Energy Management Project Report

Supercapacitor modelling in SystemC-AMS

Bálint Bujtor

Energy Management for IoT

Politecnico di Torino

2024

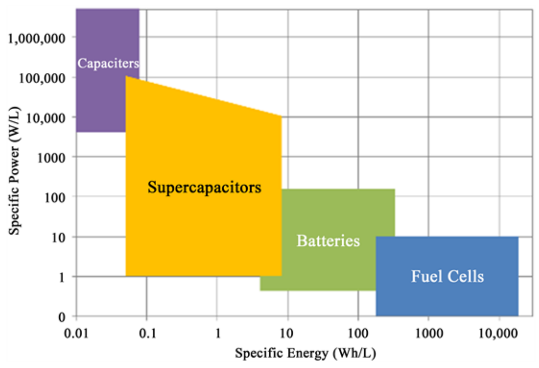

Figure 1: Ragone plot, source: [1]

1 Introduction

Energy management in resource-constrained environments has recently become a focus of the industry. The notion of performing more with less has created noticeable interest among researchers. One of the ways to achieve better power management is to diversify the system's energy storage. Besides conventional batteries novel system designs have introduced other energy storage devices such as fuel cells or supercapacitors. These devices are better suited for workloads where the system requires a short but significant amount of energy, unlike a battery [1].

The project goal is to model a supercapacitor in *SystemC* and integrate the created model into the energy simulator used during the third laboratory of the *Energy Management for IoT* course. First, the concept of supercapacitors will be presented, followed by the chosen supercapacitor model, its modelling in *SystemC*, the integration and the results.

2 Background

Electrochemical energy storage devices occupy the vast majority of the IoT market. Among these, traditional batteries are the most popular choice. However, conventional batteries along with their numerous advantages have disadvantages. Due to their intrinsic nature 1, they are not well suited to provide power for 'spike' like consumer current requests.

2.1 Supercapacitor

Supercapacitors offer a unique tradeoff between energy storage and power storage. They are considered to be between capacitors and batteries in terms of properties. They can provide high, over 100 Amper currents quasi-immediately, similar to capacitors. Moreover, they are charged and discharged rapidly and have over 100,000 charge cycles [2]. They also offer high energy density, similar to but inferior to batteries.

These properties allow supercapacitors to be used in applications that require high power only for a short time, followed by a longer idle period when the supercapacitor can be recharged.

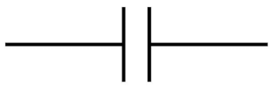

Figure 2: Single capacitance model

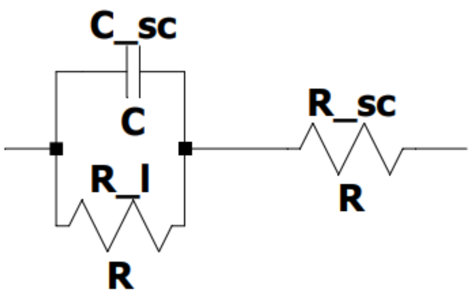

Figure 3: RC model of supercapacitance

2.2 Supercapacitor models

An appropriate model can bring many advantages to an engineer's work as it shall provide accurate behaviour of the modelled phenomenon. It shall also simplify the original formula to ease the work with the model compared to the original formula. Thus, finding an appropriate model is of great importance.

Several different supercapacitor models are used in the literature [2], [3]. The scope of this project is not to evaluate these models but only to choose the most appropriate for the needs of this project. Hence, the different possibilities will only be presented briefly.

Single capacitance

The simplest way to model a supercapacitor is with a single capacitance visible in Figure 2. However, this simple model cannot represent that the supercapacitor loses its charge over time when it is left idle. For this reason, this model is rarely used.

RC model

A more sophisticated supercapacitor model states that such a device can be accurately described with 3 discrete components. It includes C_{sc}, the supercapacitor's capacitance which is in parallel with the R_f leakage resistance. The third element connects to the previous two in series and is called series resistance R_{sc} or equivalent series resistance ESR. The model is depicted in Figure 3.

Some in the area [3] simplify the previous model by neglecting the leakage resistance $R_f = \infty$, thus obtaining a tad simpler model.

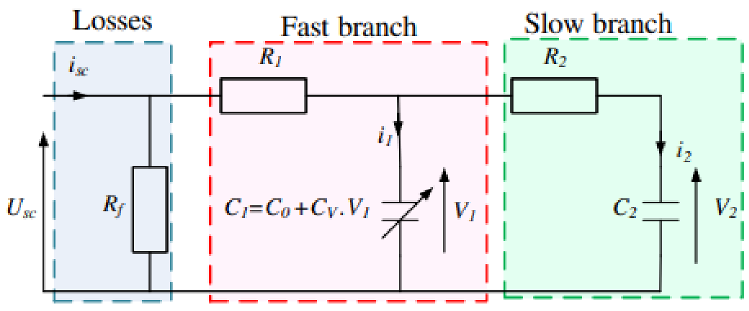

Figure 4: Two branch model of supercapacitance. Source: [2]

Two-branch model

Another choice is to model a supercapacitor with the two-branch model [2], seen in Figure 4, where a fast branch and a slow branch more precisely represent the charge and discharge characteristics of the supercapacitor.

Other models

Numerous other models exist to explain supercapacitors better, such as the multi-branch supercapacitor model [2]. Nonetheless, these models are not in the scope of this report.

2.3 Chosen supercapacitor model

The panorama of models showcased in 2.2 offers a vast choice to engineers. Selecting the proper model requires careful evaluation of the model's complexity against the required precision for the given application.

The scope of this project is not to have the best possible representation of a supercapacitor but to broaden the choices available in the third laboratory and other applications, hence, precise behaviour is not the most important aspect. Furthermore, many supercapacitor vendors only detail a few important parameters on their datasheets, preventing the engineers from inferring every parameter for a complex model, such as the multi-branch model. Datasheets mostly only describe the supercapacitor's capacitance, the maximum voltage, and the equal series resistance (ESR or R_{sc}) [5], [6].

For this reason, a suitable model for this project's purpose is the *RC model* (see 2.2), which bears the desired tradeoff between the complexity and the accuracy of the behaviour. As datasheets include information about the supercapacitors capacitance and ESR, the only unknown variable to be derived is the leakage resistance R_l. This can be computed using another property that is usually included in the datasheets, namely, the leakage current:

$$R_l = \frac{V}{I_{leakage}}$$

Using the data found in the datasheet [6] we get that:

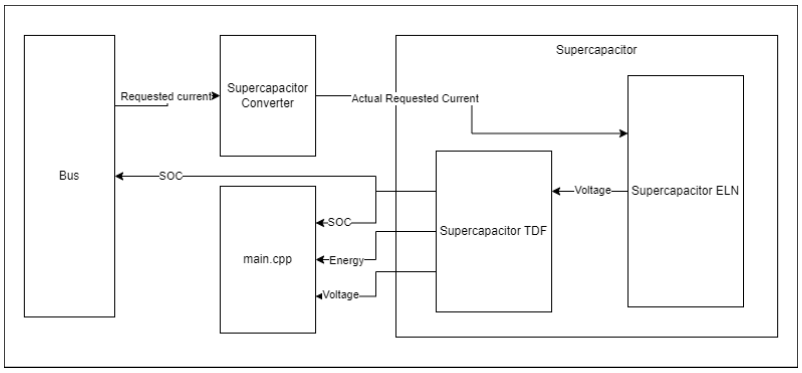

Figure 5: Supercapacitor architecture

$$R_l = \frac{V}{I_{leakage}} = \frac{3V}{7\mu A} = 428571\Omega \approx 430k\Omega$$

which is a reasonable value based on research in the area [4]. For the subsequent steps the parameters of the supercapacitor [6] will be used, having $C = 3F, V = 3V, ESR = 62m\Omega$.

With this information, everything required is available to create the selected model in *SystemC*, which will be presented in the next section.

3 Proposed solution

After explaining the theoretical background required to understand the project and choosing an appropriate supercapacitor model everything is at our disposal to implement and integrate a supercapacitor in *SystemC-AMS*, following the practices explained in [7]. As this report focuses on the supercapacitor modelling providing an introduction to *SystemC-AMS* will be omitted. For the basics, one can refer to the user guide [7].

3.1 Supercapacitor model in *SystemC*

Even though the chosen model (2.2) includes everything required to simulate it in *SystemC*, additional variables are needed to understand the supercapacitor as a power source fully. The supercapacitor has to provide the current energy level and state of charge (SOC) to suspend the simulation when this energy storage device fully depletes. Moreover, it also needs to monitor its maximum voltage to comply with the real counterpart's specifications, which can be found in the respective datasheet.

For this reason, a more complex architecture was created that complies with the requirements. The architecture's design can be seen in Figure 5. The top-level class orchestrating the operation is the *Supercapacitor* class that contains two other subclasses; its implementation can be found in 4. The *Supercapacitor ELN* class implements the chosen theoretical model using ELN components; its source code can be seen in 1. The *Supercapacitor TDF* class provides the other required metrics and controls the values of the output variables to guide the simulation; its code can be located in 2 and 3.

3.2 Supercapacitor converter

Even though, the devised supercapacitor class is complete, there are some differences between a simulation and the real-life operation of such a system. The most important is that there are always some losses in the system. To model this phenomenon a *Supercapacitor Converter* class is created that models the losses (mainly coming from non-idealities and heat dissipation). The simplest way to model these losses is to define an efficiency η that in the supercapacitor's case states how much of the provided current arrives in the system. It can be computed as: $I_{bus\_requested}/\eta = I_{actual\_requested}$, where $I_{bus\_requested}$ is the current requested from the system and $I_{actual\_requested}$ is the current that is arriving at the supercapacitor. In other words, $I_{actual\_requested}$ is how much current the supercapacitor needs to provide to fulfil the system's needs.

Moreover, another important detail that has to be dealt with is the fact that the simulation environment uses *mA*-s to operate, while the *Supercapacitor ELN* class operates in *A*-s. To solve this issue the requested current is divided by 1000 to transform it to the correct magnitude required for the *ELN* class. The implementation of the converter can be seen in 5 and in 6

3.3 Modifications to the original simulation environment

Finally, a few modifications had to be made to the original simulation environment. As the system now has two energy storage devices, it is no longer evident how the energy will be delivered to the system. To tackle this problem a simple policy has been devised, and executed by the *bus*.

To supply all the required information to the bus about the SOC of the energy storage devices has to be sent to the bus. Hence, 2 additional ports were added to the class.

The bus's policy first checks whether the current request is positive or negative; i.e. if the energy storages will be used or charged, respectively. If the energy storage devices are used, the policy checks if the supercapacitor is sufficiently charged (20%).

If so, it uses 2 variables to determine the charge distribution between the battery and the supercapacitor. The respective research states that supercapacitors are ideal to serve the needs of 'spike' like requests, e.g. requests of high currents but only for short periods. To check if a request is similar to a 'spike' we can use a simple threshold that checks if the requested current is high enough to be a possible 'spike'. If it is above this threshold it can be a 'spike'. However, the policy would need to know the length of the request to understand if it is short enough to be a 'spike'. It would need to see the future to know this, which is impossible. To address this problem, another threshold was created to prevent the policy from completely draining the supercapacitor if a long, high-current request comes. Above this second threshold, the supercapacitor is used together with the battery.

Summarizing, if the requested current is below the lower threshold the battery serves the request. If it is between the lower and the higher threshold the supercapacitor serves the request, while if it is above the higher threshold the supercapacitor and the battery serve the threshold together. The operation is visualized in 6. The lower threshold was selected based on the system's typical operation. It operates in a periodic fashion, which can be seen

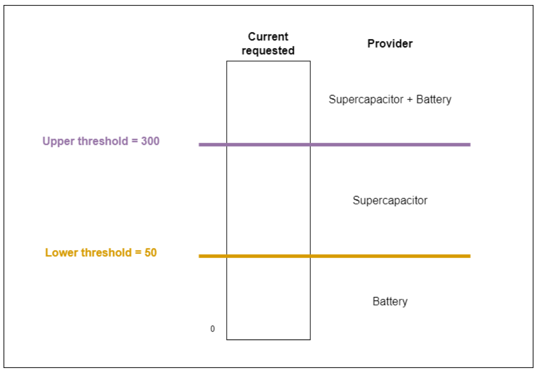

Figure 6: Bus policy

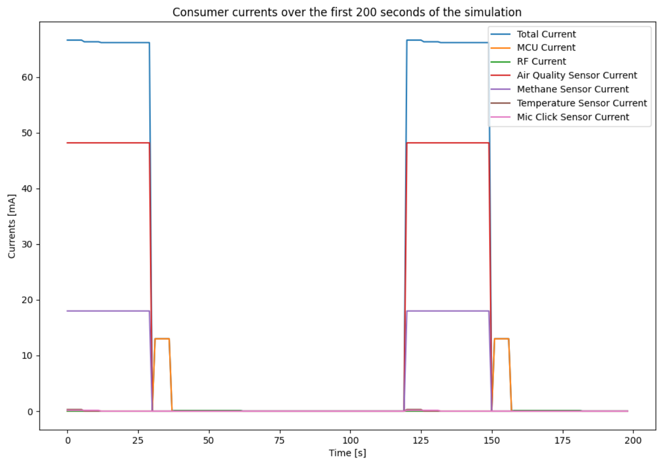

Figure 7: Periodic current request of the consumers

in 7. The aim is to use the supercapacitor during the first, highest current request. The upper limit was chosen so that the supercapacitor would not drain completely even if it had to supply a 30s request, which is typical in the simulation. The derivation can be seen in Equation 1.

$$I_{max} = \frac{V_{sc} * C_{sc}}{t_{max}} = \frac{3V * 3F}{30s} = 300mA \tag{1}$$

In case the supercapacitor's SOC is below 20% the policy only uses the battery.

The other scenario is when the current is negative, i.e. the energy storage devices are being charged. In this case, the policy first checks if the bigger device (the battery) is dying. If yes, the policy directs all the current there. Otherwise, the policy checks if the supercapacitor is not fully charged. If so, all the current is channelled to the supercapacitor. If it is fully charged the battery will be charged. This behaviour also ensures that the supercapacitor cannot be overcharged, i.e. cannot have a voltage higher than the maximum voltage specified in the datasheet. The modified code of the *bus* class can be located in 7 and 8.

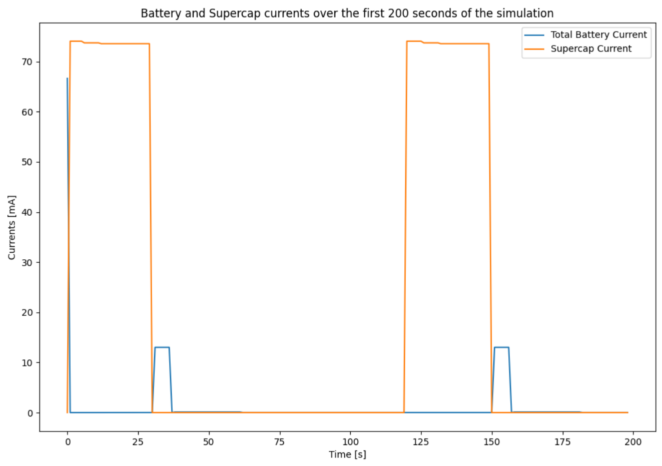

Figure 8: Battery and supercapacitor currents for 2 periods

4 Results

Upon completing every modification and extension to the simulation environment, the attention was turned to checking whether the system behaved correctly and evaluating the system's performance. In this section, the key takeaways will be communicated.

Firstly, it has to be verified if the supercapacitor's simulated behaviour matches the theoretical specifications. The formula found in [6] is used to accomplish this. Substituting the parameters with the values found in the simulation for one period of supercapacitor usage, we get:

$$C = \frac{I * (t_2 - t_1)}{V_1 - V_2} = \frac{74.058mA * (11-0)s + 73.563mA * (29-11)s}{3V - 2.283V} = 2.983F \tag{2}$$

which is almost equal to the theoretical value, verifying the correct behaviour of the ELN model (the difference stems from the not exactly precise values in the equation).

The next step is to check the behaviour of the supercapacitor and the system during the simulation. The parallel activation of the sensors is used during the simulation.

First of all, it has to be verified that the supercapacitor is being used during the time-frames when it was intended to be used.

We can see by looking at 8 and 9 that the supercapacitor supplies the system only during the peaks, otherwise, the battery is used. This only changes when the supercapacitor discharges below 10%. We can also see looking at 9 that once the PV panel starts supplying charge, it goes to the supercapacitor which allows it to start supplying energy again, even if only for short periods. It has to be stated that the higher peak current of the supercapacitor is due to the non-idealities (the efficiency η of the supercapacitor and the converter).

It is also educative to look at the charging and depletion characteristics of the supercapacitor, showcased in Figure 10. It can be seen that initially when the supercapacitor is used extensively during the peaks of the current requests its voltage (and SOC) drops to the

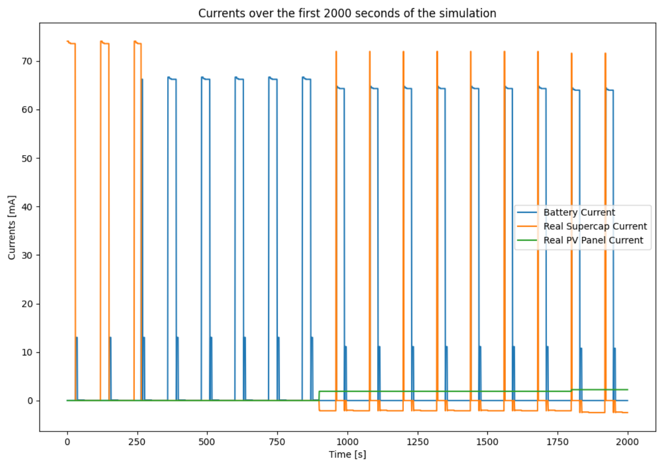

Figure 9: Currents of the storage devices and the PV panel

minimum level. It stays here until the PV panel starts charging it. When it goes beyond the minimum threshold it supplies the system again, but only for small periods. As the provided charging current increases so does the voltage of the supercapacitor until it is fully charged. At that point, the PV panel's current is channelled to the battery. The negative charge peaks observed at the end of the x-axis are the currents recharging the capacitor that depletes due to the leakage current. This behaviour reaffirms the correct supercapacitor and bus behaviour.

Finally, the interaction between the battery and the supercapacitor can also be investigated. Looking at Figure 11, it can be seen that the supercapacitor and the battery charge during the same periods, when the PV panel provides current. It can also be seen that the supercapacitor starts recharging before the battery, which is the expected behaviour. Furthermore, it can also be witnessed that the policy is not perfect. The simulation ends when the battery fully depletes, however, the supercapacitor still has some utilizable charge.

5 Conclusion and future work

During this project, a supercapacitor model was selected from the different options found in the literature. The model was implemented in *SystemC* and integrated into the EM4IoT course's third laboratory simulation environment. Results support the correct behaviour of the supercapacitor and the integration. Moreover, a bus policy was developed that orchestrates the operation of the battery and the supercapacitor ensuring their correct functioning.

As future work, the bus policy could be improved, as results show (11) that it is sub-optimal in its current form. Additionally, the threshold values can be further refined based on the system's loads. Another interesting future direction can be to compare the different supercapacitor models.

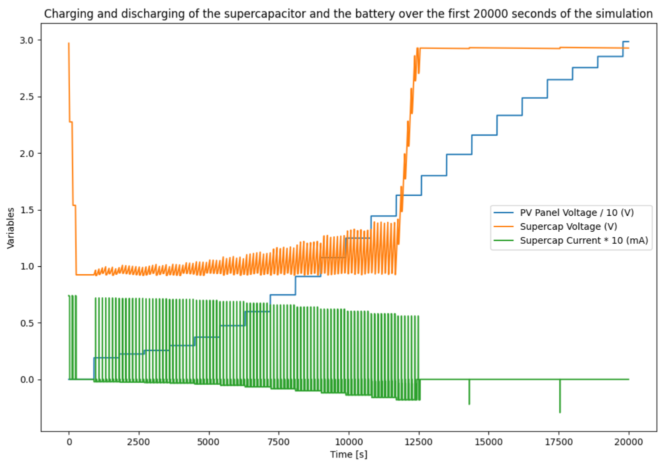

Figure 10: Charging and depletion of the supercapacitor over the first 20000 seconds

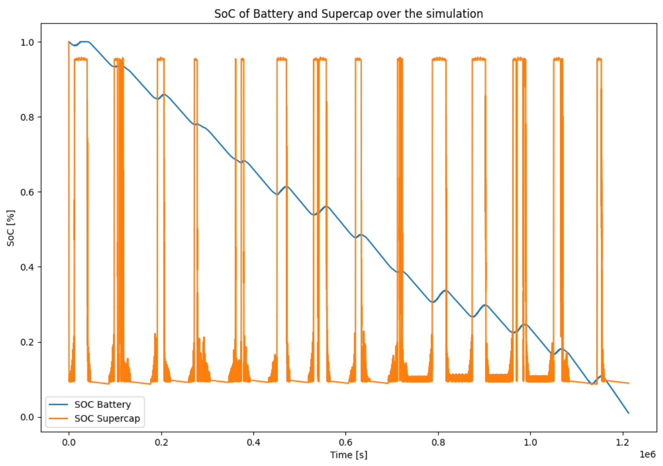

Figure 11: SOC of the battery and the supercapacitor

Bibliography

[1] Hakeem Ayornu et al., *Fabrication and Characterization of Semi-Crystalline and Amorphous Dielectric Polymer Films for Energy Storage.* https://www.scirp.org/journal/paperinformation?paperid=121475

[2] Zineb Cabrane, Soo Hyoung Lee, *Electrical and Mathematical Modeling of Supercapacitors: Comparison.* https://doi.org/10.3390/en15030693

[3] Wenlong Wang, *Modeling and Simulation of Cyber-Physical Systems using SystemC.* https://webthesis.biblio.polito.it/7579/1/tesi.pdf

[4] Gamry Instruments, *Testing Super-Capacitors Part 1: CV, EIS, and Leakage current.* https://www.gamry.com/assets/Uploads/Super-capacitors-part-1-rev-2.pdf

[5] Kyocera, *High Capacitance Cylindrical SuperCapacitors.* https://www.mouser.it/datasheet/2/40/AVX_SCC-3084824.pdf

[6] Maxwell Technologies, *3.0V 3F ULTRACAPACITOR CELL.* https://www.mouser.it/datasheet/2/257/3V_3F_datasheet-1535547.pdf

[7] Accellera *SystemC Analog/Mixed-Signal User's Guide* https://www.accellera.org/images/downloads/standards/systemc/Accellera_SystemC_AMS_Users_Guide_January_2020.pdf

Appendix A

Supercapacitor implementation in *SystemC*

In the following listings, the implementations of each class mentioned in this report are provided. The code of the classes that are not new but were significantly modified (*bus*) is also included. The *main.cpp* is not included as it contains only instantiations.

Complying with the *C++* coding conventions each class is divided into *header* files and *cpp* files. In the appendices the *cpp* files are only included if they contain more than just the *include* of the header file.

1 supercap_eln.h

```
#ifndef SUPERCAP_ELN_H
#define SUPERCAP_ELN_H

#include <systemc-ams.h>
#include "config.h"


SC_MODULE(supercap_eln)
{
    public:
        // Interface and internal components declaration
        sca_tdf::sca_in<double> pI_in; // Requested supercapacitor current
        sca_tdf::sca_out<double> pV_out; // Provided supercapacitor voltage

        // transformers bw ELN and TDF
        sca_eln::sca_tdf::sca_isource iin;
        sca_eln::sca_tdf::sca_vsink vout;

        // ELN components
        sca_eln::sca_c C_sc; // Capacitance (for supercap)
        sca_eln::sca_r R_l; // Leakage resistance
        sca_eln::sca_r R_s; // Series resistance

        // Constructor
        supercap_eln( sc_core::sc_module_name nm, double c_par, double r_l_par,
    double r_s_par, double v_max_par):
            pI_in("pI_in"),
            pV_out("pV_out"),
            iin("iin"),
            vout("vout"),
            MAX_V(v_max_par),
            C_sc("C_sc", c_par, c_par * v_max_par),
            R_l("R_l", r_l_par),
            R_s("R_s", r_s_par),
            node_top("node_top"),
            node_mid("node_mid"),
            gnd("gnd")
        {
            iin.inp(pI_in);
            iin.p(node_top);
            iin.n(gnd);

            vout.p(node_top);
            vout.n(gnd);
            vout.outp(pV_out);

            C_sc.p(node_top);
            C_sc.n(node_mid);

            R_l.p(node_top);
            R_l.n(node_mid);

            R_s.p(node_mid);
            R_s.n(gnd);

            cout << "Supercapacitor created, C:" << C_sc.value << ", R_l:" << R_l.
    value << ", R_s:" << R_s.value << endl;
        }

    private:
        const double MAX_V;

        // internal node and ref node
        sca_eln::sca_node node_top, node_mid;
        sca_eln::sca_node_ref gnd;
};

#endif
```

Algorithm A.1: supercap_eln.h

2 supercap_tdf.h

```
#ifndef SUPERCAP_TDF_H
#define SUPERCAP_TDF_H

```

```
#include <systemc-ams.h>
#include "config.h"

SCA_TDF_MODULE(supercap_tdf)
{
    public:
        // Ports
        sca_tdf::sca_in<double> pV_in; //incoming voltage

        sca_tdf::sca_out<double> pV_out; //forwarded voltage
        sca_tdf::sca_out<double> pE_out; //current energy
        sca_tdf::sca_out<double> pSoC_out; //state of charge

        supercap_tdf( sc_core::sc_module_name nm, double c_par, double v_max_par):
                                pV_in("pV_in"),
                                pV_out("pV_out"),
                                pE_out("pE_out"),
                                pSoC_out("pSoC_out"),
                                C(c_par),
                                MAX_V(v_max_par),
                                MAX_E(0.5 * C * v_max_par * v_max_par),
                                SoC_val(SOC_INIT),
                                E_val(SoC_val * MAX_E)
        {
            cout << "Supercapacitor TDF created, C:"<< C << ", max V:" << MAX_V <<
     ", MAX_E:" << MAX_E << ", SoC:" << SoC_val << ", E:" << E_val << endl;

        }

        void set_attributes();
        void initialize();
        void processing();

    private:
        const double C;
        const double MAX_V;
        const double MAX_E;
        double SoC_val;
        double E_val;
};

#endif
```

Algorithm A.2: supercap_tdf.h

3 supercap_tdf.cpp

```
#include "supercap_tdf.h"

void supercap_tdf::set_attributes()
{
    pE_out.set_timestep(SIM_STEP, sc_core::SC_SEC);
    pSoC_out.set_timestep(SIM_STEP, sc_core::SC_SEC);
    pV_out.set_timestep(SIM_STEP, sc_core::SC_SEC);

    pSoC_out.set_delay(1);

}

void supercap_tdf::initialize(){}

void supercap_tdf::processing()
{

    double tmp_v;

    tmp_v = pV_in.read();

    if(tmp_v <= 0)
    {
        cout << "ERROR: 0 or less voltage value" << " @" << sc_time_stamp() << " Value V=" << tmp_v << endl;
        sc_stop();
    }
    if(tmp_v > MAX_V)
    {
        //? if the voltage can be set from eln maybe this is not needed
        cout << "Voltage is greatar than maximum V of supercap" << " @" << sc_time_stamp() << "Value V=" << tmp_v << endl;
        tmp_v = MAX_V;
    }

    pV_out.write(tmp_v);

    if( tmp_v == 0){
        E_val = 0;
    }
    else {
        E_val = 0.5 * C * tmp_v * tmp_v;
    }

    pE_out.write(E_val);

    SoC_val = E_val / MAX_E;

    if(SoC_val <= 0.01)
    {
        cout << "SC SOC is less than or equal to 1%:" << " @" << sc_time_stamp() << endl;
        sc_stop();
    }

    pSoC_out.write(SoC_val);

}
```

Algorithm A.3: supercap_tdf.cpp

4 supercap.h

```
#ifndef SUPERCAP_H
#define SUPERCAP_H

#include <systemc-ams.h>
#include "supercap_eln.h"
#include "supercap_tdf.h"
#include "config.h"

SC_MODULE(supercap)
{
    public:
        sca_tdf::sca_in<double> pI_in; // Battery current
        sca_tdf::sca_out<double> pV_out; // Voltage
        sca_tdf::sca_out<double> pE_out; // Energy
        sca_tdf::sca_out<double> pSoC_out; // State of Charge

        // connecting signals
        sca_tdf::sca_signal<double> sVoltage;

        supercap_eln eln_module;
        supercap_tdf tdf_module;

        supercap( sc_core::sc_module_name nm, double c_par = 3.0, double r_l_par = 430000.0, double r_s_par = 0.062, double v_max_par = 3.0):
                                pI_in("pI_in"),
                                pV_out("pV_out"),
                                pE_out("pE_out"),
                                pSoC_out("pSoC_out"),
                                eln_module("eln_module", c_par, r_l_par, r_s_par, v_max_par),
                                tdf_module("tdf_module", c_par, v_max_par)
        {

            eln_module.pI_in(pI_in);
            eln_module.pV_out(sVoltage);

            tdf_module.pV_in(sVoltage);
            tdf_module.pV_out(pV_out);
            tdf_module.pE_out(pE_out);
            tdf_module.pSoC_out(pSoC_out);
        }

};

#endif
```

Algorithm A.4: supercap.h

5 supercap_converter.h

```cpp
#include <systemc-ams.h>
#include "config.h"


SCA_TDF_MODULE(supercap_converter)
{
    sca_tdf::sca_in<double> pI_bus; // Current requested/delivered to supercapacitor
    sca_tdf::sca_out<double> pI_sc; // Supercapacitor current

    SCA_CTOR(supercap_converter): pI_bus("pI_bus"),
                            pI_sc("pI_sc") {};

    void set_attributes();
    void initialize();
    void processing();

    private:
        const double eta = 0.9; // Efficiency
        double i_bus_tmp;
        double v_sc_tmp;
};
```

Algorithm A.5: supercap_converter.h

6 supercap_converter.cpp

```cpp
#include <math.h>
#include "supercap_converter.h"


void supercap_converter::set_attributes()
{
    pI_sc.set_timestep(SIM_STEP, sc_core::SC_SEC);
}

void supercap_converter::initialize() {}

void supercap_converter::processing()
{
    // Read input quantities
    // div by 1000 because the bus requests in milliamper but the ELN module work in Amps
    i_bus_tmp = pI_bus.read() / 1000;

    pI_sc.write(i_bus_tmp / eta);

}
```

Algorithm A.6: supercap_converter.cpp

7 bus.h

```cpp
#include <systemc-ams.h>

#include "config.h"


SCA_TDF_MODULE(bus)
{
    sca_tdf::sca_in<double> i_mcu; // Requested current from MCU
    sca_tdf::sca_in<double> i_rf; // Requested current from RF module
    sca_tdf::sca_in<double> i_air_quality_sensor; // Requested current from air_quality_sensor
    sca_tdf::sca_in<double> i_methane_sensor; // Requested current from methane_sensor
    sca_tdf::sca_in<double> i_temperature_sensor; // Requested current from temperature_sensor
    sca_tdf::sca_in<double> i_mic_click_sensor; // Requested current from mic_click_sensor

    sca_tdf::sca_in<double> soc_supercap; // State of Charge of the supercap to determine operation
    sca_tdf::sca_in<double> soc_battery;

    sca_tdf::sca_in<double> real_i_pv1;
    sca_tdf::sca_out<double> i_tot_batt;
    sca_tdf::sca_out<double> i_tot_sc;

    SCA_CTOR(bus): i_tot_batt("i_tot_batt"),
                   i_tot_sc("i_tot_sc"),
                   i_mcu("i_mcu"),
                   i_rf("i_rf"),
                   i_methane_sensor("i_methane_sensor"),
                   i_temperature_sensor("i_temperature_sensor"),
                   i_mic_click_sensor("i_mic_click_sensor"),
                   real_i_pv1("real_i_pv1"),
                   i_air_quality_sensor("i_air_quality_sensor"),
                   soc_supercap("soc_supercap") {};

    void set_attributes();
    void initialize();
    void processing();
};
```

Algorithm A.7: bus.h

8 bus.cpp

```cpp
#include "bus.h"


void bus::set_attributes() {}

void bus::initialize() {}

void bus::processing()
{
    const double supercap_limit = 300; // mA for 40s supply
    const double supercap_request_threshold = 50; //above this we use the supercap

    double tot_requested, tot_consumed, tot_batt_requested, tot_sc_requested,
    tot_scavenged;

    // Compute total current consumption
    tot_consumed = i_mcu.read() + i_rf.read()
                          + i_air_quality_sensor.read()
                          + i_methane_sensor.read()
                          + i_temperature_sensor.read()
                          + i_mic_click_sensor.read()
                          ;
    tot_scavenged = real_i_pv1.read();

    tot_requested = tot_consumed - tot_scavenged;

    double soc_sc = soc_supercap.read();
    double soc_batt = soc_battery.read();

    if(tot_requested > 0){
        // using the energy storage devices
        if(soc_sc > 0.1){
            //if we have enough soc in the capacitor

            if(tot_requested > supercap_request_threshold && tot_requested <
    supercap_limit){
                //current is classified as spike but not too high
                tot_sc_requested = tot_requested;
                tot_batt_requested = 0;
            }
            else if(tot_requested > supercap_limit)
            {
                // spike but too high, so battery is needed too
                tot_sc_requested = supercap_limit;
                tot_batt_requested = tot_requested - supercap_limit;
            }
            else {
                //below supercap threshold, not spike, everything by battery
                tot_sc_requested = 0;
                tot_batt_requested = tot_requested;
            }
        }
        else {
            // supercap is not charged enough, we use the battery only
            tot_sc_requested = 0;
            tot_batt_requested = tot_requested;
        }
    }

    else {
        //charging the energy storage devices
        if(soc_batt < 0.1) {
            // if battery is dying we charge only that
            tot_batt_requested = tot_requested;
            tot_sc_requested = 0;
        }
        else{
            // if the battery is not dying we prioritize the supercap
            if(soc_sc < 0.95){
                // if the supercap is not charged enough we charge that up first
                tot_sc_requested = tot_requested;
                tot_batt_requested = 0;
            }
            else {
                // if it is charged up, we charge the battery
                tot_sc_requested = 0;
                tot_batt_requested = tot_requested;
            }
        }
    }

    i_tot_batt.write(tot_batt_requested); // tot_requested >= 0 ? pow_from_battery
     : pow_to_battery
    i_tot_sc.write(tot_sc_requested);
}
```

Algorithm A.8: bus.cpp