Note that the above function outputs two time stamps: the start time as well as the finish time of the load in question. Consequently, the Computation module would calculate the battery voltage twice, at the beginning and the end of that load.

The Stimulus module also contains function `read_load_profile()` for reading the current load profile and creating the corresponding doubly-linked list of loads, as shown in Figure 3.2 and Figure 3.3. The return value of this function is a pointer to the entry of the doubly-linked list data structure. The detailed code can be found in Appendix 3.

3.2.3 Monitor module

The input ports of the Monitor module are the calculated battery voltage $Vj(t)$, the time stamp t, and the corresponding current load I_j. The monitor function is defined as an SC_METHOD process, which is sensitive to input signals $Vj(t)$ and t, i.e., it will execute whenever $Vj(t)$ and/or t change.

Taking advantage of output stream "cout", we print out the battery voltage result coming from the computation module, the related time stamp t, and the corresponding current load I_j. Figure 3.5 shows the standard debugging window in Visual Studio 2013 programing environment (displaying the results of Case 1). Detailed code of the Monitor module can be found in Appendix 3.

```
Info: (I703) tracing timescale unit set: 1 s (trace.vcd)
Time   =      0 min,    Current =     300 mA,    Voltage = 3.9614 V
Time   =    0.5 min,    Current =     300 mA,    Voltage = 3.92165 V
Time   =    0.5 min,    Current =   113.9 mA,    Voltage = 3.99609 V
Time   =   25.5 min,    Current =   113.9 mA,    Voltage = 3.88943 V
Time   =   25.5 min,    Current =   137.9 mA,    Voltage = 3.87983 V
Time   =   50.5 min,    Current =   137.9 mA,    Voltage = 3.81268 V
Time   =   50.5 min,    Current =   234.1 mA,    Voltage = 3.7742 V
Time   =   75.5 min,    Current =   234.1 mA,    Voltage = 3.69602 V
Time   =   75.5 min,    Current =   252.3 mA,    Voltage = 3.68874 V
Time   =  100.5 min,    Current =   252.3 mA,    Voltage = 3.62067 V
Time   =  100.5 min,    Current =   494.7 mA,    Voltage = 3.52371 V
Time   =  105.5 min,    Current =   494.7 mA,    Voltage = 3.47679 V
Time   =  105.5 min,    Current =       0 mA,    Voltage = 3.67467 V
Time   =  130.5 min,    Current =       0 mA,    Voltage = 3.71363 V
Time   =  130.5 min,    Current =     300 mA,    Voltage = 3.59363 V
Time   =    131 min,    Current =     300 mA,    Voltage = 3.58412 V
Time   =    131 min,    Current =     628 mA,    Voltage = 3.45292 V
```

Figure 3.5 Example of monitor's display

3.2.4 Top module

In SystemC the main function is designated by sc_main(). First, it instantiates all needed top-level modules and creates connections among them, and then starts the simulation by calling the function sc_start(). The simulation kernel takes control and executes the processes of each module depending on the events occurring at every simulated cycle [17]. The example code for our main function is given below:

```
int sc_main(int argc, char* argv[]) {// give access to argc and argv
// from outside of sc_main
      // Elaboration, to create signals to tie modules
      sc_signal<double> t;          // time stamp t
      sc_signal<double> Ij;         // current load at t
      sc_signal<Entry* > entry_signal;// pointer to the last entry
      sc_signal<Entry* > head_signal; // pointer to the first entry
      sc_signal<double> t_out;      // time stamp t for monitor module
      sc_signal<double> Ij_out;     // current load for monitor module
      sc_signal<double> Vjt;        // battery voltage result
      ////////////////////////////////////////////////////////////
      //instantiation of stimulus, computation and monitor modules
      ////////////////////////////////////////////////////////////
      stimulus TB("battery_module_stimulus",
"currentProfile_Case1.dat");
      TB.entry_out(entry_signal); TB.head_out(head_signal);
TB.Ij(Ij); TB.t(t);

      battery_voltage DUT("battery_module", "configData.dat",
"resultfile.dat");
     DUT.entry_in(entry_signal); DUT.head_in(head_signal); DUT.t(t);
DUT.Ij(Ij); DUT.t_out(t_out); DUT.Ij_out(Ij_out); DUT.Vjt(Vjt);

      mon Monitor("Monitor");
      Monitor.t(t_out); Monitor.Ij(Ij_out); Monitor.Vjt(Vjt);
      // Start the main simulation thread
      sc_start();
      cin.get();//when debugging, waiting keyboard input to keep
window display
      return 0;
}
```

As we can see in the code above, we instantiate our Stimulus module, Computation module, and Monitor module, and then connect them together using appropriate signals. This main function also passes the parameters to the module constructors. For example, we pass the name of module and name of current load profile to the Stimulus module instance as follows:

```
stimulus TB ("battery_module_stimulus","currentProfile_Case1.dat");
```

By changing the second parameter (the file name of a load profile specification), we can subject our executable battery model to different current loads.

Chapter 4

Model application and result analysis

4.1 Computational trade-offs

4.1.1 M settings

When evaluating S-Series, such as $S_{nj} = 2e^{-\gamma_n t}\sum_{m=1}^{\infty}\frac{1-e^{-(\beta_n m^2-\gamma_n)(t-t_j)}}{\beta_n m^2-\gamma_n}$ for example, we encounter an infinite summation. To handle this, we introduce a parameter M that specifies a finite number of terms to be summed. In our executable battery model, the user can specify any desired value of M to control the computational complexity and accuracy of S-Series evaluations. A larger M yields a better approximation; however, it requires more computations. To find the appropriate M setting that strikes a good trade-off between accuracy and complexity, we explored the influence of different M settings using the Case 1 load profile as an example. In our experiments, we have tried M equal to 10000, 1000, 100, and 10. Figure 4.1 shows the results for M = 10000 and M = 1000, while Figure 4.2 shows the results for M = 100 and M = 10.

M=10000	M=1000
Info: (I703) tracing timescale unit set: 1 s Time = 0.25 min, Voltage = 3.94031 V Time = 13 min, Voltage = 3.94206 V Time = 38 min, Voltage = 3.85111 V Time = 63 min, Voltage = 3.73913 V Time = 88 min, Voltage = 3.6631 V Time = 103 min, Voltage = 3.50335 V Time = 118 min, Voltage = 3.72315 V Time = 130.75 min, Voltage = 3.596 V Time = 133.5 min, Voltage = 3.40617 V Time = 138.5 min, Voltage = 3.5337 V	Info: (I703) tracing timescale unit set: 1 s Time = 0.25 min, Voltage = 3.94033 V Time = 13 min, Voltage = 3.94206 V Time = 38 min, Voltage = 3.85112 V Time = 63 min, Voltage = 3.73914 V Time = 88 min, Voltage = 3.66311 V Time = 103 min, Voltage = 3.50338 V Time = 118 min, Voltage = 3.72315 V Time = 130.75 min, Voltage = 3.59601 V Time = 133.5 min, Voltage = 3.40621 V Time = 138.5 min, Voltage = 3.53372 V

Figure 4.1 Screenshots for M equal 10000 and 1000

M=100	M=10
Info: (I703) tracing timescale unit set: 1 s Time = 0.25 min, Voltage = 3.94053 V Time = 13 min, Voltage = 3.94211 V Time = 38 min, Voltage = 3.85116 V Time = 63 min, Voltage = 3.7392 V Time = 88 min, Voltage = 3.6632 V Time = 103 min, Voltage = 3.50364 V Time = 118 min, Voltage = 3.72315 V Time = 130.75 min, Voltage = 3.59614 V Time = 133.5 min, Voltage = 3.40667 V Time = 138.5 min, Voltage = 3.53394 V	Info: (I703) tracing timescale unit set: 1 s Time = 0.25 min, Voltage = 3.94249 V Time = 13 min, Voltage = 3.94256 V Time = 38 min, Voltage = 3.85155 V Time = 63 min, Voltage = 3.73986 V Time = 88 min, Voltage = 3.66405 V Time = 103 min, Voltage = 3.50609 V Time = 118 min, Voltage = 3.72315 V Time = 130.75 min, Voltage = 3.59737 V Time = 133.5 min, Voltage = 3.41095 V Time = 138.5 min, Voltage = 3.53601 V

Figure 4.2 Screenshots for M equal 100 and 10

Note that for M=10000, which yields the most accurate S-Series evaluations among the four settings tested, the battery voltage values are very similar to those obtained using M=10. Table 4.1 shows the battery voltage difference when using M = 10000 and M = 10. The largest difference in Table 4.1 is 0.1403%, which suggests that M = 10 is a reasonable setting.

Table 4.1 Results comparison when M=10000 and M=10

t	Vj (M=10000)	Vj (M=10)	Difference |%|
138.5	3.5337	3.53601	0.0654%
133.5	3.40617	3.41095	0.1403%
130.75	3.596	3.59737	0.0381%
118	3.72315	3.72315	0%
103	3.50335	3.50609	0.0782%
88	3.6631	3.66405	0.0259%
63	3.73913	3.73986	0.0195%
38	3.85111	3.85155	0.0114%
13	3.94206	3.94256	0.0127%
0.25	3.94031	3.94249	0.0553%

4.1.2 H settings

Parameter H is another controllable variable that helps simplify battery voltage computation in our executable battery model. As we can see in Figure 1.1, H represents the number of most recent current loads prior to a time interval $[t_j, t_j + d_j]$. We can compute the battery voltage using only H previous loads $I_{j-H}, I_{j-H+1}, I_{j-H+2}, \ldots, I_{j-1}$, as opposed to considering the entire load history. To accomplish this, we introduce correcting terms correcting terms $C_{n(j-H-1)}$ and $C_{p(j-H-1)}$ representing the missing loads $I_1, I_2, \ldots\ldots, I_{j-H-1}$, as shown in formulas (2) and (3) in Chapter 2. When index $(j - H - 1)$ is larger than 1, we use the following equations:

$$\mathrm{C_{n(j-H-1)} = C_{n(j-H-2)} + I_{j-H-1}F_{n(j-H-1)}} \tag{4}$$

$$\mathrm{C_{p(j-H-1)} = C_{p(j-H-2)} + I_{j-H-1}F_{p(j-H-1)}} \tag{5}$$

Since $\mathrm{C_{n(j-H-2)}}$ and $\mathrm{C_{p(j-H-2)}}$ are already known via recursion from previously calculation of the $\mathrm{V_{j-H-1}(t)}$, we only need to calculate the values of $\mathrm{I_{j-H-1}F_{n(j-H-1)}}$ and $\mathrm{I_{j-H-1}F_{p(j-H-1)}}$. Consequently, we need to evaluate the series $\mathrm{S_{n(j-H-1)}}$ and $\mathrm{S_{p(j-H-1)}}$ in order to calculate $\mathrm{F_{n(j-H-1)}}$ and $\mathrm{F_{p(j-H-1)}}$ for the present time $\mathrm{t_j}$, and then compute the battery voltage $\mathrm{V_j(t)}$. To avoid these series evaluations, we reuse the values of $\mathrm{S_{n(j-H-1)}}$ and $\mathrm{S_{p(j-H-1)}}$ computed for the past time instance $\mathrm{t^* \in [t_{j-1}, t_{j-1}+d_{j-1}]}$ corresponding to previous calculations of $\mathrm{V_{j-1}(t^*)}$. After replacing index j with $\mathrm{j-1}$ in formula (2) and (3), as if computing $\mathrm{V_{j-1}(t^*)}$ we obtain the equations (6) and (7), described below:

$$\sum_{k=1}^{j-2} I_k F_{nk} \approx C_{n(j-H-2)} + I_{j-H-1}F_{n(j-H-1)} + I_{j-H}F_{n(j-H)} + \cdots + I_{j-2}F_{n(j-2)} \tag{6}$$

$$\sum_{k=1}^{j-2} I_k F_{pk} \approx C_{p(j-H-2)} + I_{j-H-1}F_{p(j-H-1)} + I_{j-H}F_{p(j-H)} + \cdots + I_{j-2}F_{p(j-2)} \tag{7}$$

We simply use the first two terms on the right-hand sides as $\mathrm{C_{n(j-H-1)}}$ and $\mathrm{C_{p(j-H-1)}}$. Instead of recomputing $I_{j-H-1}F_{n(j-H-1)}$ and $I_{j-H-1}F_{p(j-H-1)}$, we reuse them from the previous calculation. As a result, we only need to recompute at most H terms for the new value of t, while the remaining $(\mathrm{j - H - 1})$ terms are reused. By following these steps, the complexity of computing battery voltage will drop significantly.

Another important consequence of the user's choice of H is related to memory requirements. It is possible to limit the size of our doubly-linked list only to H+1 entry (not including the terminating head entry). Such a list would be constructed dynamically: if there are already H+1 load entries present, appending a new load entry would involve deletion of the oldest load entry from the list. Shorter load histories (smaller H) would

require shorter linked lists to store the load information used during battery voltage calculations, which essentially would reduce memory requirements but also reduce the computational accuracy.

If we take a look at the file CurrentProfile_Case1.dat in Appendix 2, we will find that there are 10 loads. Therefore, possible values of H would be between 0 and 9: H=0 corresponds to the worst-case approximation (no load history accounted for), while H=9 corresponds to the best-case scenario of no approximation, since the entire current load profile is used for voltage calculation. Figure 4.3 compares the results for H=0 and H=9. From the voltages listed in the box, we can see that the results are notably different.

H=0, M=10	H=9, M=10
Info: (I703) tracing timescale unit set: 1 Time = 0.25 min, Voltage = 3.94249 V Time = 13 min, Voltage = 3.9426 V Time = 38 min, Voltage = 3.85155 V Time = 63 min, Voltage = 3.73977 V Time = 88 min, Voltage = 3.6638 V Time = 103 min, Voltage = 3.50536 V Time = 118 min, Voltage = 3.71711 V Time = 130.75 min, Voltage = 3.59046 V Time = 133.5 min, Voltage = 3.39897 V Time = 138.5 min, Voltage = 3.52018 V	Info: (I703) tracing timescale unit set: Time = 0.25 min, Voltage = 3.94249 V Time = 13 min, Voltage = 3.94256 V Time = 38 min, Voltage = 3.85155 V Time = 63 min, Voltage = 3.73986 V Time = 88 min, Voltage = 3.66405 V Time = 103 min, Voltage = 3.50609 V Time = 118 min, Voltage = 3.72315 V Time = 130.75 min, Voltage = 3.59737 V Time = 133.5 min, Voltage = 3.41095 V Time = 138.5 min, Voltage = 3.53601 V

Figure 4.3 Different H setting with results

After trying different H values, we have found that H=3 is a reasonable setting. The battery voltage computed by our executable battery model when H=3 and H=9 is shown in Figure 4.4 and in Table 4.2.

H=9, M=10	H=3, M=10
Info: (I703) tracing timescale unit set: 1 Time = 0.25 min, Voltage = 3.94249 V Time = 13 min, Voltage = 3.94256 V Time = 38 min, Voltage = 3.85155 V Time = 63 min, Voltage = 3.73986 V Time = 88 min, Voltage = 3.66405 V Time = 103 min, Voltage = 3.50609 V Time = 118 min, Voltage = 3.72315 V Time = 130.75 min, Voltage = 3.59737 V Time = 133.5 min, Voltage = 3.41095 V Time = 138.5 min, Voltage = 3.53601 V	Info: (I703) tracing timescale unit set: 1 Time = 0.25 min, Voltage = 3.94249 V Time = 13 min, Voltage = 3.94256 V Time = 38 min, Voltage = 3.85155 V Time = 63 min, Voltage = 3.73986 V Time = 88 min, Voltage = 3.66406 V Time = 103 min, Voltage = 3.5061 V Time = 118 min, Voltage = 3.72317 V Time = 130.75 min, Voltage = 3.59739 V Time = 133.5 min, Voltage = 3.41098 V Time = 138.5 min, Voltage = 3.53604 V

Figure 4.4 Comparison between H=9 and H=3

Table 4.2 Results comparison when H=3 and H=9

t	Vj (H=3 M=10)	Vj (H=9 M=10)	Difference |%|
138.5	3.53604	3.53601	0.0008%
133.5	3.41098	3.41095	0.0009%
130.75	3.59739	3.59737	0.0000%
118	3.72317	3.72315	0.0000%
103	3.5061	3.50609	0.0000%
88	3.66406	3.66405	0.0000%
63	3.73986	3.73986	0.0000%
38	3.85155	3.85155	0.0000%
13	3.94256	3.94256	0.0000%
0.25	3.94249	3.94249	0.0000%

From Table 4.2, we can see that the case of M=10 and H=3 strikes a good balance between accuracy and complexity, with the maximum difference less than 0.001%.

To explore the impact of M and H settings on the battery voltage computation time, we performed 4 experiments with different M and H combinations, taking the current load profile of Case 3 as an example (8 current loads). We measured the CPU time elapsed while computing $Vj(t)$ at t=110.5min at the beginning of the last current load Ij=265.6 mA. The CPU in question was Intel's i5-3337U operating at 1.80 GHz with 8 GB RAM.

Our timing measurements are shown in Table 4.3. We can see that the calculated battery voltage value is the same, but the elapsed time varies from 2.0 ms (M = 1000, H = 0) to 46.03 ms (M = 10000, H = 7). For M = 100 and M = 10 (regardless of H), our timing measurements produced 0.0 ms, which was discarded as unusable data for comparison purposes.

Table 4.3 Battery voltage computation time measurement

	M=10000	M=1000
H=7	time=46.03ms, voltage = 3.58V	time=2.0ms, voltage =3.58 V
H=0	time=24.01ms, voltage = 3. 58V	time=2.0ms, voltage =3.58 V

In summary, the user can specify any desired value of M and H to reach an acceptable balance between accuracy and complexity by modifying configData.dat parameter file in our executable battery model.

4.2 Battery lifetime estimation

One practical application of our executable battery model is battery lifetime estimation. The cut-off voltage $V_{cut-off}$ is the prescribed lower-limit voltage at which a battery is considered to be exhausted [18]. We define the battery lifetime as the time stamp when the battery voltage crosses the cut-off voltage $V_{cut-off}$. The user can take advantage of this model to make battery-aware decisions based on this calculated battery lifetime, e.g., deciding on the tasks' schedule and tasks execution operation clock frequency order to minimize the battery drain, to prolong the battery life with decreasing system current load for battery recovery when the battery voltage is approaching $V_{cut-off}$, or to save important data before battery-powered system shuts down. This section describes the implementation of battery lifetime estimation. Different current loads schedules will be compared in terms of their impact on battery lifetime.

4.2.1 Implementation of battery lifetime estimation

To estimate the battery lifetime, our goal is to find the time stamp t in equation (1) at which the calculated battery voltage becomes less than $V_{cut-off}$. First, we will take a closer look at the Case 2 load profile (see Appendix 2). It has eight current loads, where the peak load of 628.0 mA happens from 105.5 min to 110.5 min, as shown in Figure 4.5.

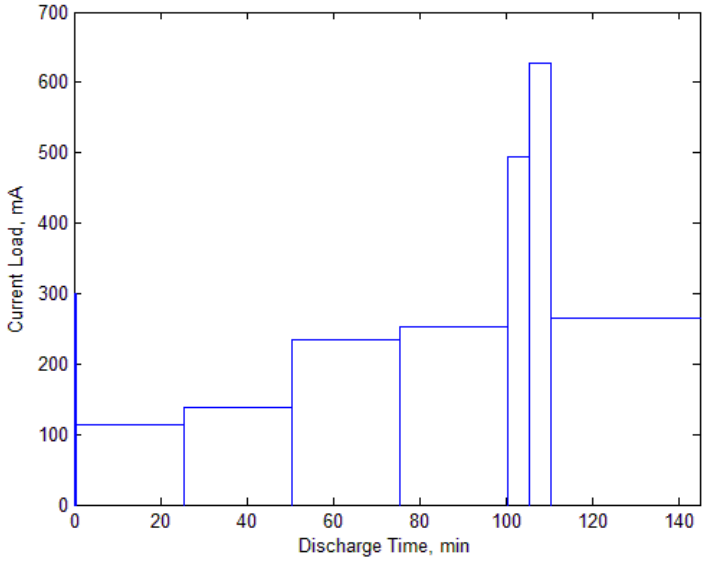


Figure 4.5 Case 2 with high peak load of 628.0mA from 105.5min to 110.5min

For illustrative purposes, we set $V_{cut-off}$ = 3.4V in our configData.dat. We can traverse all time stamps along the time axis and compare every calculated battery voltage at every time stamp against $V_{cut-off}$. However, such a comparison method is highly inefficient.

Instead, we only sample critical time stamps: the starting time and the end time of every current load. Taking the peak current load interval (starting from 105.5min to 110.5min) for example, the stimulus module supplies both t = 105.5 min and 110.5 min to the computation module's input port "t", as shown in Figure 3.4. Then computation module calculates the battery voltages at both time stamps. The battery voltage calculation yields 3.42347V at 105.5 min, which is above $V_{cut-off}$, and 3.35313V at 110.5 min, which is below $V_{cut-off}$. From the above observation, we can tell that the battery "died" between 105.5 min and 110.5 min. Figure 4.6 shows the displayed output of our battery model. We can see that all battery voltages calculated at boundaries of load time intervals are listed. The "death" time estimate, calculated by our model, is also printed at the end.

```
Time   =      0 min,   Current =     300 mA,    Voltage = 3.9614 V
Time   =    0.5 min,   Current =     300 mA,    Voltage = 3.92165 V
Time   =    0.5 min,   Current =   113.9 mA,    Voltage = 3.99609 V
Time   =   25.5 min,   Current =   113.9 mA,    Voltage = 3.88943 V
Time   =   25.5 min,   Current =   137.9 mA,    Voltage = 3.87983 V
Time   =   50.5 min,   Current =   137.9 mA,    Voltage = 3.81268 V
Time   =   50.5 min,   Current =   234.1 mA,    Voltage = 3.7742 V
Time   =   75.5 min,   Current =   234.1 mA,    Voltage = 3.69602 V
Time   =   75.5 min,   Current =   252.3 mA,    Voltage = 3.68874 V
Time   =  100.5 min,   Current =   252.3 mA,    Voltage = 3.62067 V
Time   =  100.5 min,   Current =   494.7 mA,    Voltage = 3.52371 V
Time   =  105.5 min,   Current =   494.7 mA,    Voltage = 3.47679 V
Time   =  105.5 min,   Current =     628 mA,    Voltage = 3.42347 V

Battery failure detected at time interval from 105.5 to 110.5 min
Current load = 628 mA, Vcutoff = 3.4 V

Using fine-grain timing accuracy of 0.1 min
Battery dies at  107 min, Battery voltage = 3.40009 V
```

Figure 4.6 Battery lifetime in Case 2

To search the time stamp of battery "death" in the time interval from 105.5 min to 110.5 min, we use $delta$ = 0.1 min, which determines the timing granularity. The code for detecting the battery "death" time is shown below.

```
////////////////////////////////////////////////////////
//battery discharge time stamp searching
////////////////////////////////////////////////////////
if (result_temp < Vcut) {
      cout << "\nBattery failure detected at time interval from "
<<entry->step->startTime<<" to "<< now << " min\n";
```

```
	cout << "Current load = " << Ij_buf << " mA, Vcutoff = " <<
Vcut << " V\n\n";
	// for now - delta, - delta, ... until start_time, rec-
calculate Vj until detecting > Vcut;
	// Print out cut-off time
	for (right_now = now - delta; right_now > entry->step-
>startTime; right_now -= delta)
	{
		double numerator_new = alpha_n + Ij_buf*Fnj(right_now,
entry) + cal_IkFnk(head, entry, right_now);
		double denominator_new = alpha_p - Ij_buf*Fpj(right_now,
entry) - cal_IkFpk(head, entry, right_now);
		if (denominator_new > 0) {
			result_temp_buf = V0 - r*Ij_buf / 1000 -
phi*((gamma_n + gamma_p)*right_now + log(numerator_new /
denominator_new));
			if (result_temp_buf >= Vcut)
			//supposed to find a point that makes v bigger than
Vcut at current Ij;
			{
				cout << "Using fine-grain timing accuracy of
" << delta << " min\n";
				cout << "Battery dies at  " << right_now << "
min, Battery voltage = " << result_temp_buf << " V\n\n";
				fprintf(savefile, "%lf %lf %lf\n", right_now,
Ij_buf, result_temp_buf);
				break;
			}
		}
		else {
			cout << "Out-of-bounds denominator detected at time
" << now << " min (load current = " << Ij_buf << ")\n\n";
			sc_stop();
			}
	}
	// if we could not find the time during the traversal
	if (result_temp_buf < Vcut)
		// battery must die at the starting time of interval
	{
		cout << "Battery dies at the start of " << entry->step-
>startTime << " min,  Battery voltage = " << result_temp << " V\n\n";
	}
	//when battery voltage crosses the Vcut-off, the system will
shut down, the calculation after this point is meaningless
	//stop the simulation at this time.
	sc_stop();
}
```

As shown in the code, we loop through each time stamp from the end time of the failing load, in *delta* decrements towards the starting time of that load. At each time stamp, we calculate the corresponding battery voltage: if it crosses the threshold $V_{cut-off}$, we break the loop and report the time stamp under consideration as “death” time.

Figure 4.7 shows us the battery voltage curve associated with the Case 2 load profile. We can see that the battery voltage crosses $V_{cut-off}$ = 3.4V in the time interval from 105.5 min to 110.5 min, which corresponds to the peak load. During that time interval the voltage values are calculated with $delta$ time resolution. The threshold value of 3.4 is reached at 107.0 min.

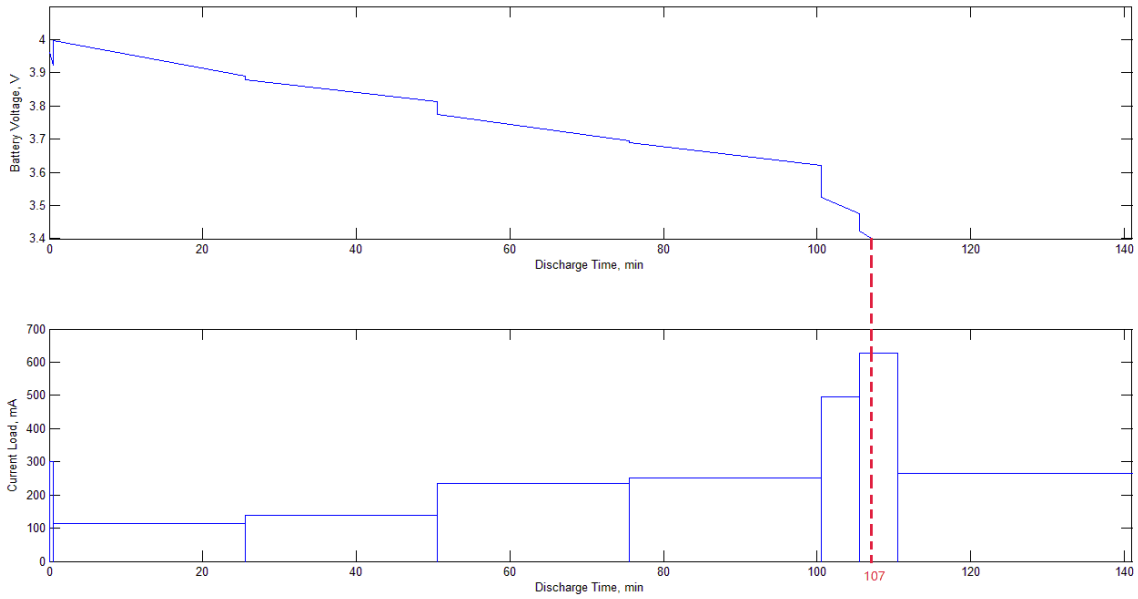


Figure 4.7 Battery voltage in Case 2

4.2.2 Battery lifetime comparisons

From an ESL design perspective, we want to see how different current load schedules influence the battery lifetime. In the following sections, we will experiment with different current load schedules and compare their corresponding battery lifetime.

4.2.2.1 Heavy peak load

Recalling Figure 4.5, the battery "died" during the peak load interval in Case 2. We rearranged current loads in Case 2 to obtain a different load profile, designated here as Case 3 and shown in Figure 4.8. We can see from Figure 4.5 and Figure 4.8, that both cases have the peak load of 628 mA. The first current load of Case 2 and Case 3 is the same (300 mA for 0.5 mins), and so is the last current load (265.6 mA for 35 mins). The remaining loads are arranged in non-increasing order in Case 3, and their counterparts in Case 2 are scheduled in increasing order.

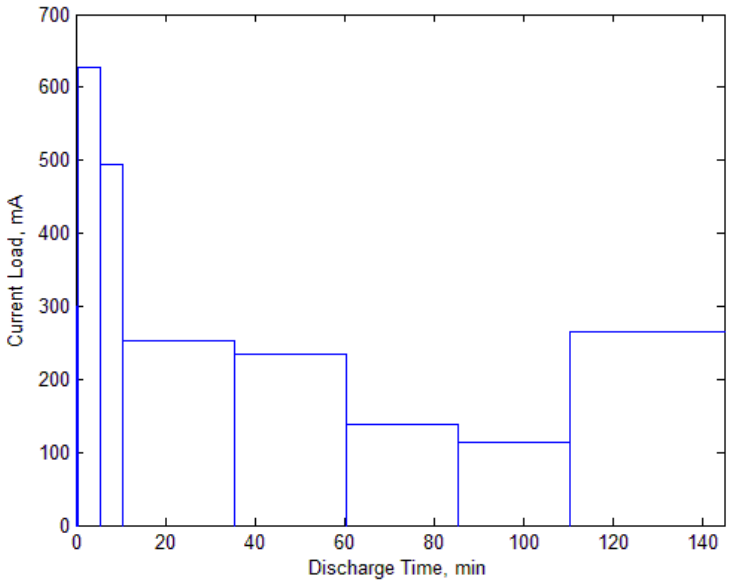


Figure 4.8 Case 3 with high peak load of 628.0mA from 0.5min to 5.5min

Figure 4.9 displays our battery voltage calculation results for Case 3, and Figure 4.10 shows the corresponding voltage curve graphically. We can see that the "death" time is now 139.9 min, which is much better than the battery lifetime in Case 2 (107.0 min). These battery lifetime estimation results for Case 2 and Case3 indicate that the battery can handle peak current load better at the beginning of a load profile, when the battery is relatively fresh. This result is in line with the conclusions in [16], asserting that load sequencing can significantly affect the battery lifetime.

```
Time   =     0 min,    Current =    300 mA,    Voltage = 3.9614 V
Time   =   0.5 min,    Current =    300 mA,    Voltage = 3.92165 V
Time   =   0.5 min,    Current =    628 mA,    Voltage = 3.79045 V
Time   =   5.5 min,    Current =    628 mA,    Voltage = 3.65017 V
Time   =   5.5 min,    Current =  494.7 mA,    Voltage = 3.70349 V
Time   =  10.5 min,    Current =  494.7 mA,    Voltage = 3.66216 V
Time   =  10.5 min,    Current =  252.3 mA,    Voltage = 3.75912 V
Time   =  30.5 min,    Current =  252.3 mA,    Voltage = 3.70447 V
Time   =  35.5 min,    Current =  234.1 mA,    Voltage = 3.72294 V
Time   =  60.5 min,    Current =  234.1 mA,    Voltage = 3.64978 V
Time   =  60.5 min,    Current =  137.9 mA,    Voltage = 3.68826 V
Time   =  85.5 min,    Current =  137.9 mA,    Voltage = 3.65868 V
Time   =  85.5 min,    Current =  113.9 mA,    Voltage = 3.66828 V
Time   = 110.5 min,    Current =  113.9 mA,    Voltage = 3.63886 V
Time   = 110.5 min,    Current =  265.6 mA,    Voltage = 3.57818 V

Battery failure detected at time interval from 110.5 to 145.5 min
Current load = 265.6 mA, Vcutoff = 3.4 V

Using fine-grain timing accuracy of 0.1 min
Battery dies at  139.9 min, Battery voltage = 3.40081 V
```

Figure 4.9 Battery lifetime in Case 3

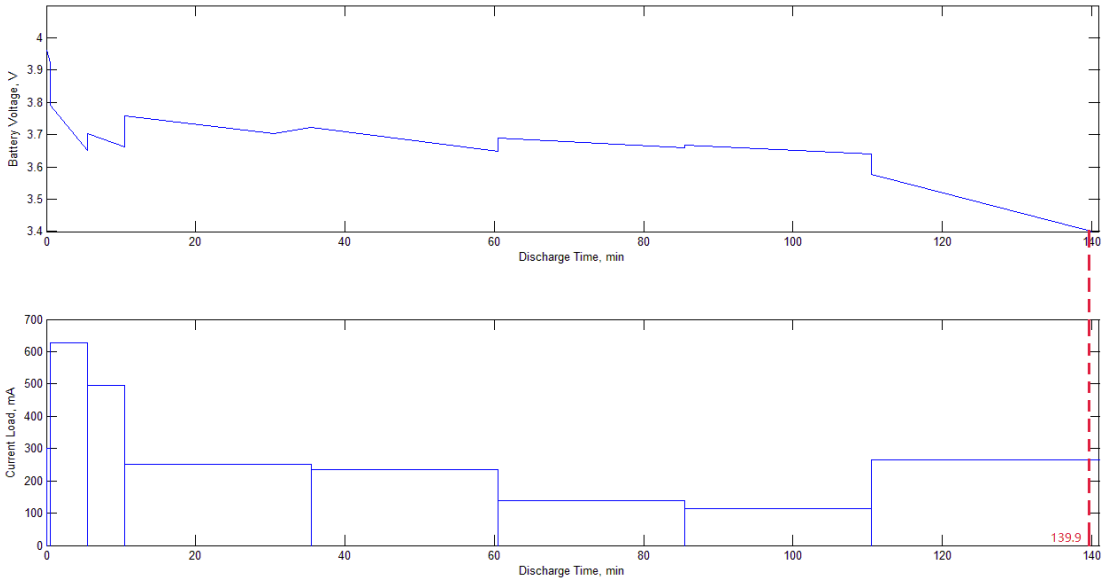


Figure 4.10 Battery voltage in Case 3

Would it be beneficial to convert a non-periodic load into a periodic loads series, i.e. "spread out" the heavy loads across the profile? To answer this question, we created another load profile, designated here as Case 4 and shown in Figure 4.11. Essentially, we break down the current loads from Case 2 into 5 cycles, only keeping the first load and the last load unaltered.

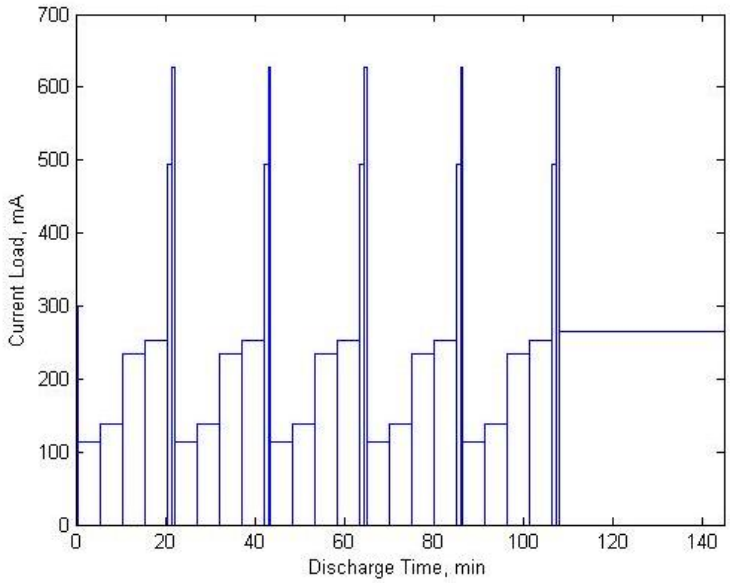


Figure 4.11 Case 4 current loads spread out

Figure 4.12 displays our battery voltage calculation results for Case 4, and Figure 4.13 shows the corresponding voltage curve graphically. We can see that the "death" time is

138.6 min, which is much better than the battery lifetime in Case 2 (107.0 min), and slightly worse than the lifetime in Case 3 (139.9 min).

```
Time    =    86.5 min,    Current =     628 mA,    Voltage = 3.46134 V
Time    =    86.5 min,    Current =   113.9 mA,    Voltage = 3.66698 V
Time    =    91.5 min,    Current =   113.9 mA,    Voltage = 3.68221 V
Time    =    91.5 min,    Current =   137.9 mA,    Voltage = 3.67261 V
Time    =    96.5 min,    Current =   137.9 mA,    Voltage = 3.66575 V
Time    =    96.5 min,    Current =   234.1 mA,    Voltage = 3.62727 V
Time    =   101.5 min,    Current =   234.1 mA,    Voltage = 3.6087 V
Time    =   101.5 min,    Current =   252.3 mA,    Voltage = 3.60142 V
Time    =   106.5 min,    Current =   252.3 mA,    Voltage = 3.58477 V
Time    =   106.5 min,    Current =   494.7 mA,    Voltage = 3.48781 V
Time    =   107.5 min,    Current =   494.7 mA,    Voltage = 3.47007 V
Time    =   107.5 min,    Current =     628 mA,    Voltage = 3.41675 V
Time    =     108 min,    Current =     628 mA,    Voltage = 3.40485 V
Time    =     108 min,    Current =   265.6 mA,    Voltage = 3.54981 V

Battery failure detected at time interval from 108 to 143 min
Current load = 265.6 mA, Vcutoff = 3.4 V

Using fine-grain timing accuracy of 0.1 min
Battery dies at  138.6 min, Battery voltage = 3.40032 V
```

Figure 4.12 Battery lifetime in Case 4

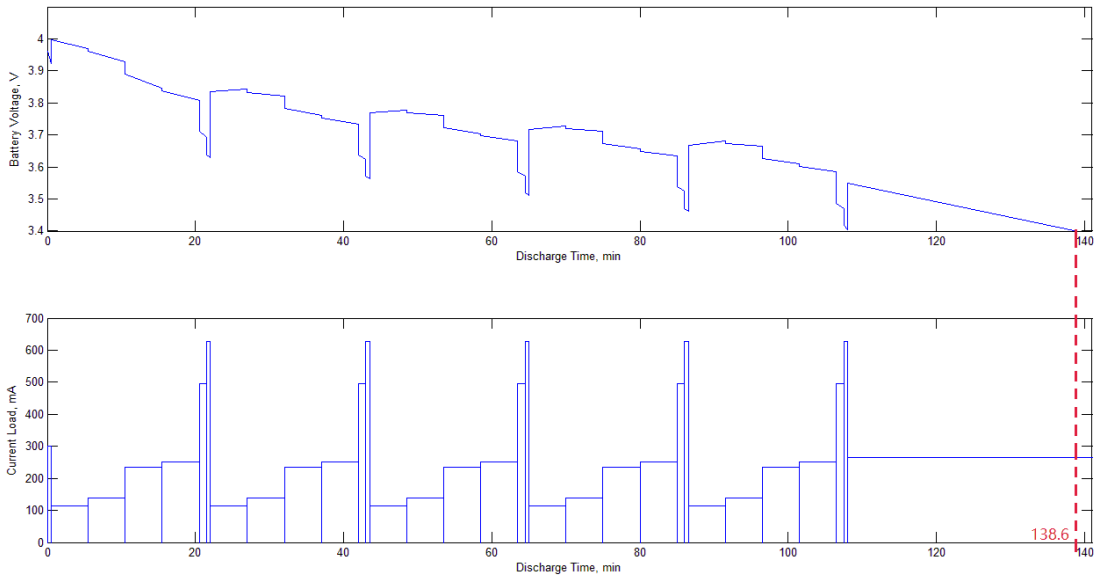


Figure 4.13 Battery voltage in Case 4

4.2.2.2 Light peak load

Figure 4.14 and Figure 4.15 show current load profiles for Case 5 and Case 6 (detailed in Appendix 2). In both cases, the peak load is only 222.7 mA, which can be viewed as a relatively light load. The only difference between these two cases is ordering of their current loads in time.

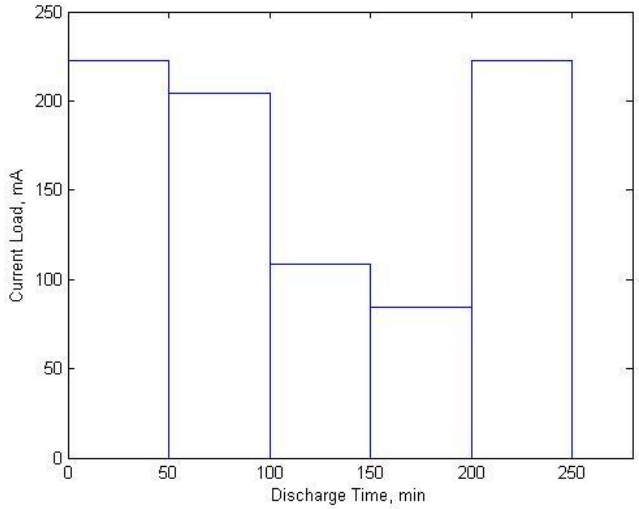


Figure 4.14 Case 5 with light peak load

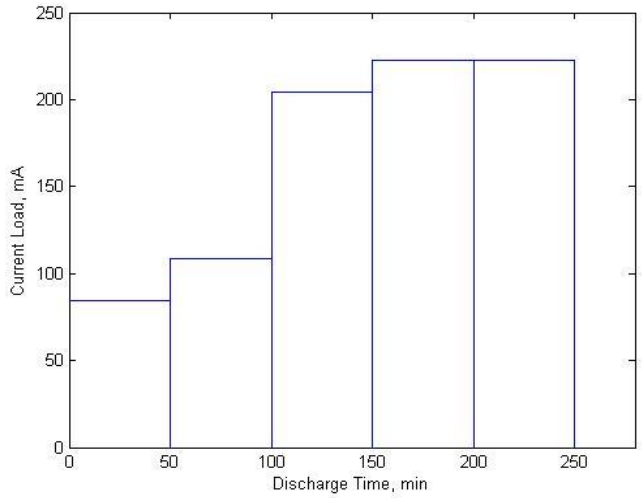


Figure 4.15 Case 6 with light peak load

Figure 4.16 and Figure 4.17 show the respective battery voltage values calculated by our battery model for Case 5 and Case 6, respectively. Figure 4.18 and Figure 4.19 show the corresponding battery voltage curves for Case 5 and Case 6, respectively. We can the estimated battery lifetime 203.9 min in Case 5 and 202.5 min in Case 6. These battery

lifetime results indicate that for relatively light peak loads, load sequence does not have significant impact, unlike the cases of heavy peak loads discussed in the previous section.

```
Time   =      0 min,    Current =   222.7 mA,     Voltage = 3.99232 V
Time   =     50 min,    Current =   222.7 mA,     Voltage = 3.71415 V
Time   =     50 min,    Current =   204.5 mA,     Voltage = 3.72143 V
Time   =    100 min,    Current =   204.5 mA,     Voltage = 3.61313 V
Time   =    100 min,    Current =   108.3 mA,     Voltage = 3.65161 V
Time   =    150 min,    Current =   108.3 mA,     Voltage = 3.59252 V
Time   =    150 min,    Current =    84.3 mA,     Voltage = 3.60212 V
Time   =    200 min,    Current =    84.3 mA,     Voltage =  3.526 V
Time   =    200 min,    Current =   222.7 mA,     Voltage = 3.47064 V

Battery failure detected at time interval from 200 to 205 min
Current load = 222.7 mA, Vcutoff = 3.4 V

Using fine-grain timing accuracy of 0.1 min
Battery dies at  203.9 min, Battery voltage = 3.40033 V
```

Figure 4.16 Battery lifetime in Case 5

```
Time   =      0 min,    Current =    84.3 mA,     Voltage = 4.04768 V
Time   =     50 min,    Current =    84.3 mA,     Voltage = 3.87557 V
Time   =     50 min,    Current =   108.3 mA,     Voltage = 3.86597 V
Time   =    100 min,    Current =   108.3 mA,     Voltage = 3.78323 V
Time   =    100 min,    Current =   204.5 mA,     Voltage = 3.74475 V
Time   =    150 min,    Current =   204.5 mA,     Voltage = 3.6298 V
Time   =    150 min,    Current =   222.7 mA,     Voltage = 3.62252 V
Time   =    200 min,    Current =   222.7 mA,     Voltage = 3.42665 V

Battery failure detected at time interval from 200 to 205 min
Current load = 222.7 mA, Vcutoff = 3.4 V

Using fine-grain timing accuracy of 0.1 min
Battery dies at  202.5 min, Battery voltage = 3.40115 V
```

Figure 4.17 Battery lifetime in Case 6

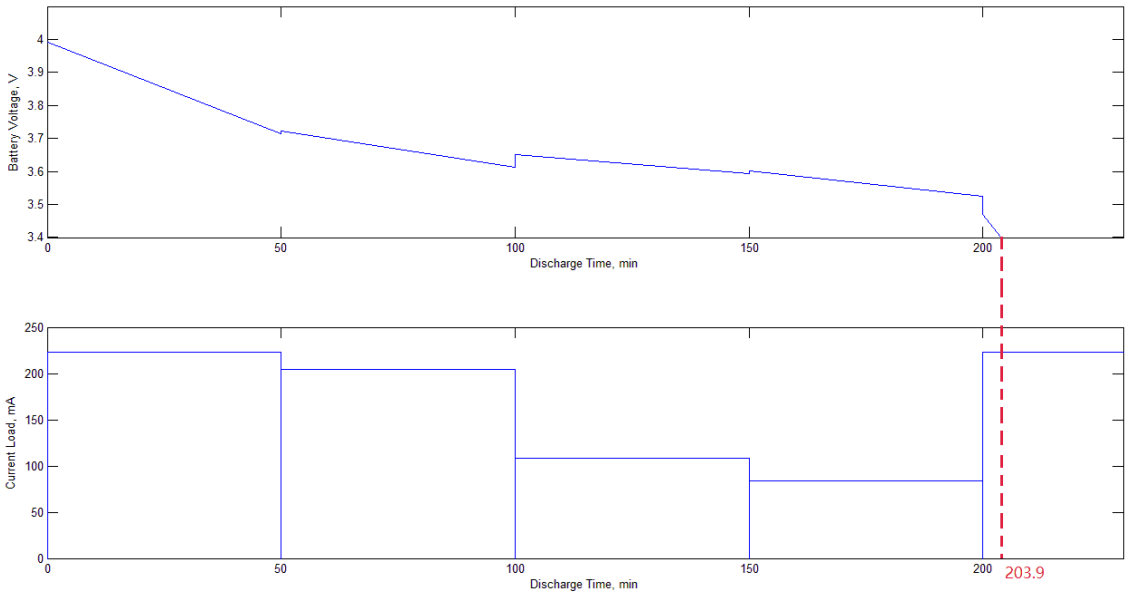


Figure 4.18 Battery voltage in Case 5

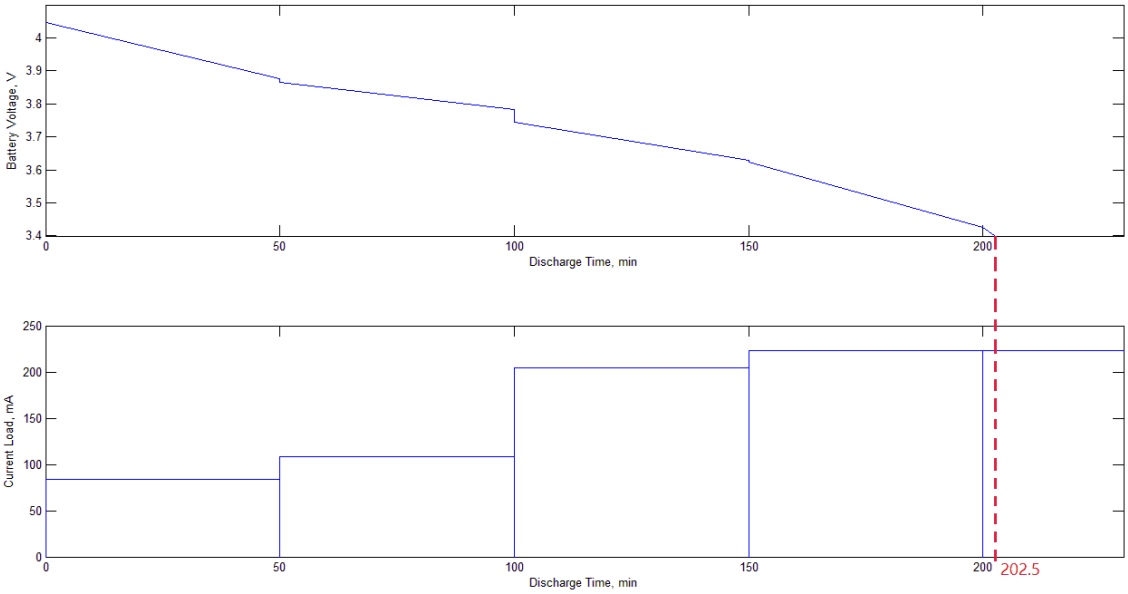


Figure 4.19 Battery voltage in Case 6

4.2.2.3 Battery voltage recovery effect

We have already observed charge recovery effect in Case 1, which has an idle period (current load equals to zero), lasting from 105.5 min to 130.5 min. Figure 4.20 shows the corresponding battery voltage curve. We notice that during the idle period the battery

voltage is increasing. This phenomenon is known as a charge recovery effect. An ESL designer can assess the impact of inserting idle periods within a load profile to keep the battery alive as long as possible.

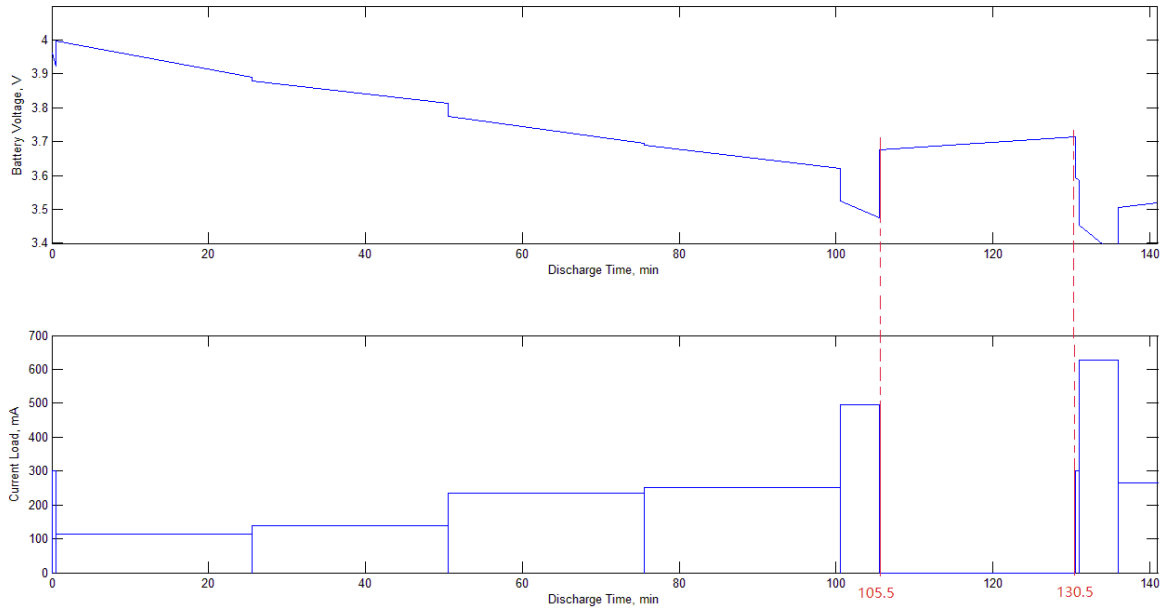


Figure 4.20 Battery voltage recovery effect in Case 1

Chapter 5

Conclusion

This project implemented a battery model from [9] in SystemC language, a C++ class library which is well-suited for describing heterogeneous system designs at various levels of abstraction. Our executable model is compatible with any existing SystemC software, thus facilitating its integration into various ESL design descriptions, to assess the impact of a given system current load profile on the battery behavior.

The model features 9 parameters capturing internal battery-specific characteristics:

- r represents the ohmic resistance (Ohm);
- V_0 represents reference voltage (Volt);
- φ determines the voltage curve flatness (Volt);
- α_n and $\alpha_p$determine initial battery capacity (Coulomb);
- β_n and $\beta_p$characterize short-term capacity losses, usually observed at high discharge rates (1/sec);
- γ_n and γ_p characterize long-term capacity losses, usually observed as the battery ages (1/sec).

There are also 4 additional implementation-specific parameters controlled by the user to achieve desired accuracy-complexity trade-offs:

- M limits the number of terms summed when evaluating S-series;
- H limits the number of previous loads (history length) considered when evaluating F-factors;
- $\mathrm{V_{cut-off}}$ is the battery voltage threshold value below which the battery is treated as "dead" (discharged);
- delta is the time precision setting used when searching for the time instance at which the battery voltage crossed the threshold value.

We have demonstrated the utility of our executable model using 6 load profiles adopted from [16]. We have illustrated the benefit of arranging a current load sequence in a decreasing order of their currents (as opposed to an increasing order) at the presence of heavy peak loads. We have also illustrated a charge recovery effect observed during zero-load time intervals, which can be utilized by an ESL designer to prolong the battery

lifetime. Last but not least, we have tested various *M* and *H* settings and their impact on the accuracy of battery voltage calculations.

Our recommendations for future research are as follows. The battery model provides the means for evaluating system's current load profiles, but further work is needed to develop methods for model-guided synthesis of current load profiles that maximize the battery lifetime under performance constraints. Further work on accuracy-complexity trade-offs can also be of practical value: for example, one can explore the possibilities and implications of automatically changing adaptive *M* and *H* settings during the battery voltage calculations, which would yield a self-calibrating executable model implementation. Finally, given that the battery voltage must be regulated by DC-DC converters, future efforts should include modelling of the power supply behavior as a whole, including both a battery and a DC-DC converter.

Bibliography

[1] Doyle, M., Fuller, T., and Newman, J., Modeling of Galvanostatic charge and Discharge of the Lithium PolymerInsertion Cell, In *Journal Electochemical Society.,* vol.14 pp.1526-1533,6 1993.

[2] Newman, J. and Thomas-Alyea, K. 2004. Electrochemical Systems, 3Ed. John Wiley&Sons, New York.

[3] Krintz, C., Wen, Y., and Wolski, R. 2004. Application-level prediction of battery dissipation. In *Proceeding of the International Symposium on Low Power Electronics and Design.* ACM/IEEE, 224-229.

[4] Szumanowski, A. and Chang, Y. 2008. Battery management system based on battery nonlinear dynamics modeling. In *IEEE Trans. Veh. Techn. 57,* 3, 1425-1432.

[5] Rao, R., Singhal, G. Kumar, A., and Navet, N. 2005 battery model for embedded systems. In *Proceedings of the International Conference on VLSI Design*. IEEE, 105-110.

[6] Dubarry, M., and Liaw, B. 2007. Development of a universal modeling tool for rechargeable lithium batteries. In *J. Power Sources 174,* 856-860.

[7] Jossen, A. 2006. Fundamentals of battery dynamics. In *J. Power Sources 154,* 530-538.

[8] Rong, R., and Pendram, M. 2006. An analytical model for predicting the remaining battery capacity of lithium-ion batteries. In *IEEE Trans. VLSI Sys. 14,* 5, 441-451.

[9] Rakhmatov, D. 2009. Battery voltage modeling for portable systems. *ACM Trans. Des. Autom. Elect. Syst., 14,* 2, Article 29 (March 2009), 36 pages.

[10] SystemC definition https://en.wikipedia.org/wiki/SystemC

[11] SystemC Modules http://www.asic-world.com/systemc/modules2.html

[12] SystemC Modules and Processes https://www.doulos.com/knowhow/systemc/tutorial/modules_and_processes/

[13] Simpson, P. A., FPGA Design: Best Practices for Team-based Reuse, 2 edition, pp.23, Springer, New York.

[14] Bailey, B., and Martin, G. 2009. ESL models and their application: Electronic System Level Design and Verification in Practice. pp.143 Springer, New York.

[15] Rogin, F.and Drechsler, R. Debugging at the Electronic System Level. Springer, New York.

[16] Rakhmatov, D., Vrudhula, S., Wallah, A. 2003. A model for battery lifetime analysis for organizing applications on a pocket computer. In *IEEE Trans. VLSI Sys. VOL 11, No.6.*

[17] Open SystemC Initiative, SystemC 2.0.1 Language Reference Manual, San Jose, CA

[18] Voltage cut-off in battery https://en.wikipedia.org/wiki/Cutoff_voltage

[19] Viredaz, M. and Wallach, D. , 2001, Power evaluation of a handheld computer: A case study, Compaq QRL Research Rep. Palo Alto, CA

Appendix 1

1. configData.dat

Type and name	Setting examples	Reference and unit
double V0;	3.75	reference voltage, Volt
double r;	0.4	ohmic resistance, Ohm
double phi;	0.09	voltge curve flatness, Volt
double alpha_n;	900	initial battery capacity negative, Coulomb
double alpha_p;	35760	initial battery capacity positive, Coulomb
double beta_n;	2.5	short term capacity loss negative, 1/Min
double beta_p;	0.29	short term capacity loss positive, 1/Min
double gamma_n;	0.0000016	long term capacity loss negative, 1/Min
double gamma_p;	0.0000016	long term capacity loss positive, 1/Min
int M;	10	number of terms
int H;*	4	number of history terms, usually use the number that makes the duration as half an hour
double Vcut;	3.4	Cutoff voltage, threshold of battery death, Volt
double delta;	0.1	time step, the accuracy of battery lifetime prediction, Volt

*Note:

H = 4 is used in Case 1, 10 current loads (see Appendix 2)

H = 4 is used in Case 2, and Case 3, 8 current loads (see Appendix 2).

H = 30 is used in Case 4, 32 current loads (see Appendix 2).

H = 3 is used in Case 5 and Case 6, 5 current loads (see Appendix 2).

Appendix 2

1. CurrentProfile_Case1.dat content

Starting time	Current load	Duration
0.00000	300.0	0.500000
.500000	113.9	25.00000
25.5000	137.9	25.00000
50.5000	234.1	25.00000
75.5000	252.3	25.00000
100.500	494.7	5.00000
105.500	0.000	25.00000
130.500	300.0	0.500000
131.000	628.0	5.00000
136.000	265.6	5.00000

2 CurrentProfile_Case2.dat content

Starting time	Current load	Duration
0.00000	300.0	0.500000
.500000	113.9	25.00000
25.5000	137.9	25.00000
50.5000	234.1	25.00000
75.5000	252.3	25.00000
100.500	494.7	5.00000
105.500	628.0	5.00000
110.5	265.6	35.00000

3 CurrentProfile_Case3.dat content

Starting time	Current load	Duration
0.00000	300.0	0.500000
.500000	628.0	5.00000
5.5000	494.7	5.00000
10.5000	252.3	20.00000
35.5000	234.1	25.00000
60.500	137.9	25.00000
85.500	113.9	25.00000
110.5	265.6	35.00000

4 CurrentProfile_Cas4.dat content

0.00000	300.0	0.50000
0.50000	113.9	5.00000
5.50000	137.9	5.00000
10.5000	234.1	5.00000
15.5000	252.3	5.00000
20.5000	494.7	1.00000
21.5000	628.0	0.50000
22.0000	113.9	5.00000
27.0000	137.9	5.00000
32.0000	234.1	5.00000
37.0000	252.3	5.00000
42.0000	494.7	1.00000
43.0000	628.0	0.50000
43.5000	113.9	5.00000
48.5000	137.9	5.00000
53.5000	234.1	5.00000
58.5000	252.3	5.00000
63.5000	494.7	1.00000
64.5000	628.0	0.50000
65.0000	113.9	5.00000
70.0000	137.9	5.00000
75.0000	234.1	5.00000
80.0000	252.3	5.00000
85.0000	494.7	1.00000
86.0000	628.0	0.50000
86.5000	113.9	5.00000
91.5000	137.9	5.00000
96.5000	234.1	5.00000
101.500	252.3	5.00000
106.500	494.7	1.00000
107.500	628.0	0.50000
108.000	265.6	35.0000

5 CurrentProfile_Case5.dat content

Starting time	Current load	Duration
0.00000	222.7	50.0000
50.0000	204.5	50.0000
100.000	108.3	50.0000
150.000	84.3	50.0000
200.000	222.7	50.0000

6 CurrentProfile_Case6.dat content

Starting time	Current load	Duration
0.00000	84.3	50.0000
50.0000	108.3	50.0000
100.000	204.5	50.0000
150.000	222.7	50.0000
200.000	222.7	50.0000

Appendix 3

1 "battery_estimate.h" (linked-list data structure).

```
#ifndef battery_estimate_H
#define battery_estimate_H
#include <math.h>
#include <stdlib.h>
//////////////////////////////////////////////////////////////////////
//construct double-link list data structure from current load profile
//////////////////////////////////////////////////////////////////////
//define the field "Step"
typedef struct {
	double startTime;		/* tj*/
	double currentLoad;		/* Ij */
	double loadDuration;		/* dj*/
} Step;
//define the node with pointer to previous and next node
typedef struct entry {
	Step *step;			/* entry=step*/
	struct entry *next, *prev;	/* next and previous entry */
} Entry;
#endif
```

2. “Stimulus.h” (Stimulus module).

```
#include "systemc.h"
#include "battery_estimate.h"

class stimulus : public sc_module {
public:
	sc_out <Entry* > entry_out; // pointer to the last entry
	sc_out <Entry* > head_out;  // pointer to the head entry
	sc_out <double> t;          // the time stamp of "now"
	sc_out <double> Ij;         // j_th current load

	void do_write_out();
	Entry* read_load_profile();

	SC_HAS_PROCESS(stimulus);
	stimulus(sc_module_name nm, std::string fnm) : sc_module(nm),
_filename(fnm) {
		_head = read_load_profile();    // read current load profile
and get the head of linked list
		SC_THREAD(do_write_out);        // execute the process only
once
	}
private:
	Entry* _head;
	std::string _filename;
};
```

3 “stimulus.cpp” (Stimulus module).

```
#include "systemc.h"
#include "stimulus.h"
#include <stdlib.h>
```

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <sstream>
using namespace std;
////////////////////////////////////////////////
//related data structure construction functions
////////////////////////////////////////////////
Step *createStep(double start, double current, double duration) {
	Step *temp;
	if ((temp = (Step*)malloc(sizeof(*temp))) == NULL) {
		printf("\n\n*** ERROR: fail allocating memory...\n\n");
		return NULL;
	}
	temp->startTime = start;
	temp->currentLoad = current;
	temp->loadDuration = duration;
	return temp;
}
/************************/
Entry *createEntry(Step *step) {
	Entry *temp;
	if ((temp = (Entry*)malloc(sizeof(*temp))) == NULL) {
		printf("\n\n*** ERROR: fail allocating memory...\n\n");
		return NULL;
	}
	temp->step = step;
	temp->prev = NULL;
	temp->next = NULL;
	return temp;
}
/************************/
void addEntry(Entry *head, Entry *entry) {
	Entry *temp;
	temp = head->prev;
	if (temp == NULL) temp = head;
	temp->next = entry;
	entry->prev = temp;
	entry->next = head;
	head->prev = entry;
}
////////////////////////////////////////////////////////////////
//open the data profile and then output the structure pointer
////////////////////////////////////////////////////////////////
Entry* stimulus::read_load_profile()
{
	/*** Open file containing  current profile ***/
	double start, current, duration;
	Step* step;
	Entry *head, *entry;
	/*** Read current profile and create step list ***/
	if ((head = (Entry*)malloc(sizeof(*head))) == NULL) {
		printf("\n\n*** ERROR: fail creating a list head...\n\n");
		exit(0);
	}
	head->step = NULL;
	head->prev = NULL;
	head->next = NULL;

	ifstream datafile;
```

```
	datafile.open(_filename.c_str());

	/* read data profile and create a list of steps */
	std::string line;
	while (std::getline(datafile, line)) {
		std::istringstream lineStream(line);
		lineStream >> start;
		lineStream >> current;
		lineStream >> duration;

		step = createStep(start, current, duration);
		if (step == NULL) {
			printf("\n\n*** ERROR: fail creating a step
structure...\n\n");
			exit(0);
		}
		// create entry
		entry = createEntry(step);
		if (entry == NULL) {
			printf("\n\n*** ERROR: fail creating a list
entry...\n\n");
			exit(0);
		}
		addEntry(head, entry);
	}

	datafile.close();
	return head;
}
////////////////////////////////////////////////////////////////////////
//Write the output port with a series of Ij, entry, head, and t
////////////////////////////////////////////////////////////////////////
void stimulus::do_write_out()
{
	Entry* entry;
	double start_time, execution_time;

	//write port t twice for each interval
	for (entry = _head->next; entry != _head; entry = entry->next)
	{
		head_out.write(_head);
		entry_out.write(entry);
		start_time = entry->step->startTime;
		execution_time = entry->step->loadDuration;
		Ij.write(entry->step->currentLoad);
		//write t with the starting time of interval
		t.write(start_time);
		//wait for duration time, to write t second time
		wait(execution_time * 60, SC_SEC);
		//keep the values on ports Ij, head_out, and entry_out
		//write t with the end time of interval
		t.write(start_time + execution_time);
		//wait 0 second, be ready to write t with the starting time of
next interval
		wait(0, SC_SEC);
	}
}
```

4.”compute.h” (Compute module).

```
#include "systemc.h"
#include "battery_estimate.h"
#include <stdlib.h>
////////////////////////////////////////////////////////////////////////
//package the kernel of battery voltage computation
////////////////////////////////////////////////////////////////////////
SC_MODULE(battery_voltage) {

	sc_in<Entry* > entry_in;
	sc_in<Entry* > head_in;
	sc_in<double>  t;
	sc_in<double>  Ij;

	sc_out<double> t_out;
	sc_out<double> Ij_out;
	sc_out<double> Vjt;

	void Voltage_Computation();
	void read_configData();
	double Fnj(double, Entry *);
	double Fpj(double, Entry *);
	double cal_IkFnk(Entry *, Entry *, double);
	double cal_IkFpk(Entry *, Entry *, double);

	SC_HAS_PROCESS(battery_voltage);
	battery_voltage(sc_module_name nm, std::string fnm, std::string
sfnm) : sc_module(nm), _filename(fnm), _save_file_name(sfnm){
		SC_METHOD(Voltage_Computation);
		dont_initialize();    // don't call computation process at time
0;
		sensitive << t << Ij; // execute when t or Ij is changed;

		read_configData();    // read the parameter file;

		savefile = fopen("resultfile.dat", "w");//open a file to be
written;
		_Cn = 0;              //coefficient Cn for approximation
calculation
		_Cp = 0;              //coefficient Cp for approximation
calculation
	}

private:
	double _Cn, _Cp;
	std::string _filename;
	std::string _save_file_name;
	FILE* savefile;
};
```

5 “compute.cpp” (Compute module).

```
#include "systemc.h"
#include "compute.h"
#include <math.h>
#include <vector>
#include<iomanip>
#include <chrono>
```

```
#include <ctime>
using namespace std;
////////////////////////////////////////////////////////////////////////
//define the 13 parameters of voltage model
double V0;      // reference voltage, Volt
double r;       // ohm resistance, Ohm
double phi;     // voltage curve flatness, Volt
double alpha_n; // initial battery capacity negative, Coulomb
double alpha_p; // initial battery capacity positive, Coulomb
double beta_n;  // short term capacity loss negative, 1/Min
double beta_p;  // short term capacity loss positive, 1/Min
double gamma_n; // long term capacity loss negative, 1/Min
double gamma_p; // long term capacity loss positive, 1/Min
int M;          // number of terms
int H ;         // number of history terms, usually use the number that
makes the duration as half an hour
double Vcut; // Cutoff voltage, threshold of battery death
double delta;// time step, when checking at which timestamp the battery die
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////

/*** Open configuration file containing 13 parameters ***/
void  battery_voltage::read_configData()
{
      FILE* configData;
      configData = fopen(_filename.c_str(), "r");
      /*** Read battery profile ***/
      fscanf(configData, "%lf\n", &V0);
      fscanf(configData, "%lf\n", &r);
      fscanf(configData, "%lf\n", &phi);
      fscanf(configData, "%lf\n", &alpha_n);
      fscanf(configData, "%lf\n", &alpha_p);
      fscanf(configData, "%lf\n", &beta_n);
      fscanf(configData, "%lf\n", &beta_p);
      fscanf(configData, "%lf\n", &gamma_n);
      fscanf(configData, "%lf\n", &gamma_p);
      fscanf(configData, "%d\n", &M);
      fscanf(configData, "%d\n", &H);
      fscanf(configData, "%lf\n", &Vcut);
      fscanf(configData, "%lf\n", &delta);
}
////////////////////////////////////////////////////////////////////////
// related function expression
// current load related computation functions
//F_nj=(e^(-γ_n*t_j )-e^(-γ_n*t))/γ_n+S_nj
//F_pj=(e^(γ_p*t)-e^(γ_p*t_j ))/γ_p +S_pj
// present entry in data structure is needed as argument
// time stamp is as the other argument
////////////////////////////////////////////////////////////////////////
double battery_voltage::Fnj(double t, Entry *entry)
{
      double Fnj_temp = 0;
      double Snj_temp = 0;
      for (int m = 1; m <= M; m++) {
            Snj_temp += (1 - exp(-(beta_n*m*m - gamma_n)*(t - entry->step-
>startTime))) / (beta_n*m*m - gamma_n);
      }
      Snj_temp = 2 * exp(-gamma_n*t)*Snj_temp;
      Fnj_temp = (exp(-gamma_n*(entry->step->startTime)) - exp(-gamma_n*t))
/ gamma_n + Snj_temp;
```

```
	return Fnj_temp;
}

double battery_voltage::Fpj(double t, Entry *entry)
{
	double Fpj_temp = 0;
	double Spj_temp = 0;
	for (int m = 1; m <= M; m++) {
		Spj_temp += (1 - exp(-(beta_p*m*m + gamma_p)*(t - entry->step->startTime))) / (beta_p*m*m + gamma_p);
	}
	Spj_temp = 2 * exp(gamma_p*t) * Spj_temp;
	Fpj_temp = (exp(gamma_p*t) - exp(gamma_p*entry->step->startTime)) / gamma_p + Spj_temp;
	return Fpj_temp;
}
////////////////////////////////////////////////////////////////////////////
//history load related computation functions
////////////////////////////////////////////////////////////////////////////
double battery_voltage::cal_IkFnk(Entry *head, Entry *last, double t)
{
	double sum_IkFnk, Snk_temp, Fnk;
	double start, current, duration;
	int m;
	Entry *entry = NULL;		// current entry
	Entry *entryH = NULL;		// history entry
	int h = 0;			// counter for number of calculated terms
	bool flag = false;		// flag shows that the calculation backwards is not reach the very beginning of data structure
	sum_IkFnk = 0;
	for (entry = last->prev; entry != head; entry = entry->prev) {
		start = entry->step->startTime;
		current = entry->step->currentLoad;
		duration = entry->step->loadDuration;
		Snk_temp = 0;

		// compute Snk
		for (m = 1; m <= M; m++) {
			Snk_temp += (exp(-gamma_n*(start + duration))*exp(-beta_n*m*m*(t - start - duration)) - exp(-gamma_n*start)*exp(-beta_n*m*m*(t - start))) / (beta_n*m*m - gamma_n);

		}
		Snk_temp = 2 * Snk_temp;
		// compute Fnk
		Fnk = (exp(-gamma_n*start) - exp(-gamma_n*(start + duration))) / gamma_n + Snk_temp;
		// update the sum of IkFnk (e.g. compute the H history)
		sum_IkFnk += current* Fnk;
		// update the loads that have counted as history
		h++;
		// stop when H loads have been counted
		if (h == H) {
			entryH = entry->prev;
			if (entryH != head)  flag = true;
			break;
		}
	}
	// add correcting term (_Cn) to obtain the approximate sum
	//calculate the Cn(j-H-1)=I(j-H-1)Fn(j-H-1)  then
```

```
	if (flag == true){
		start = entryH->step->startTime;
		current = entryH->step->currentLoad;
		duration = entryH->step->loadDuration;
		Snk_temp = 0;

		for (m = 1; m <= M; m++) {
			Snk_temp += (exp(-gamma_n*(start + duration))*exp(-
beta_n*m*m*(t - start - duration)) - exp(-gamma_n*start)*exp(-beta_n*m*m*(t
- start))) / (beta_n*m*m - gamma_n);

		}
		Snk_temp = 2 * Snk_temp;
		Fnk = (exp(-gamma_n*start) - exp(-gamma_n*(start + duration)))
/ gamma_n + Snk_temp;

		_Cn += current*Fnk;

		sum_IkFnk += _Cn;
	}

	return sum_IkFnk;
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
double battery_voltage::cal_IkFpk(Entry *head, Entry *last, double t)
{
	double sum_IkFpk, Spk_temp, Fpk;
	double start, current, duration;
	int m;
	Entry *entry = NULL;
	Entry *entryH = NULL;
	int h = 0;
	bool flag = false;
	sum_IkFpk = 0;
	for (entry = last->prev; entry != head; entry = entry->prev) {

			start = entry->step->startTime;
			current = entry->step->currentLoad;
			duration = entry->step->loadDuration;
			Spk_temp = 0;

			for (m = 1; m <= M; m++) {
				Spk_temp += (exp(gamma_p*(start + duration))*exp(-
beta_p*m*m*(t - start - duration)) - exp(gamma_p*start)*exp(-beta_p*m*m*(t
- start))) / (beta_p*m*m + gamma_p);
			}
			Spk_temp = 2 * Spk_temp;
			Fpk = (exp(gamma_p*(start + duration)) -
exp(gamma_p*start)) / gamma_p + Spk_temp;
			sum_IkFpk += current* Fpk;
			h++;
			if (h == H) {
				entryH = entry->prev;
				if (entryH != head)  flag = true;
				break;
			}
	}
	//calculate the Cp(j-H-1)=I(j-H-1)Fp(j-H-1)
	if (flag == true){
```

```
		start = entryH->step->startTime;
		current = entryH->step->currentLoad;
		duration = entryH->step->loadDuration;
		Spk_temp = 0;

		for (m = 1; m <= M; m++) {
			Spk_temp += (exp(gamma_p*(start + duration))*exp(-
beta_p*m*m*(t - start - duration)) - exp(gamma_p*start)*exp(-beta_p*m*m*(t
- start))) / (beta_p*m*m + gamma_p);

		}
		Spk_temp = 2 * Spk_temp;
		Fpk = (exp(gamma_p*(start + duration)) - exp(gamma_p*start)) /
gamma_p + Spk_temp;
		_Cp += current*Fpk;
		sum_IkFpk += _Cp;
	}
		return sum_IkFpk;
}
////////////////////////////////////////////////////////////////////////////
// computation kernal
////////////////////////////////////////////////////////////////////////////
void battery_voltage::Voltage_Computation() {

	double result_temp = 0;
	double result_temp_buf = 0;
	double right_now = 0;
	// read relevant ports
	double now = t.read();				// read the time
stamp port
	double Ij_buf = Ij.read();			// read current load
	Entry* head = head_in.read();			// read head port
	Entry* entry = entry_in.read();			// read entry port
	////////////////////////////////////////////////
	// declare a timer
	std::clock_t timer_start;
	double timer_duration;
	timer_start = std::clock();
	////////////////////////////////////////////////
	//call related functions
	double Fnj_value = Fnj(now, entry);			// call
function to calculate Fnj
	double Fpj_value = Fpj(now, entry);		// call function to
calculate Fpj
	//double Ij = entry->step->currentLoad;		// Ij is the jth
workload
	double sum_IkFnk = cal_IkFnk(head, entry, now); // head->prev refers to the
last step
	double sum_IkFpk = cal_IkFpk(head, entry, now);
	double numerator = alpha_n + Ij_buf*Fnj_value + sum_IkFnk;
	double denominator = alpha_p - Ij_buf*Fpj_value - sum_IkFpk;
	////////////////////////////////////////////////
	//count the timer and print out
	timer_duration = ((std::clock() - timer_start) / (double)CLOCKS_PER_SEC) *
1000;
	cout << "printf: time = " << timer_duration << '\n';
	////////////////////////////////////////////////
	if (denominator > 0) {
		result_temp = V0 - r*Ij_buf / 1000 - phi*((gamma_n +
```

```
gamma_p)*now + log(numerator / denominator));
	}
	else {
		cout << "Out-of-bounds denominator detected at time " << now <<
" min (load current = " << Ij_buf << ")\n\n";
		sc_stop();
	}
	////////////////////////////////////////////////////
	//count the timer and print out
	std::clock_t c_end = std::clock();
	auto t_end = std::chrono::high_resolution_clock::now();

	std::cout << std::fixed << std::setprecision(2) << "CPU time used: "
		<< 1000.0 * (c_end - c_start) / CLOCKS_PER_SEC << " ms\n"
		<< "Wall clock time passed: "
		<< std::chrono::duration<double, std::milli>(t_end -
t_start).count()
		<< " ms\n";
	////////////////////////////////////////////////////
	t_out.write(now);
	Ij_out.write(Ij_buf);
	Vjt.write(result_temp);

	////////////////////////////////////////////////////////
	//battery discharge time stamp searching
	////////////////////////////////////////////////////////
	if (result_temp < Vcut) {
		cout << "\nBattery failure detected at time interval from "
<<entry->step->startTime<<" to "<< now << " min\n";
		cout << "Current load = " << Ij_buf << " mA, Vcutoff = " <<
Vcut << " V\n\n";
		// for now - delta, - delta, ... until start_time, rec-
calculate Vj until detecting > Vcut;
		// Print out cut-off time
		for (right_now = now - delta; right_now > entry->step-
>startTime; right_now -= delta)
		{
			double numerator_new = alpha_n + Ij_buf*Fnj(right_now,
entry) + cal_IkFnk(head, entry, right_now);
			double denominator_new = alpha_p - Ij_buf*Fpj(right_now,
entry) - cal_IkFpk(head, entry, right_now);
			if (denominator_new > 0) {
				result_temp_buf = V0 - r*Ij_buf / 1000 -
phi*((gamma_n + gamma_p)*right_now + log(numerator_new / denominator_new));
				if (result_temp_buf >= Vcut)
				//supposed to find a point that makes v bigger
than Vcut at current Ij;
				{
					cout << "Using fine-grain timing accuracy of
" << delta << " min\n";
					cout << "Battery dies at  " << right_now <<
" min, Battery voltage = " << result_temp_buf << " V\n\n";
				fprintf(savefile, "%lf %lf %lf\n", right_now,
Ij_buf, result_temp_buf);
					break;
				}
			}
			else {
				cout << "Out-of-bounds denominator detected at
time " << now << " min (load current = " << Ij_buf << ")\n\n";
```

```
                        sc_stop();
                        }
            }
            // if we could not find the time during the traversal
            if (result_temp_buf < Vcut)
                  // battery must die at the starting time of interval
            {
                  cout << "Battery dies at the start of " << entry->step-
>startTime << " min,  Battery voltage = " << result_temp << " V\n\n";
            }

            //when battery voltage crosses the Vcut-off, the system will
shut down, the calculation after this point is meaningless
            //stop the simulation at this time.
            sc_stop();
      }
////////////////////////////////////
//save file at last
////////////////////////////////////
      fprintf(savefile, "%lf %lf %lf\n", now, Ij_buf, result_temp);
}
```

6. “monitor.h” (Monitor module).

```
#include "systemc.h"
#include<iomanip>
using namespace std;

SC_MODULE(mon) {
      sc_in <double> Ij;
      sc_in <double> t;
      sc_in <double> Vjt;

      void monitor() {
            cout << "Time    = " << setw(6) << t.read() << " min,    Current
= " << setw(7) << Ij.read() << " mA,     Voltage = " << setw(6) << Vjt.read()
<< " V\n";
      }
      SC_CTOR(mon)
      {
            SC_METHOD(monitor);
            dont_initialize();
            sensitive << Vjt << t << Ij;
      }
};
```

7 “main.cpp” (main program).

```
#include "systemc.h"
#include "compute.h"
#include "stimulus.h"
#include "battery_estimate.h"
#include "monitor.h"

int sc_main(int argc, char* argv[]) {// give access to argc and argv //
from outside of sc_main
      // Elaboration, to create signals to tie modules
      sc_signal<double> t;            // time stamp t
      sc_signal<double> Ij;           // current load at t
```

```
        sc_signal<Entry* > entry_signal;// pointer to the last entry
        sc_signal<Entry* > head_signal; // pointer to the first entry
        sc_signal<double> t_out;        // time stamp t for monitor module
        sc_signal<double> Ij_out;       // current load for monitor module
        sc_signal<double> Vjt;          // battery voltage result
        ////////////////////////////////////////////////////////////
        //instantiation of stimulus, computation and monitor modules
        ////////////////////////////////////////////////////////////
        stimulus TB("battery_module_stimulus", "currentProfile_Case3.dat");
        TB.entry_out(entry_signal); TB.head_out(head_signal); TB.Ij(Ij);
TB.t(t);

        battery_voltage DUT("battery_module", "configData.dat",
"resultfile.dat");
        DUT.entry_in(entry_signal); DUT.head_in(head_signal); DUT.t(t);
DUT.Ij(Ij); DUT.t_out(t_out); DUT.Ij_out(Ij_out); DUT.Vjt(Vjt);

        mon Monitor("Monitor");
        Monitor.t(t_out); Monitor.Ij(Ij_out); Monitor.Vjt(Vjt);
        // Start the main simulation thread
        sc_start();
        cin.get();//when debugging, waiting keyboard input to keep window display
        return 0;
}
```