Modeling and Simulation of Cyber-Physical Systems using SystemC

Wenlong Wang

Collegio di Ingegneria Informatica, del Cinema e Meccatronica
Politecnico di Torino

This dissertation is submitted for the degree of
Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA
(COMPUTER ENGINEERING)

April 2018

I would like to dedicate this thesis to my loving parents …

Abstract

The term Electrical Energy System (EES) encompasses a wide range of application contexts as well as different spatial and physical scales, ranging from micro-scale smart systems-on-chip (millimeter/microWatt scale) to large energy distribution grids (kilometer/megaWatt scale). Regardless of their scale, all EESs aggregate similar types of components, (energy/power sources, energy storage devices, power converters/transformers, power transfer interconnects, and loads) and have similar functional properties and optimization objectives.

Normally when someone would like to build an EES design, a good knowledge on both energy system design and programming on SystemC-AMS is required. They need to start from designing their system top-down from abstract specifications, e.g., on paper, EDA tools or somewhere else. Then start to write codes for each EES component manually. When they finish coding, to make sure the implementation is consistent with their design, users need to iterate the process of simulating and proofreading to find any contradiction between their initial design and code, and of making the necessary modifications and tuning on their code until everything is fine. This process is difficult, costly and time consuming. It could be even more difficult when they manage a large project with a relative large amount of files. Till now, there is no Integrated Development Environment (IDE) for accomplishing the entire process of EES design on SystemC-AMS. By this reason, developing an IDE specified for EES design is a meaningful work.

The goal of my thesis is automating the process of constructing EESs, with the support of a custom IDE. The starting point is the formalization of the EES architecture, by classifying system components into different types, depending on their role in the energy flow: *Load*, *Energy storage device (ESD)*, *Power source*, *Converter*, *Bus*, *Arbiter* and *Bridge*. As a next step, I provide an IDE as a solution which allows users to build their customized design graphically by dragging-and-dropping components and connections between them. This process is supported by a Graphics User Interface (GUI), to make the process of constructing EES system more user-friendly. Users can then associate each component to a model of its behavior, either by choosing from a library of predefined component models or by including custom or proprietary models. This configuration is then used as input to a code generation tool, that creates the corresponding SystemC-AMS code for future simulation.

This thesis starts from introducing the framework of EESs. Then it presents the developed IDE and the proposed approach for building EESs and generating corresponding code. Finally three examples are provided to demonstrate the working process of the IDE.

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 SystemC and its AMS extension

The system-level vision of the power dimension of a cyber-physical system may be implemented in a number of languages and frameworks, depending on the desired speed/accuracy trade-off and on the need for integration with other simulation tools[3]. In this work the choice fell on SystemC and on its Analog Mixed Signal (AMS) extension for two main motivations. First of all, SystemC is a standard language, thus extensible and free from compatibility and reuse issues, typical of proprietary tool. Furthermore, both SystemC and SystemC-AMS cover a number of abstraction levels, thus allowing to cover a wide range of models and to find appropriate simulation speed/accuracy tradeoffs. The following of this section details the implementation process.

SystemC is a widely deployed extension to C/C++ for describing HW constructs, ranging from register-transfer level up to transactional level. Its recent AMS extension was designed for modelling and simulating interacting analog/mixed-signal functional subsystems. This allows to extend the adoption of a SystemC-based environment also to extra-functional, continuous time domains.

SystemC-AMS (*Analog Mixed Signal*) is the extension of SystemC for modelling and simulating interacting analog and mixed-signal subsystems and HW/SW subsystems[3]. The goal is thus to allow an early simulation and validation of the overall embedded system. SystemC-AMS provides different abstraction levels to cover a wide variety of domains [3].

Timed Data-Flow (TDF) features the modeling of discrete time processes, where simulation is accelerated by defining a static schedule. Traditional HDL processes are scheduled statically by considering their producer-consumer dependencies. Scheduling is computed before simulation starts, thus avoiding the overhead of a discrete-event simulation.

To define the behavior of the component, three parameters need to be determine:

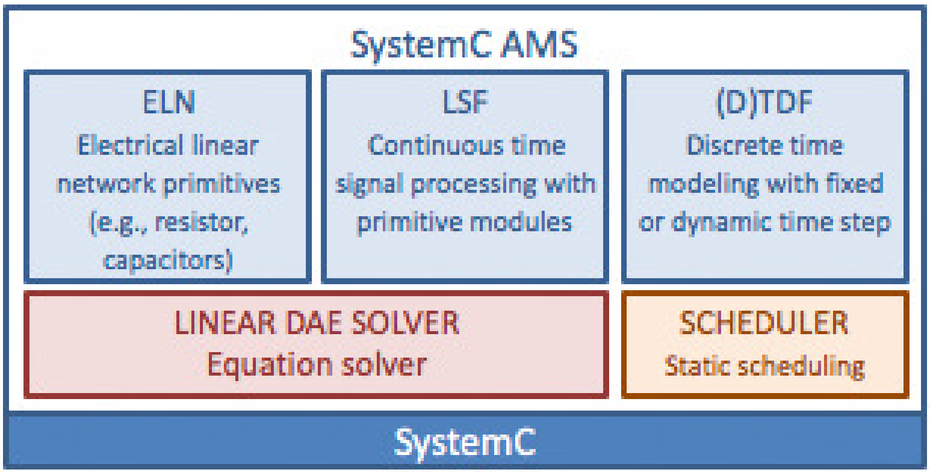


Figure 1.1 SystemC-AMS language hierarchy

- At initialization time (initialization() function)
- At each activation (processing() function)
- Define the activation timestep for each module

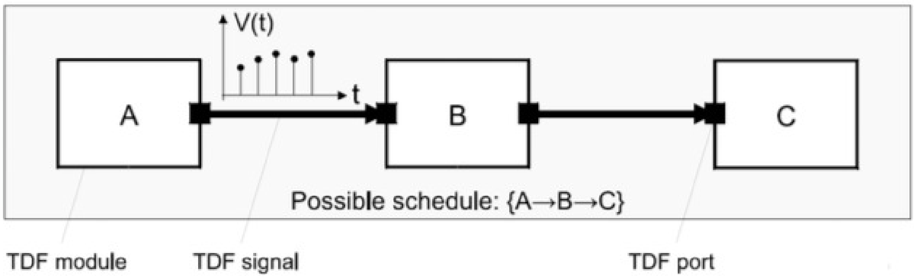


Figure 1.2 SystemC-AMS TDF

Linear Signal Flow (LSF) supports the modeling of continuous time behaviors, through a library of primitive modules, e.g., addition, multiplication, integration, or delay, each associated with a linear equation. A LSF model is built by connecting such primitives. Each primitive model is associated with a linear equation modeling its functionality, e.g.,the integration function. As a result, a LSF model defines a system of linear equations that is solved by a linear DAE solver.

Electrical Linear Network (ELN) models electrical networks through the instantiation of predefined linear network primitives, e.g., resistors or capacitors, where each primitive is associated with an electrical equation, which are used as macro models for describing the continuous-time relations between voltages and currents. Each primitive is associated

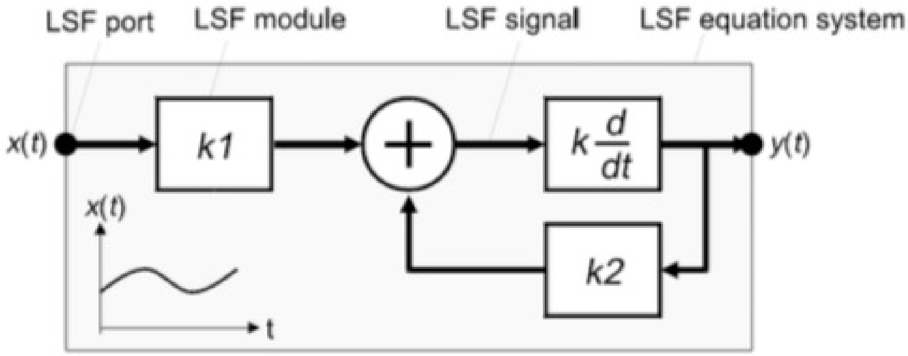


Figure 1.3 SystemC-AMS LSF

with the corresponding electrical equation. A SystemC-AMS AD solver analyzes the ELN and LSF components to derive the equations modeling system behavior, that are solved to determine system state at any simulation time. In Figure 1.4, it shows ELN model examples of resistors, capacitors and inductances.

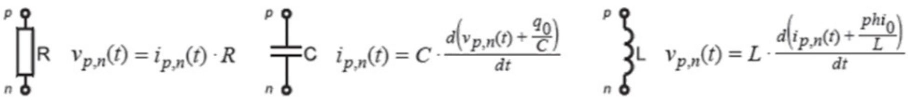


Figure 1.4 SystemC-AMS ELN model examples

1.2 Qt

Qt is an application framework that is used for developing application software that can be run on various software and hardware platforms with little or no change in the underlying codebase, while still being a native application with native capabilities and speed[2].

There are three main features of Qt:

- **Fast**

  Qt provide a highly productive C++ framework complete with cross-platform libraries, APIs and tools for faster time to market.

- **Easy**

  Qt's easy-to-use and flexible IDE and design tools include ready-made controls and out-of-the box functionality for efficient UI design using drag and drop tools, declarative programming with Qt Markup Language (QML) or imperatively with C++.

- **Future-proof**

  Qt's open, extensible and modular C++ framework supports a cost-efficient software development life cycle.

Qt allows users to use large amount of predefined libraries to rapidly build their applications. Qt is a perfect tool for the ones who need a cross-platform software solution. User can both develop and compile their application across Windows and Linux-like systems. A strong feature on Graphics application built is supported by Qt. By this reason, Qt is one of the most popular tools on Graphics interface development domain.

1.3 Modeling and Simulation of Electrical Energy Systems (EES)

The term Electrical Energy Systems (EES) encompasses a wide range of application contexts as well as different spatial and physical scales, ranging from micro-scale smart systems-on-chip (millimeter/microWatt scale) to large energy distribution grids (kilometer/megaWatt scale). Regardless of their scale, all EESs aggregate similar types of components, (energy/power sources, energy storage devices, power converters/transformers, power transfer interconnects, and loads) and have similar functional properties and optimization objectives.

The traditional approach to EES design relies on a model-based paradigm, which uses built-in models provided by commercial simulation platforms like Matlab/Simulink. The support by commercial tools is the main if not the only motivation for the adoption of this approach. While robust and easy to use, in fact, such tools are not easily extensible and are subject to changes of the underlying, proprietary simulation backbone. Furthermore, these tools are not designed to efficiently co-simulate the physical portion (usually continuous-time) and the cyber (*i.e.*, control) portion (usually discrete-time) of the system. This feature clearly limits the possibility of designing EESs following a systematic approach guided by user-defined optimization criteria.

To tackle these two limitations, recently several approaches have appeared in the literature that aim at applying methods borrowed from the domain of electronic systems design [21, 9–11, 4]. These solutions mainly differ in two aspects: the choice of the underlying simulation engine (Matlab [4] vs. SystemC vs. ad-hoc developed C/C++-based simulators [21, 11]), and their degree of generality (specialized for some type of EES, *e.g.*, a smart grid, [11, 4], or general-purpose [21, 10]). One common feature shared by these solutions is that they rely on a database of pre-characterized models of the various EES components. The characterization is carried out by assuming a given level of abstraction and a given semantics of the model and thus the issue of the modularity of the models, (i.e., the possibility of replacing a model of a component with a different one) is not generally addressed [9].

The SystemC language was firstly designed for functional modeling and simulation. In order to adopt an it also in the context of EES, the following steps should be adopted:

- define the main components of an EES and of information and energy flows between them;
- define a standard interface for each class of EES components, so that interfaces only depend on the role of each component, rather than on the characteristics of its implemented model. This allows to simulate EES components at different levels of detail, with the adoption of a number of alternative models, thus determining a trade-off between accuracy and simulation performance;
- analysis of the support of the main models for EES components, to build a straightforward methodology for the construction of the EES simulation.

The proposed approach above profits from the adoption of the SystemC language as baseline framework. Indeed, SystemC is highly modular and thus allows to decouple interface definition from the adopted implementation level. Furthermore, the Analog and Mixed Signal (AMS) extension of SystemC eases the representation of low level circuit models, thus easing the implementation effort. Finally, the adoption of SystemC allows to integrate simulation of functionality and power domain, thus allowing future extensions in the direction of mixed functional and extra-functional simulation.

Chapter 2

Proposed Methodology

This chapter outlines the general methodology proposed by this thesis. I propose a methodology to support EES design: to accomplish design of a certain EES, two phrases of work should to be done. The first phase is the logical design phase, in which designers decide how many components will be used in their design, the topology relationship between components and the model of each component. After the logical design phase, the next step is the SystemC code generation phase. During this phase, the code corresponding to the logical design of that EES would be generated. By this methodology, any EES design could be automatically implemented into SystemC codes which is ready to be simulated, such that people could build their design rapidly in SystemC. In the following part of this Chapter, I will provide a general view of my contribution by this thesis.

2.1 Definition of EES architecture

Before going into details of EES design, the EES system architecture should be introduce first. Components in EES are classified into several kinds of EES components. Each kind of EES components gathers the components with similar functionality and role in entire system. For instance, microcontrollers and sensors consume power from the components, which provide power to implement a certain functionality. All the components that have the same characteristic as microcontrollers and sensors, i.e., which act as a consumer in a EES are classified as load component. In Chapter 3, I will introduce the definition of EES system architecture into details.

2.2 EES integrated development environment

According to the methodology, the first phase is the logical design phase. As I mentioned before, during this phase, three steps need to be done, as shown in Figure 2.1:

- **Instantiation of components**

  Components needed to accomplish energy distribution of design are inserted to logical design (top of Figure 2.1).

- **Instantiation of connections**.

  Connections between components are instantiated, and the corresponding topology relationship between components is built (middle of Figure 2.1).

- **Model selection**.

  User-defined or library models are selected as implementation of each component (bottom of Figure 2.1).

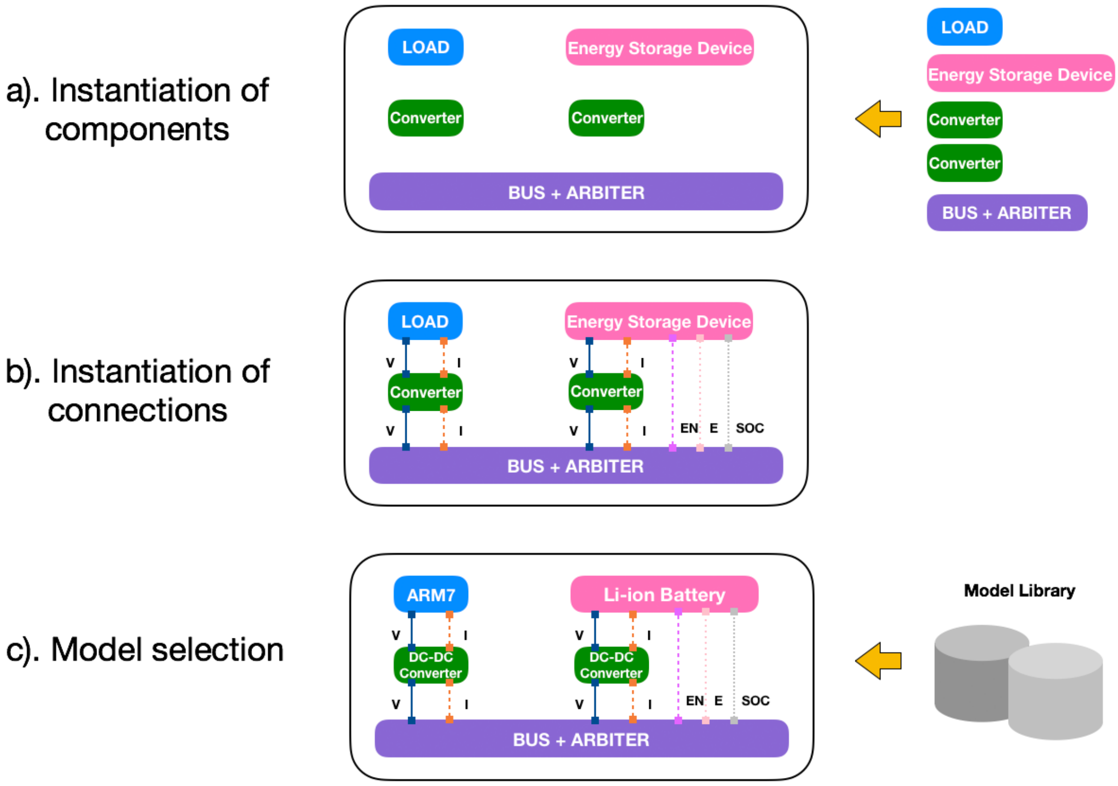


Figure 2.1 Logical design phase

In the traditional flow, all aforementioned steps are done manually. It normally takes a long time, and it is extremely inconvenient when designers modify their design. To solve this problem, I developed an integrated development environment (IDE) to automate the design flow of EES. At the same time, it gives much more flexibility to the designer such that they are able to modify their design in a rapid and convenient way. The functionality of IDE will be introduced in Chapter 4.

2.3 SystemC code generation

After the logical design phase, the second step is to generate the SystemC implementation of the EES design, to validate the EES designed by the user through simulation. In the traditional design flow, designers need to generate SystemC code implementing the design manually, which means they must write the codes line by line by themselves, thus resulting in a time-consuming work. To increase the efficiency of SystemC code generation, the IDE I developed supports the function to generate SystemC code automatically. During the logical design phase, the IDE collects information about current design. In the second phase, the IDE processes such information to generate SystemC code, which is able to simulated during the future work. The process of code generation will be introduced in Chapter 5.

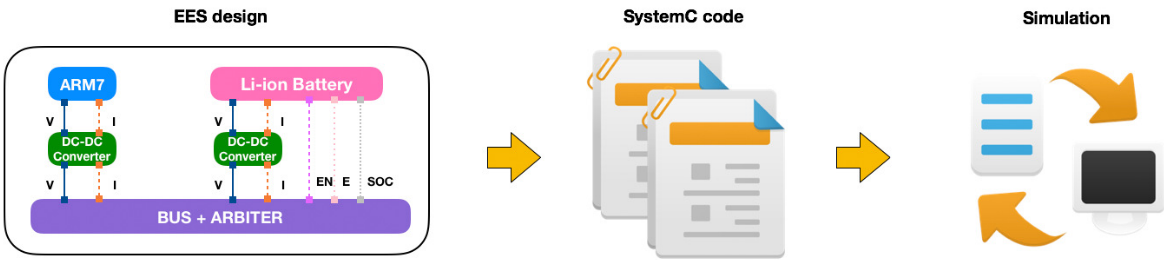


Figure 2.2 Process of code generation

Chapter 3

Framework

3.1 Electrical Energy Systems (EES)

Electrical Energy Systems (EES) are systems which consume, generate, distribute and store energy at various scales, ranging from smart systems-on-chip to smart grids. By this feature, components are classified into seven different kinds which are *Load*, *Energy storage device (ESD)*, *Power source*, *Converter*, *Bus*, *Arbiter* and *Bridge*.

3.2 System Architectural Template

Depending on the different role of components from the power perspective in system, it is necessary to define a template for the EES system. This will impact on the EES simulation infrastructure in terms of both implementation and interface of the components.

Figure 3.1 outlines the proposed template of smart system from the power perspective. Depending on the power feature, components are classified into following EES components:

- **Loads** are components which consume power and implement a certain functionality (memories, sensors, digital cores and etc).
- **Power Sources** are almost infinite sources of power (thermoelectric energy generators and photovoltaic cells).
- **Energy Storage Devices (ESD)** store and provide energy to entire system when needed (batteries, fuel cells and supercapacitors).
- **Converters** adapt current and voltage between two different voltage domains.

- **Bus** is nothing but a wire that holds a given voltage level. Therefore, any model suitable for electrical interconnects can work to model the power bus. Each component is connected to the Charge Transfer Interconnect (CTI) bus through a converter module, in order to maintain compatibility of voltage levels.
- **Arbiters** are used to decide policy for controlling components connected to corresponding bus.
- **Bridges** act as converters between different buses. Thus, their interfaces feature a couple of current (I) and voltage (V) ports for each connected bus.

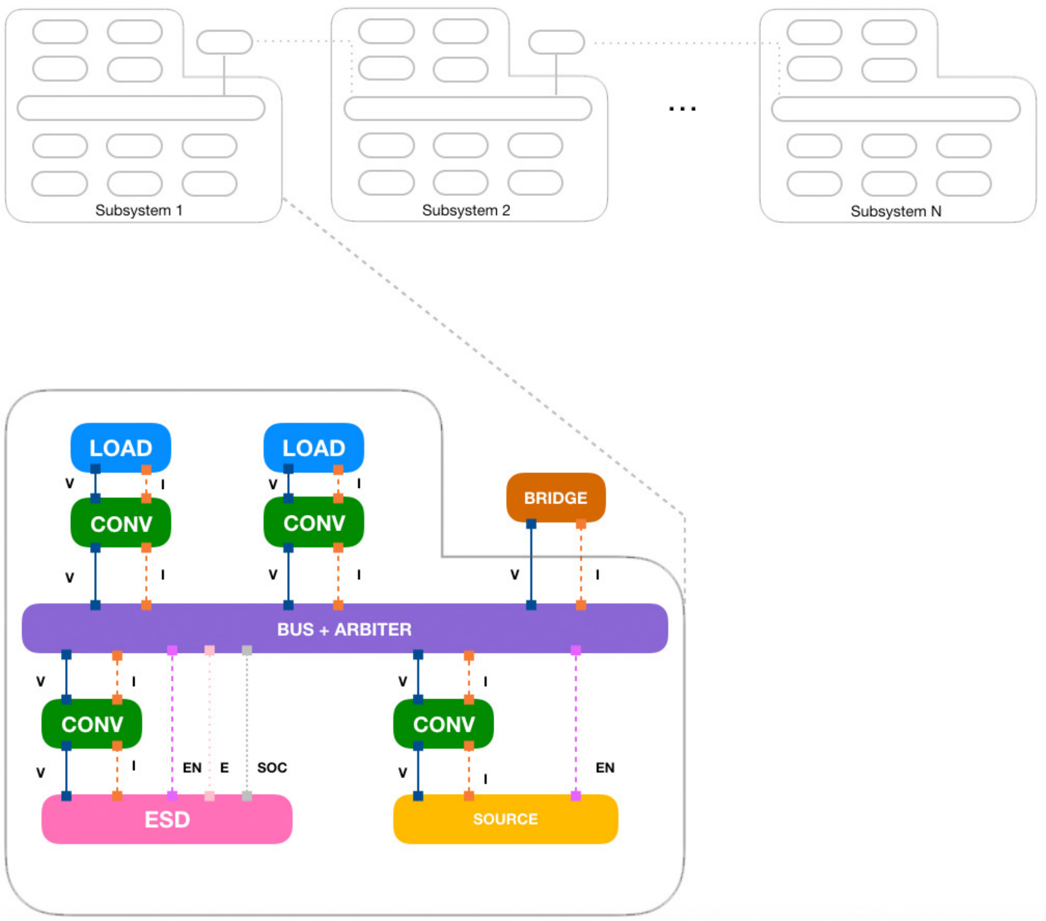


Figure 3.1 Template of the reference architecture of a smart system from the power perspective, including the power bus and the connected components

3.3 Power Interfaces

The different role of components impacts on their simulation characteristics. Energy providers may be enabled or disabled, depending on the operating conditions (*e.g.*, to prefer a power source to an energy storage device, whenever possible). Furthermore, energy storage devices require an intrinsic status information, to trace the available charge over time. As a result, the interface of a component strictly depends on its role inside of the system, thus reflecting the information and energy flow *w.r.t.* the other components.

Table 3.1 Interface of Each Class of Components.

Component	Instances (#)	Power interface
Load	l	(V, I)
ESD	s	(V, I, SOC, E, En)
Power source	p	(V, I, En)
Converter	$c = s + p + l$	(V, I, V, I)
Arbiter	1	$((SOC, E, En)^s, (En)^p, (V, I)^c)$
Power bus	1	$((SOC, E, En)^s, (En)^p, (V, I)^c)$
Bridge	b	(V, I, V, I)

Table 3.1 lists the main components, together with their typical number of occurrences and the ports and connections modeled in the system. (For the sake of clarity, it is assumed that the system contains only one component per type, *i.e.*, $l = s = p = 1$). In this version of the system, the power bus and the arbiter have been merged in a single component, as the goal is to provide a simple simulatable interface, rather than to reflect the physical components of the power system. This *enhanced power bus* constitutes an abstraction of the power behaviors of the overall heterogeneous system.

The interface of each component describes what power information is shared with the remainder of the system. V and I are voltage and current, respectively. SOC and E are the state of charge and the energy (*i.e.*, capacity) of the ESD components. En is an enabling signal, used by the arbiter to activate an ESD or a power source. Environmental parameters that may influence the behavior of components (*e.g.*, temperature and solar radiation) are not modeled on the component interface, as they affect the component behavior rather than the overall system.

The system energy flow moves energy from ESDs and power sources to loads. As a result, the interface of each class of components is as follows:

- *loads* share information about required current (I) and operating voltage (V);

- *ESDs* share their voltage (V), whereas they are provided with the current demand (I) of the system. Furthermore, they must communicate their state of charge (SOC) and their nominal capacity (E), so that they can be activated by the arbiter through an enable signal (En) depending on the actual energy capability;
- the interface of *power sources* includes the supplied current (I) and voltage (V), and an activation signal (En);
- the key role of the *arbiter* is to determine what ESD or power source to use, based on the loads request for power. Therefore, its interface includes a couple of (SOC, E) ports for each ESD and an activation signal (En) for each ESD and power source;
- the *power bus* connects all components, thus its interface consists of a number of I and V ports;
- *converters* feature a couple of ports (I, V) for the input and output values of current and voltages of each connected component;
- *bridges* act as converters between different power buses, and thus feature a couple of ports (I, V) for each connected bus.

3.4 Power components and available models

The definition of the system template and the formalization of simulation interfaces constitute a skeleton of the framework, as they formalize energy and information flows. Each component must then be implemented by adopting suitable models.

Available models strictly depend on the modeled class of components, but they can be classified in terms of abstraction level, i.e., in terms of accuracy w.r.t. the simulated physical phenomena. Functional models implement component evolution with a function (e.g., modeling an equation, a finite state machine or even a simple waveform). A well known example of functional model is the Peukert's model for batteries [15]. On the other hand, circuit models emulate the evolution of a component by building an equivalent electrical circuit [14], reproducing the internal dynamics of the component.

The following sections analyse each class of components, and sketch how each class can be modeled, by adopting effective models suitable for the proposed approach.

3.5 Load

The term load comprises any component which requires certain of power amount during implement a certain functionality. It could be memories, sensors, digital cores and etc. In the guideline of EES, such components are treated with a very different perspective. Their functional evolution is indeed disregarded, and components are seen as *black boxes that require a certain amount of current (I) at a given voltage level (V)*, as shown in Figure 3.2.

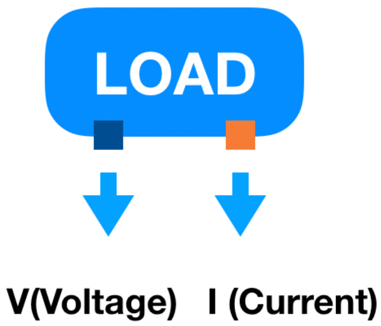


Figure 3.2 Generic interface of a load

The interface adopted for loads highlights that the focus of the overall chapter is solely on tracing the smart system energy flows, represented in terms of voltage levels and current demand/production over time of the various components. The main consequence of this view is that models for functional components are simplistic, to the point that they may fall back to synthetic `V` and `I` traces over time. Figure 3.3 shows some of the mostly adopted models for loads, divided depending on the different levels of adherence to the component evolution.

The most accurate models are *execution traces*, obtained with experimental measurements applied to the component during a typical excerpt of its execution. These model are made up of a couple of waveforms reproducing how current demand and voltage actually evolved over time during the sampled execution excerpt. The traces may be repeated periodically, to simulate longer executions. Figure 3.3.1 exemplifies this by showing a trace for current demand (left) and one for voltage level of the component (right).

In case experimental measurements are not possible or they are considered too accurate for simulation purposes, they can be replaced with *synthetic traces* (Figure 3.3.2). The accuracy of such models strictly depends on their construction process. Typical consumption and voltage values can be extracted from component datasheet, and the trace may be built by relying on statistical information, by reflecting some typical consumption profile of the component, e.g., power values that are more frequent than others. Figure 3.3.2 shows an

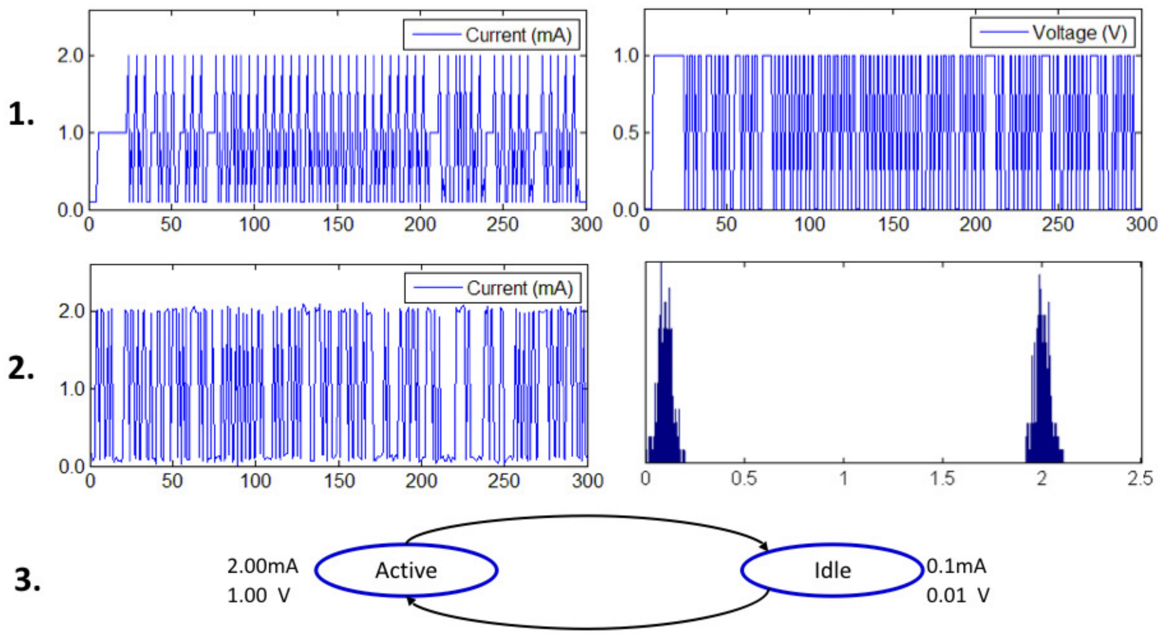


Figure 3.3 Examples of Power Models for Loads: Execution Traces for Current and Voltage (1), Synthetic Statistic Traces (2), and a Power State Machine (3).

example trace modeled as a bimodal distribution (left), where the most frequent values correspond to the typical active and idle current demands (right).

Finally, loads may be modeled as *state machines* listing the internal states of the component (Figure 3.3.3). Transition from one state to another may depend on overall system information or on timers, reproducing a typical execution flow of the component and its dynamic power management policy.

3.6 Power Source

Since power source elements generate power by transforming some environmental quantity in electrical energy. Power sources are the most diverse ones among the various components. The variety of their characteristics tends in fact to follow the scale of the relative system: they range from the μW/mW scale of MEMS-based energy micro-scavengers to the MW scale of large wind turbines, and the very scavenging mechanism can be quite different. Any power source can be considered as a component that generates voltage and current waveforms over time.

This variety in the typologies of power sources makes their modeling poorly scalable and marginally re-usable, thus complicating the objective of using an as much as possible unified modeling approach. The large variety of types of power sources is reflected thus by the many available options for their modeling, ranging from multiphysics-based mechanical models and equation-based mathematical models, up to electrical circuit equivalent models and functional (continuous or discrete-time) macro-models suitable for system-level simulation.

Any power source can be considered as a component that generates *voltage and current waveforms over time*, as shown in Figure 3.4 (signals `V` and `I`). The power produced by the power source strictly depends on the harvested quantity, modeled as an input waveform over time (signal `H`). The power source may be further affected by other environment characteristics, such as temperature (signal `E`). The proposed model is clearly agnostic both of the type of power source and of the scale of managed energy.

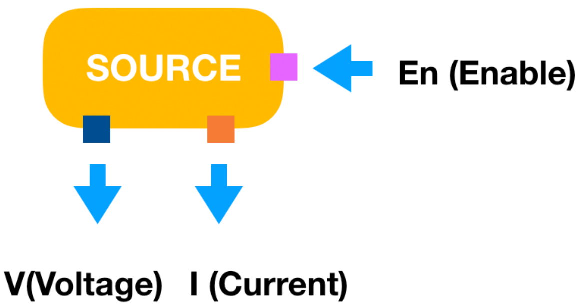


Figure 3.4 Generic interface of a power source.

The analysis of more than 50 datasheets for different types of power sources, including photovoltaic cells and piezoelectric harvesters, lead to the definition of *three main templates* for describing power source behavior[20]: :

- Class 1 graphs are *Current* vs. *Voltage or Power* vs. *Voltage* graphs (Figure 3.5.1), that reproduce the dependency *w.r.t.* the harvested quantity through a number of current/voltage (power/voltage) curves, each one associated with a specific value for the harvested quantity. Power/voltage plots are similar, since they are straightforwardly derived from current/voltage ones by multiplying voltage and current values.

- Class 2 graphs are *Power* vs. *Resistance* graphs. This type of specifications, often used for piezo-electric power sources, models the power source as a family of power/resistance curves, each associated with a specific value of the harvested quantity (Figure 3.5.2).

- Class 3 graphs are *Power* vs. *Harvested Quantity* graphs,popular as specifications of piezo-electric harvesters (Figure 3.5.3). Here voltage is defined in the datasheet as one or more pre-defined voltage levels, while current must be derived from the voltage and the power behavior.

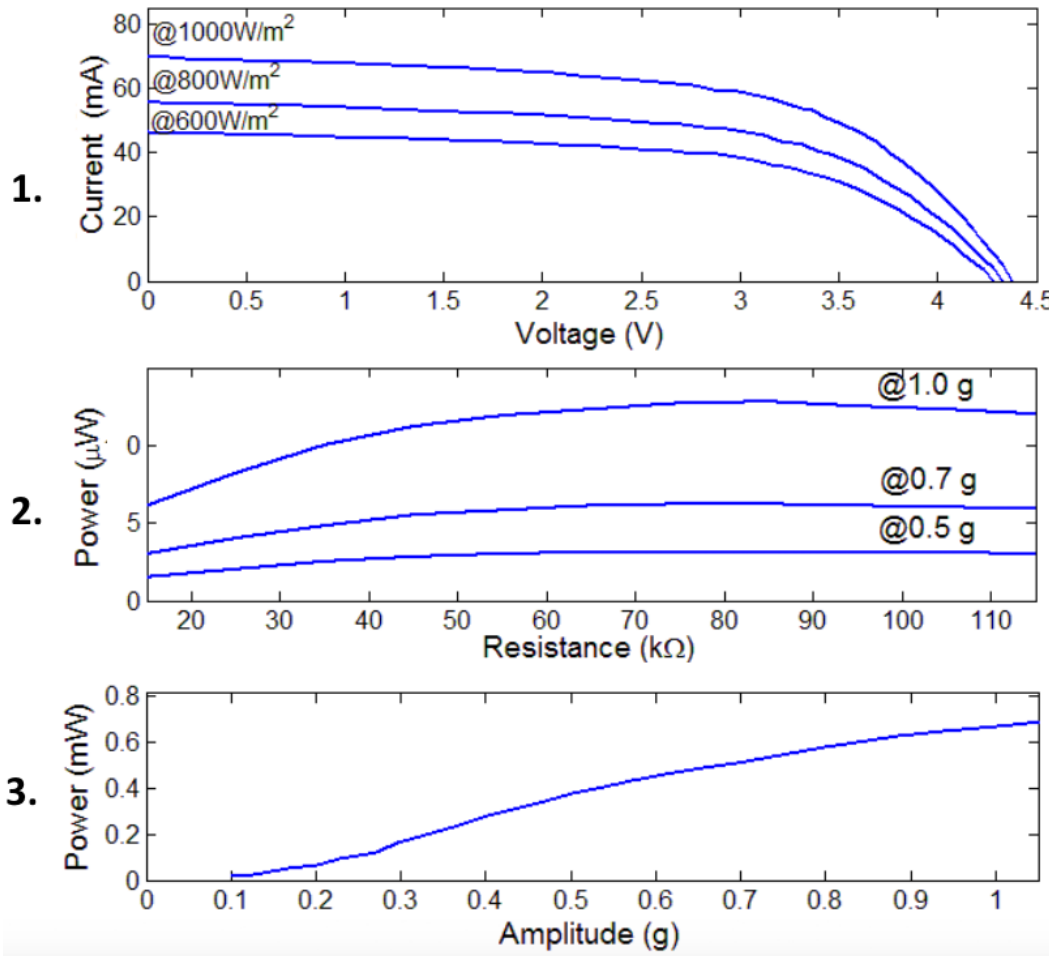


Figure 3.5 Examples of Datasheets graphs: a Class 1 Graph for certain Photovoltaic Cell (Each Curve is Associated to a Value for Irradiance); a Class 2 Graph for a Piezo-Electric Harvester(parameterized *w.r.t.* the corresponding acceleration value); a Class 3 Graph for a Piezo-Electric Harvester.

As a reference form, we adopt Class 3, *i.e.*, *power* vs. *harvested quantity* graphs, as these curves conceptually represent the actual "behavior" of a power source, *i.e.*, a device that generates power according to some environmental quantity. It is thus necessary to re-cast the other two classes of graphs to the canonical form, by making the dependency on the harvested quantity explicit.

Both the classes associate *a curve* to each value of the harvested quantity, thus not univocally identifying an output quantity (*i.e.*, either power or voltage). The necessary transformations reduce the graph to the canonical form by extracting the *maximum power point* (MPP) of the device *w.r.t.* the harvested quantity , *i.e.*, by determining voltage and current values that yield the maximum power for a given environmental condition. In case of

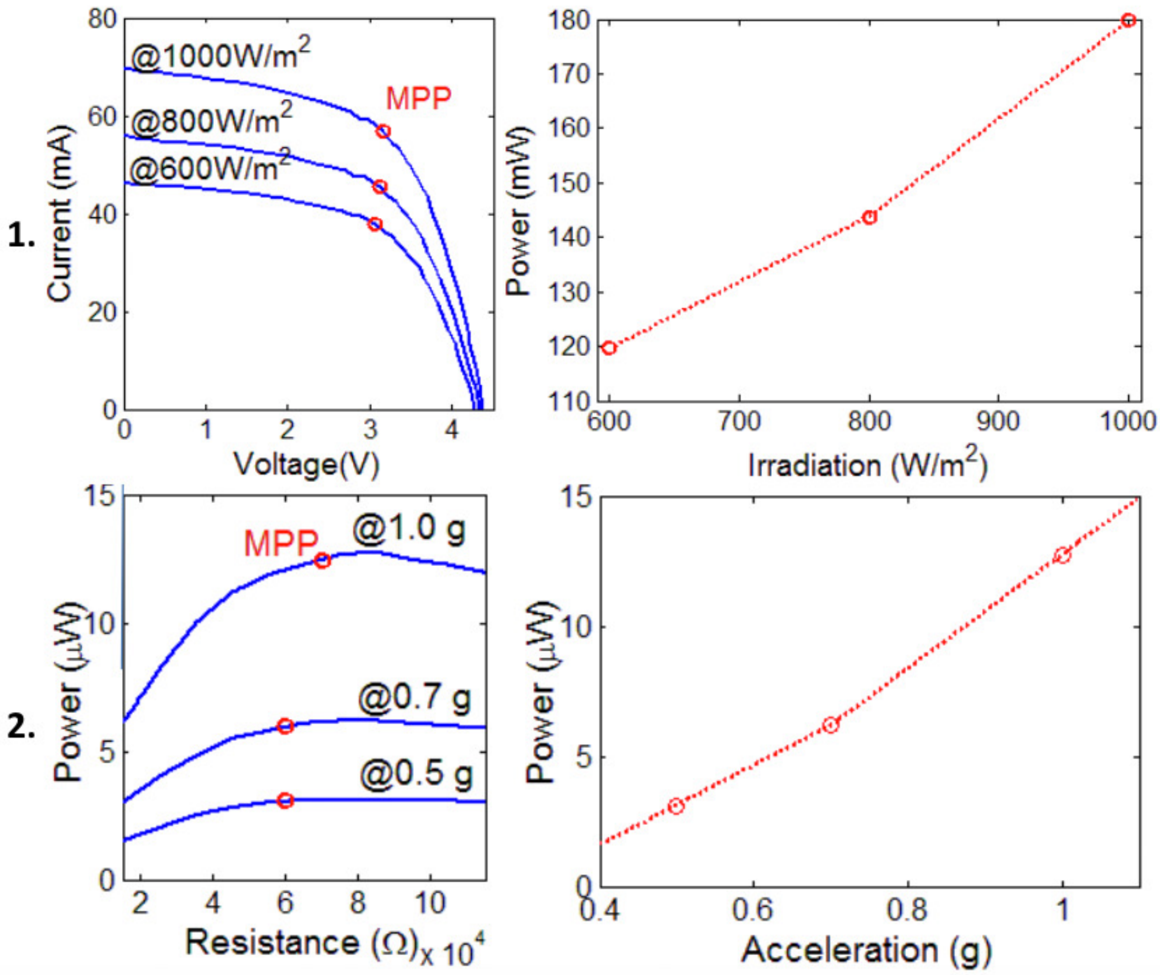


Figure 3.6 Construction of the Generic Power Source Model for the Photovoltaic Cell in Figure 3.5.1 (Class 1) and of the Piezo-Electric Harvester in Figure 3.5.2 (Class 2).

power *vs.* voltage graphs (class 1) or power *vs.* resistance graphs, the MPP for each value of the harvested quantity is the maximum of the corresponding curve. Current *vs.* voltage graphs are instead reduced to power *vs.* voltage graphs. The power source general model can then be easily constructed by plotting the MPPs for the known values of the harvested quantity, which can be interpolated to define a continuous curve. Figure 3.6.1 highlights the MPP for each irradiance value on the current *vs.* voltage class 1 graph in Figure 3.5.1 (left) and the resulting canonical form (right-hand side). Figure 3.6.2 shows the result of applying the same approach to the class 2 graph in Figure 3.5.2.

The behavior of the power source is then described as a pair of curves extracted from the canonical form, modeling voltage and power *vs.* the harvested quantity. Note that the extraction of such information is agnostic both of the type of power source and of the scale of managed energy, and it allows to include the power source models in a wide range of simulation infrastructures.

3.7 Energy Storage Device (ESD)

Energy storage devices (ESD) store energy and provide energy to loads. When necessary, ESDs could be charged by power sources. *batteries* and *super-capacitors* are typical ESDs.

As introduced in previous sections, one key feature of our methodology is to assume a unique interface for a given type of component. Similarly to power sources, ESDs require a general interface that can fit the large variety of available devices. Such interface is as depicted in Figure 3.7.

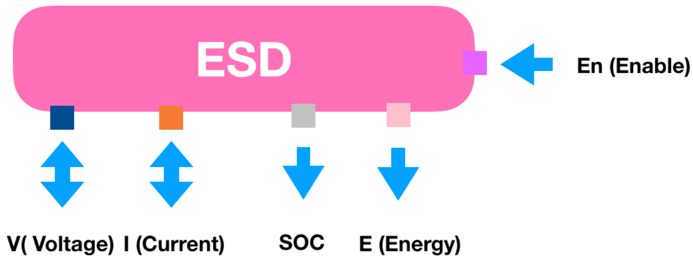


Figure 3.7 Generic interface of a energy storage device

The interface contains the two "native" quantities, *i.e.*, voltage (`V`) and current (`I`); notice that unlike loads and power sources, for ESDs these signals are bidirectional, since it is expected that an ESDs can both accumulate and deliver energy.

Moreover, an ESD features two state signals, denoting respectively its state of charge (`SOC`) and its residual nominal capacity (`E`). Although the two are related, their relation depends on the type of ESD and therefore two separate signals are maintained. Finally, a control signal `Enable` allows one to selectively disconnect the ESD from a load or source to implement specific management policy.

Modeling of Batteries

We consider two popular models with different tradeoffs between accuracy and complexity, namely a functional model based on Peukert's law and a behavioral model based on a circuit equivalent of a battery. These two models are general enough to be applied to *any* type of battery (chemistry or form factor) and they can easily be identified based on a limited set of information typically available in battery datasheets.

The functional model based on *Peukert's equation*[15] expresses the non-linear relation between the battery current I and the equivalent capacity (represented in our model by signal E) through the Peukert's coefficient $n > 1$:

$$E = I^n \cdot t$$

here t denotes time. The state of charge of the battery in this model is obtained as

$$SOC = 1 - \frac{I \cdot t}{E}$$

This model is able to track the load-dependent capacity property of a battery, but not its sensitivity to the current dynamics; current I is assumed to be constant. Identification of this model, *i.e.*, determining its only parameter (n), can be easily done by tabulating (current, discharge times) pairs from the typical battery discharge curves contained in datasheets (Figure 3.8.1) and by fitting these values against a $\frac{1}{I^n}$ curve equation (Figure 3.8.2).

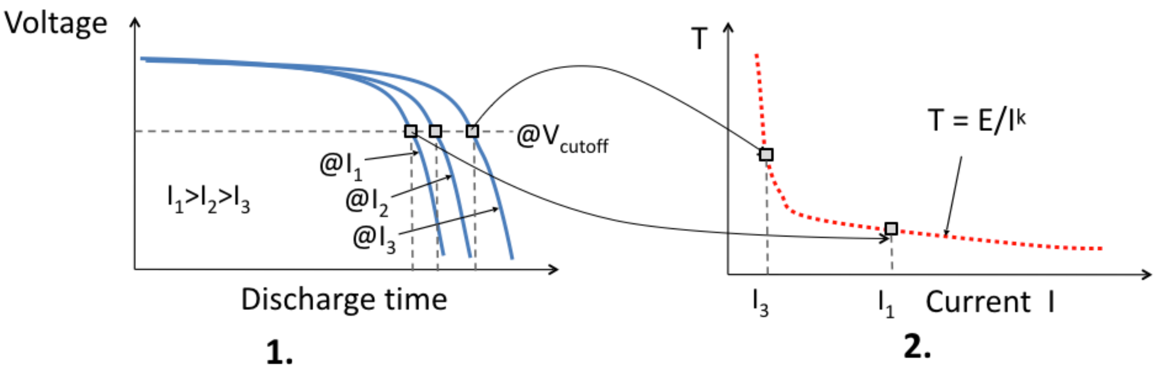


Figure 3.8 Identifying Peukert's Equation from Battery Discharge Curves.

The second and more accurate model is an *electrical circuit equivalent* that mimics the battery behavior [14]. Although many circuit equivalent have been proposed in the literature, we adopt a relatively simple model (Figure 3.9) that simplifies its identification process by requiring only a few, publicly available information.

The model consists of a left part that models the battery lifetime, and consists of a capacitor C (reprenting the storage capacity of the battery in Amp-hourss), and a current generator representing the current requested by the load I_{batt}. The rightmost part models the transient behavior (dynamics) and is mainly characterized by the resistor R that denotes the battery internal resistance; the voltage drop $R \cdot I_{batt}$ that affects the actual battery output V_{batt} mimicks the fact that the battery voltage is adversely affected by larger currents. The connection

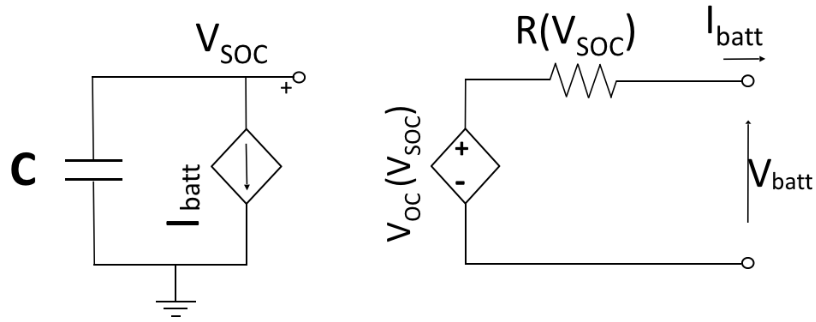


Figure 3.9 Circuit-Equivalent Model Template used for Batteries.

between the two parts is modeled by the voltage generator $V_{OC}(V_{SOC})$: it represents the fact that the open-circuit (OC) voltage of the battery depends on its state-of-charge (SOC), represented by the potential V_{SOC} of the capacitance. Notice also that in the most general case, the internal resistance is also depending on the SOC.

This baseline model is quite modular and can be augmented by incorporating additional elements such as:

- Accounting for the frequency dependence of the internal resistance, *i.e.*, making it an internal impedance; this corresponds to adding one or more RC blocks with different time constants to model short- or long-term components of the transient behavior

- Accounting for the self-discharge of the battery; this is done by adding a resistor in parallel to the capacitance C.

Other "non-behavioral" effects such as temperature dependence or aging can also be included by adding electrical components in the circuit equivalent; since these variants represent long-term effects that are noticeable only across several charge-discharge cycles (days or months) they are not considered in my work.

The basic model of Figure 3.9 can be easily identified by using the (voltage, capacity) or (voltage, SOC) curves provided in most datasheets. At least two curves (for different discharge currents) are needed to determine the internal resistance R (Figure 3.10). At a given voltage, the two curves provide two reference points of the battery voltage $V_{batt}(SOC_1)$ (corresponding to a current I_1) and $V_{batt}(SOC_2)$ (current I_2). By writing the voltage equations in the right-side mesh of the circuit we can write two equations (corresponding to the two currents I_1 and I_2) that allows determining the two unknowns, *i.e.*, $R(V_{SOC})$ and $V_{OC}(SOC)$.

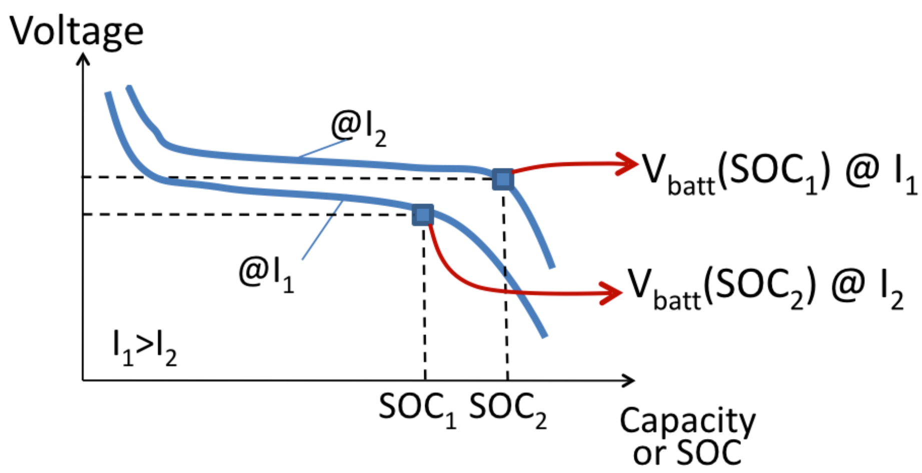


Figure 3.10 Model Parameter Identification for the Model of Figure 3.9.

Modeling of Super-Capacitors

Being an electrical device, a well-accepted model for supercapacitors is an equivalent electrical circuits; as for batteries, the circuit can have different levels of complexity, depending on the phenomena considered in the model.

The circuit considered in this chapter is depicted in Figure 3.11. This first-order model consists of three main components. The capacitor obviously represents the nominal capacitance of the supercapacitor. The series resistance R_s, usually referred to as the *equivalent series resistance* (ESR), is the main contributor to power loss during charging and discharging of the capacitor; its values are generally quite small (in the order of mΩs). The model includes also a parallel resistance R_p which affects s the self-discharge of the capacitor. R_p is always much larger than the ESR (order of kΩs).

More sophisticated supercapacitors models have been devised in the literature; they might include a series inductance, a voltage-dependent capacitor in parallel with C, or replace the (C, R_s) pair with a series of cascaded RC elements to model the frequency dependence of the capacitance. Although the model Figure 3.11 abstracts away these second-order effects, it is accurate enough for a system-level exploration of the power flow for our context.

Identification of the three main model parameters (C, R_s and R_p) in Figure 3.11 is immediate for the former two, but less obvious for R_p. Supercapacitor datasheets systematically provide data for C and R_s, but very seldom informmation about R_p. For this reason, in this chapter we disregard R_p in the model (*i.e.*, assume $R_p = \infty$) and stick to a simple R-C series model of the supercapacitor. In this way, the two values of C and R_s need simply to be transcribed from the corresponding datasheet.

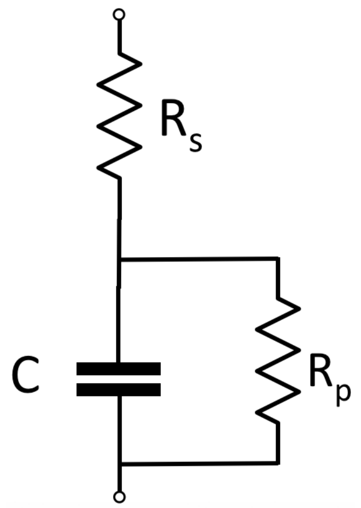


Figure 3.11 Equivalent Circuit Model of a Supercapacitor.

With this choice, we neglect the self discharge behavior of the supercap; as a matter of fact, the battery models considered in the previous Section do not include the modeling of self discharge either (although this effect is less evident in batteries). In that respect, therefore, the battery and supercapacitor models used in our analysis are consistent in terms of modeled effects.

3.8 Converter

When interfacing two different voltage domains, power conversion need to be adapted.Since in our context all signal are DC, only DC/DC conversion is relevant. Figure 3.12 indicates the generic interface of a DC/DC converter consists two pairs of voltage and current signals. For generality, in the figure signals are bi-directional to denote the possibility of a bi-directional flow of the power.

Functionally speaking, the DC/DC converter simply adapts input power to match output power by mean of appropriate circuitry. This process can be characterized by the efficiency of the conversion η [16]: :

$$\eta = \frac{P_{out}}{P_{in}} = \frac{V_{out}I_{out}}{V_{in}I_{in}}$$

The conversion is in general non-ideal and not all input power is transferred to the output, hence $\eta < 1$. The difference between $P_{out} - P_{in}$ represents the *losses* of the converter.

Since the DC-DC converter is an electronic device, in principle a circuit-level model consisting of the interconnection of the discrete components would guarantee the highest

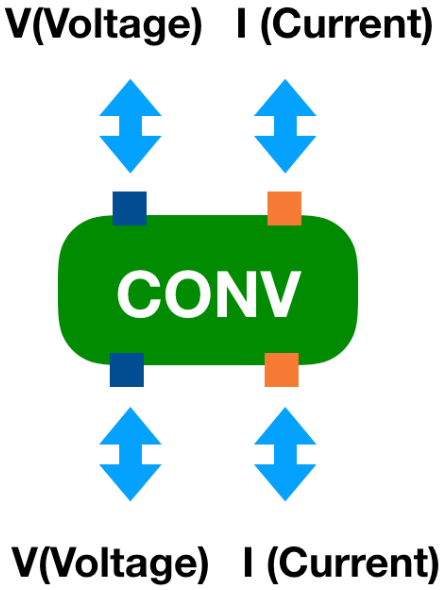


Figure 3.12 Generic interface of a converter

accuracy. However, this would require a specific model for any specific type of converter (*e.g.*, switching vs. linear) and would slowdown the overall simulation. Conversely, a system-level, functional model of the DC-DC converter the describes its efficiency does capture all the non-idealities and is also independent of its implementation.

Key for such a model is the determination of the model parameters, *i.e.*, which quantities affect efficiency. In general, conversion efficiency is affected, in order of relevance, by (i) output current I_{out}, (ii) difference between input and output voltage $\Delta V = |V_{in} - Vout|$, and (iii) absolute values of V_{in} and V_{out}. This applies to the most complex converter architectures containing diodes, inductors, and capacitors (*switching* converters). For simpler architectures like those based on resistive elements (*linear* converters), efficiency is essentially determined by ΔV.

Figure 3.13 shows an example efficiency curve from a step-down switching converter by Maxim in which we can notice how efficiency is quite far from the ideal value, especially when I_{out} gets smaller.

Based on these considerations, we model the power flow of a DC-DC converter through its efficiency, as a quadratic function of the two most relevant parameters (I_{out} and ΔV), as follows:

$$\eta(I_{out}, \Delta V) = k_1 I_{out}^2 + k_2 \Delta V^2 + k_3 I_{out} + k_4 \Delta V + k_5.$$

In case the dependency of efficiency on one of the two parameters is irrelevant (*e.g.*, I_{out} for linear converters), that parameter will simply not appear in the equation. Coefficients

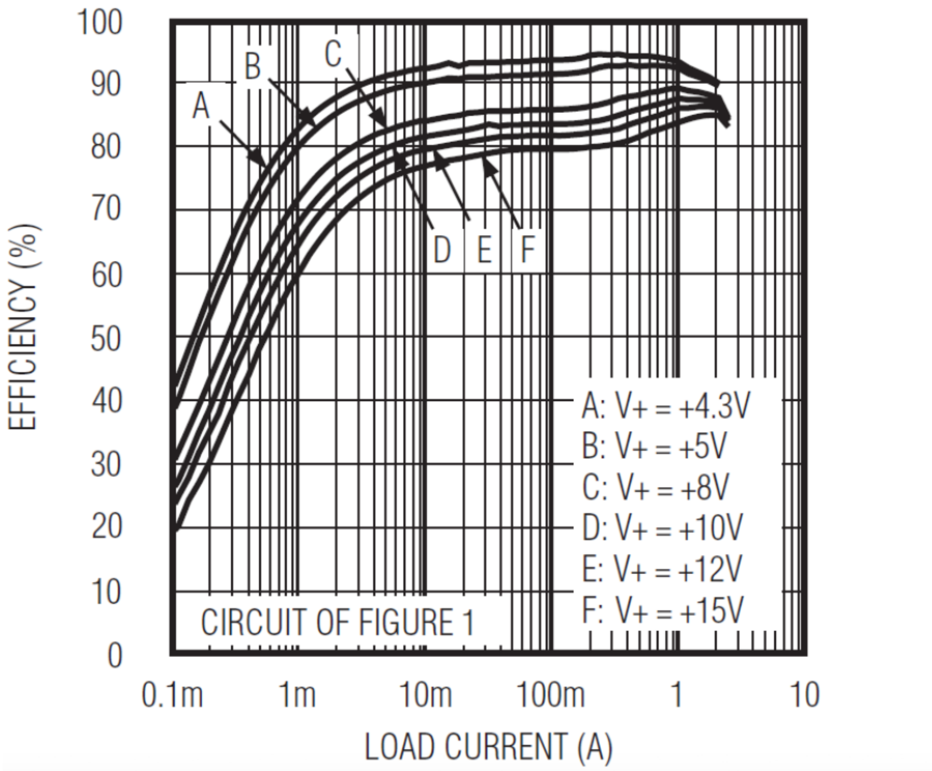


Figure 3.13 Converter Efficiency Curves for the Maxim 1626 Step-Down Converter (When Assuming V_{out} equal to +3.3V).

$k_i, i = 1, \ldots, 5$ are calculated by least-square fitting of the model template, using data obtained by digitizing the efficiency curves provided in the datasheet of the specific device.

3.9 Bus

All components are connected through a Charge Transfer Interconnect bus (CTI bus). It essential to carry around a given voltage level similarly to what happens to "functional" buses in a computing system.

Generally speaking, the power bus is nothing but a wire that holds a given voltage level. Therefore, any model suitable for electrical interconnects can work to model the CTI bus - options range from ideal wires to distributed RLC interconnects.

Accurate models of the bus are particularly relevant in larger-scale energy systems such as micro-grids or hybrid electric vehicles, where interconnections connect components over large areas and support high voltages and can thus incur in significant losses In smart systems, where the lengths of this interconnect is relatively small and voltages in the order of a few volts are supported, we do not need an extremely accurate model for the CTI bus. Two simple options with increasing accuracy are envisioned:

1. an *ideal dc bus* consisting of an ideal wire and a voltage generator at the desired bus voltage;

2. a *purely capacitive bus*, whose capacitance is determined based on the overall bus length and size.

3.10 Arbiter

Arbiter is a kind of EES component which contains policy information of EESs. Different from those components like loads, converters, ESDs, power sources and bridges, which are real devices existing in EES. Arbiter is a kind of logical component which is not a real component connected to entire system. It may be implemented by certain load component like micro-processor. Arbiters acts as controllers in systems. In most cases, since arbiter and bus are both logical components, to simplify the complexity of system view system, arbiters are merged to bus components.

For an arbiter, the key feature is that it decides the policy of entire system [19, 5]. A policy is the rules to decide how certain EES will act depending on the components it contains currently. For example, in a design with two loads, a photo-voltaic panel, a battery and four converters as shown in Figure 3.14, a possible policy could be a “charge allocation” policy, which is as follows:

As long as the power drawn from the PV panel satisfies the power demand of the loads, loads are supplied by the power source. Otherwise, the loads are supplied by the battery, until the SOC is below 10%. Finally, when the PV panel is able to provide power and the power demanded by the loads is 0, the battery is charged by the power source.

In Chapter **??**, when EES examples are introduced, more details about how policy works in a real EES would be described.

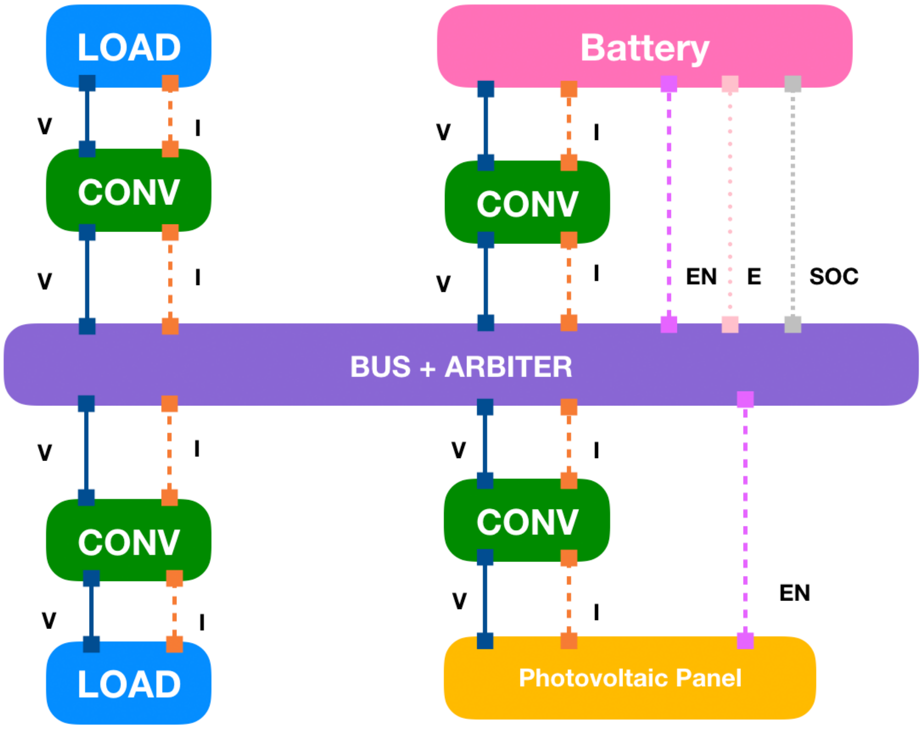


Figure 3.14 An ESS Design Example with several components

Chapter 4

EES Integrated Development Environment

4.1 Description

Normally when someone would like to build an EES system, a good knowledge on both energy system design and programming on SystemC-AMS is required. They need to start from designing their system top-down from abstract specifications, e.g., on paper, EDA tools or somewhere else. Then start to write codes for each EES components manually. When they finish coding, to make sure the implementation is consistent with their design, users need to iterate the process of simulating and proofreading to find any contradiction between their initial design and code, and of making the necessary modifications and tuning on their code until everything is fine. This process is difficult, costly and time consuming. They have to suffer a long working process from they have a general idea about their system to get the entire implementation with SystemC-AMS code. It could be even more difficult when they manage a large project with a relative large amount of files. Till now, there is no Integrated Development Environment (IDE) for accomplishing an entire process of EES system design on SystemC-AMS.

In this chapter, an IDE developed by me will be introduced. It will help user to rapidly build their design and generate corresponding code. With this IDE, the process of constructing EES systems could be automated.

In this chapter, the design flow of EES will be firstly introduced in Section 4.2. Then a general view of the IDE graphics user interface will be mentioned in Section 4.3. The design functions of EES including operations related to components and connections will be described in following section (Section 4.4). Finally, the last section (Section 4.5) in this chapter will be devoted to demonstrate the mechanism of library management.

4.2 EES Design Flow

Before introducing details of IDE, it is better to describe EES design flow first such that it could help readers to understand why the IDE is design like that and how the IDE works.

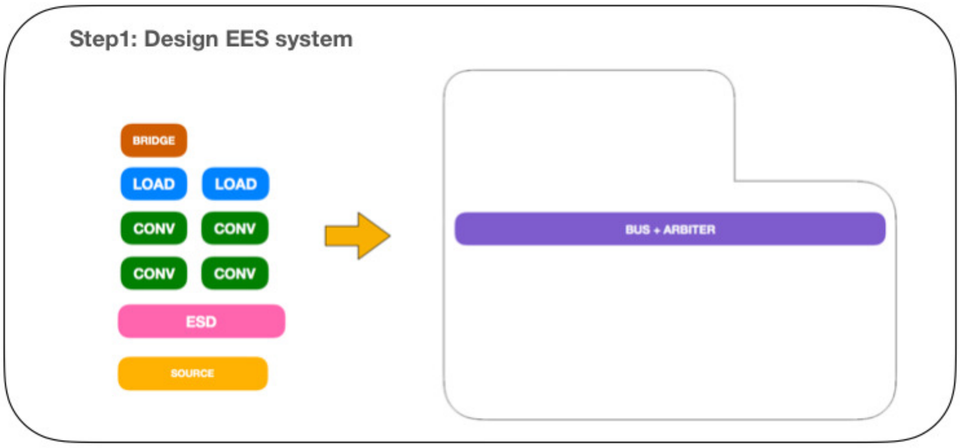
In EES system design, there are three main steps as shown by figure 4.1:

- Users firstly need to build their design by inserting EES components and creating connections between these components. At this time, only logical design is accomplished: all components are something like empty boxes, waiting for model assignment as shown in Figure 4.1a.
- Then models will be assigned to each functional component as shown in Figure 4.1b. These models are stored in an extendable model library. Users are able to maintain this library by importing, modifying and deleting models.
- Finally, the last step is code generation, in which the corresponding files of each component will be filled by code. After this step, the generated codes are ready for future simulation as shown in Figure 4.1c.

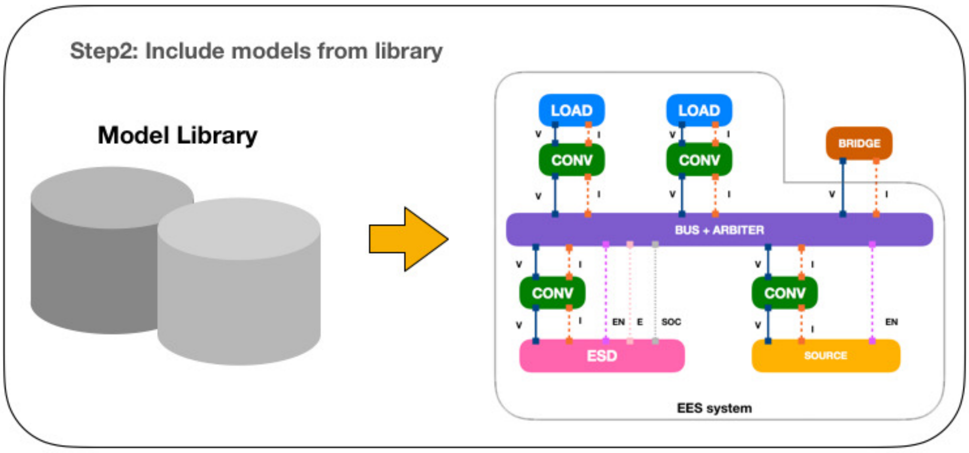
4.3 Graphics User Interface

In the IDE, a graphics user interface (GUI) is provided to offer a user-friendly service for users to build their design. The GUI is made of four main areas, which are workspace, graphics view, menu bar and system status window as shown in Figure 4.2.

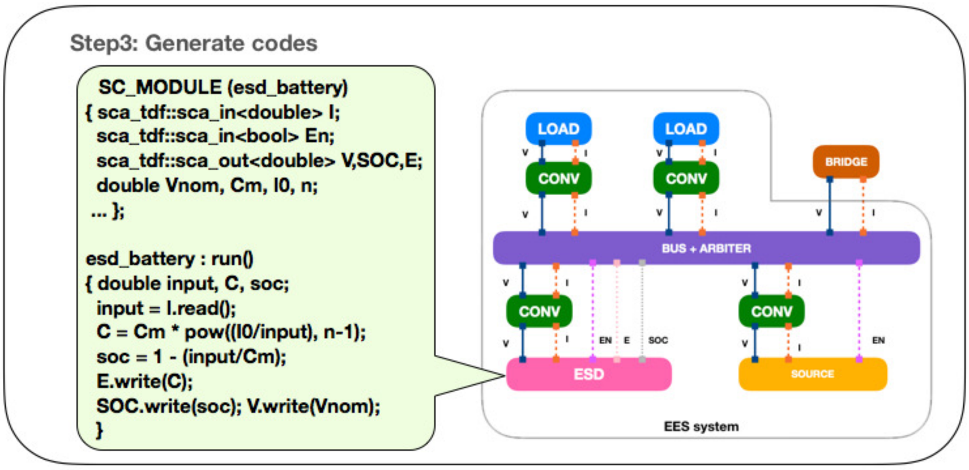
- **Workspace** contains two sub-windows, the former used for displaying information about how files are organized in operating system and the latter is used for monitoring EES component information.
- **Graphics view** is the main operational region for users to design customize EES systems and view file content of generated files.
- **Menu Bar** consists with buttons for the most often-used functions, such as creating new project, code generation and etc.



(a) Design flow step 1: Design EES system



(b) Design flow step 2: Include models from library



(c) Design flow step 3: Generate code

Figure 4.1 Design Flow

- **System status window** gives user feedback information when they interact with certain functionality.

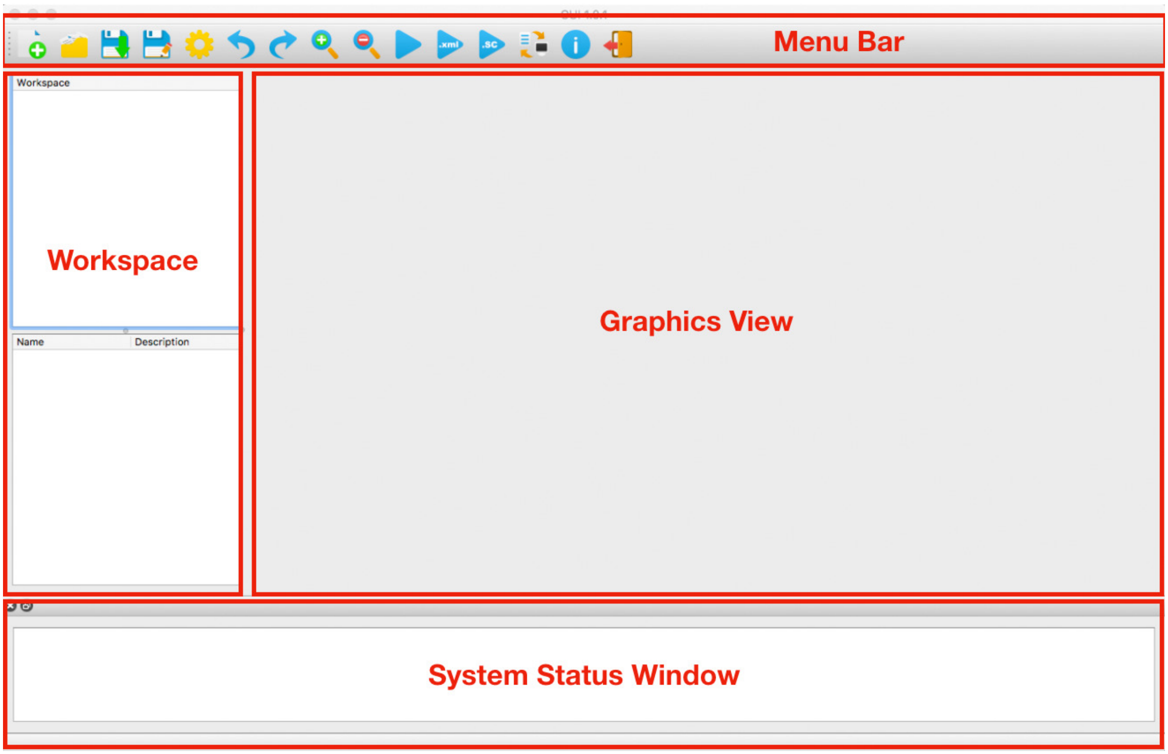


Figure 4.2 Regions of Graphics Interface

To construct a Electronic Energy system, the first step is to create a new project in GUI. By clicking the button for creating new project in menu bar, a dialog for configuring information about new project appears as shown in Figure 4.3. When all the configuration information is confirmed, click "OK" button. A project with all information configured will be prepared well for users as shown in Figure 4.4.

As shown in Figure 4.4, when a new project is created, the corresponding project information (directories, files and default component information) would be displayed in the workspace. Users are able to check how their files are organized in operating system by checking contents in the upper window of workspace. In the lower window of workspace, EES component information will be monitored. When user performs some future operations like code generation in Chapter 5, corresponding file information will be added to upper window of the workspace. At the same time, system feedback information would be displayed in system status window. System feedback information is the information specified to identify system status, such as if the project is successfully created, project path, etc.

Main operations for constructing an EES system will be done in **graphics view** region of the GUI. User can build their EES system by drag-and-drop EES components in this area and they can also check and modify the content of generated SystemC files in this area like shown in Chapter 5.

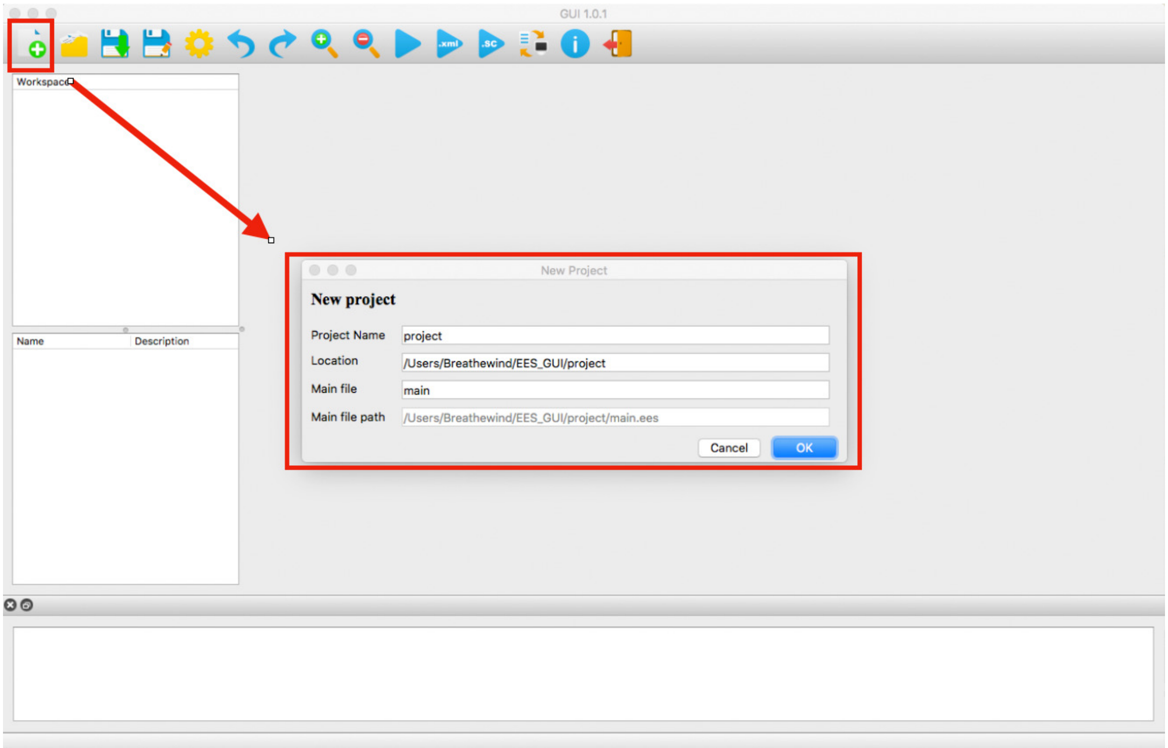


Figure 4.3 Create a new project in GUI

In the GUI after a new project created, the **graphics view** (described in the beginning of this section) consists of three parts: **tab bar**, **button bar** and **operation area** as shown in Figure 4.5.

- **Tab bar** is used for switching content displayed in graphics view.
- **Button bar** contains eleven buttons which could be divided into two different kinds:
  - **Component operation related buttons** which are the button *LOAD*, *ESD*, *SOURCE*, *CONV*, *BUS*, *ARBITER*, *BRIDGE* and *Delete component*. Beside *Delete component* button devoted to remove one component from current design, all the other buttons are used for insert corresponding component into current design.
  - **Connection operation related buttons** which are button *Fast connection*, *Delete connection* and *Normal connection*. The meaning and functionality of the button

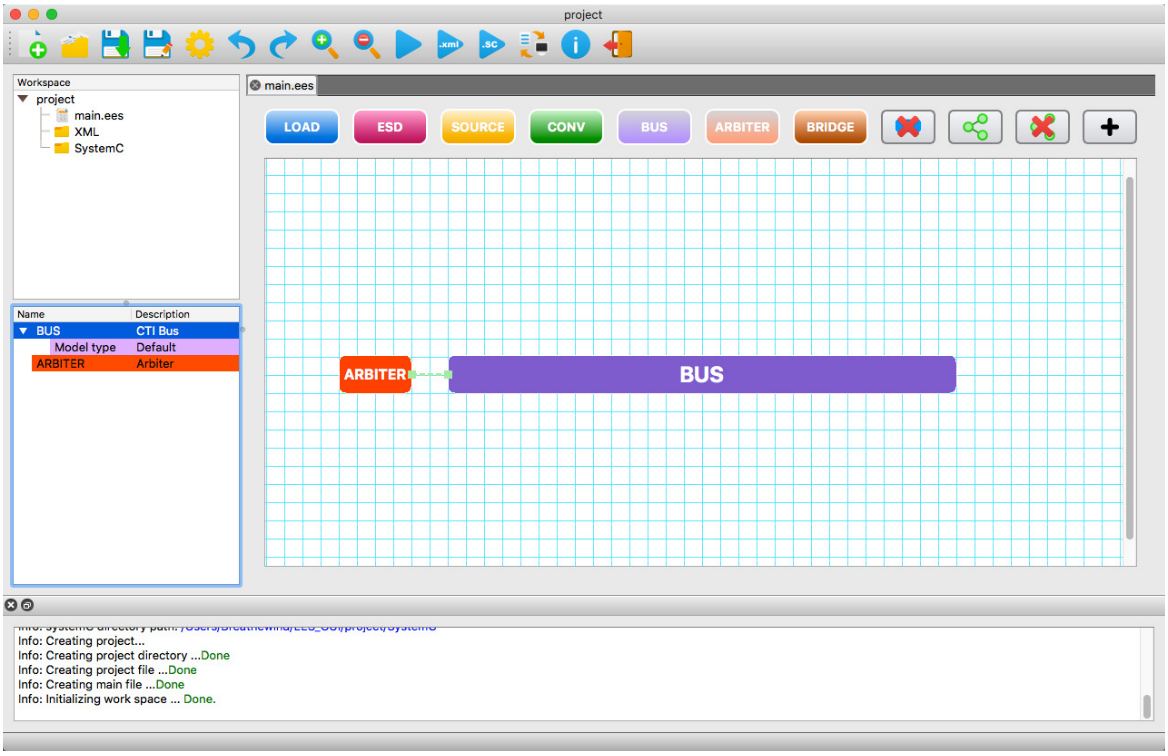


Figure 4.4 GUI after a new project is created

Fast connection and *Normal connection* will be introduced in Section 4.4.2. Here the reason why the order is *Fast connection* - *Delete connection* - *Normal connection* need to be explain. *Fast connection* function offers a very convenient and efficient way to connect two components. When users try to connect two components, *fast connection* function is much easier to use than the *normal connection* function. *Normal connection* function is included to my work only for increasing the flexibility of the connection creation in some very special cases. By this reason, I prefer to order the buttons related to connections by how often they would be used.

- **Operation area** displays corresponding content *w.r.t.* the tab chosen by **tab bar**. Generally, there are two kinds of content mainly would be displayed in this area:
  - **Design content**, which is the graphical design of EES system with various components.
  - **Text content**, which is the text content of generated files.

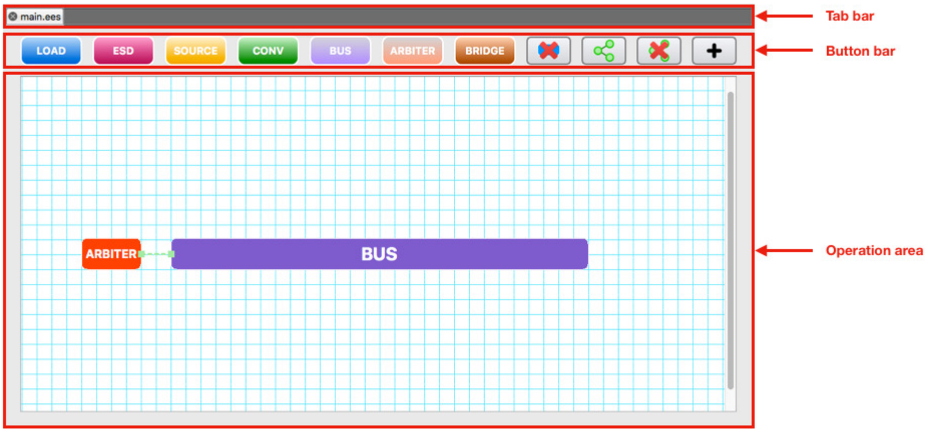


Figure 4.5 Graphics view

4.4 Functions for EES Design

4.4.1 Operations related to components

There are three basic operations related to components, which are *Create components*, *Modify components* and *Delete components*. With these operations, users are able to import any components they need to design space.

Create components

After a new project is created, the system contains only the bus and the bus arbiter. When users want to add a component, they need to click on the icon of the corresponding class in the button bar, and add it to the graphics view. This corresponds to instantiating a component of that class.

Basically, there are two approaches to configure model type to certain (user-defined or even third-party) component. One approach is to assign a model from model library to target component. The other approach is that users can import component design directly from certain file. By the second approach, users have more flexibility to import the files they need directly to their design space.

Among all the functional components, the load component is the special one comparing with other components, since only load components support a third configuration approach which allows users to build their customized model, based on information on the typical load behavior. In the following part, normal approaches and customized approach will be described separately.

Normal approaches to create components

All components have two default configuration approaches: *library model approach* and *load from file approach*.

"Library model approach" allows users to select an existed model from library and assign it to the EES component they would like to create. This approach requires the definition of a model library, containing the most important models for each EES component (as listed in Section 3). The library will be detailed in Section 4.5. In Figure 4.6, it shows a dialog for creating a ESD component by a model from library. By switching content of the Model tab bar, users are able to check model information and model port map among different models. When required model is selected, corresponding ESD component will be created and assigned the corresponding implementation.

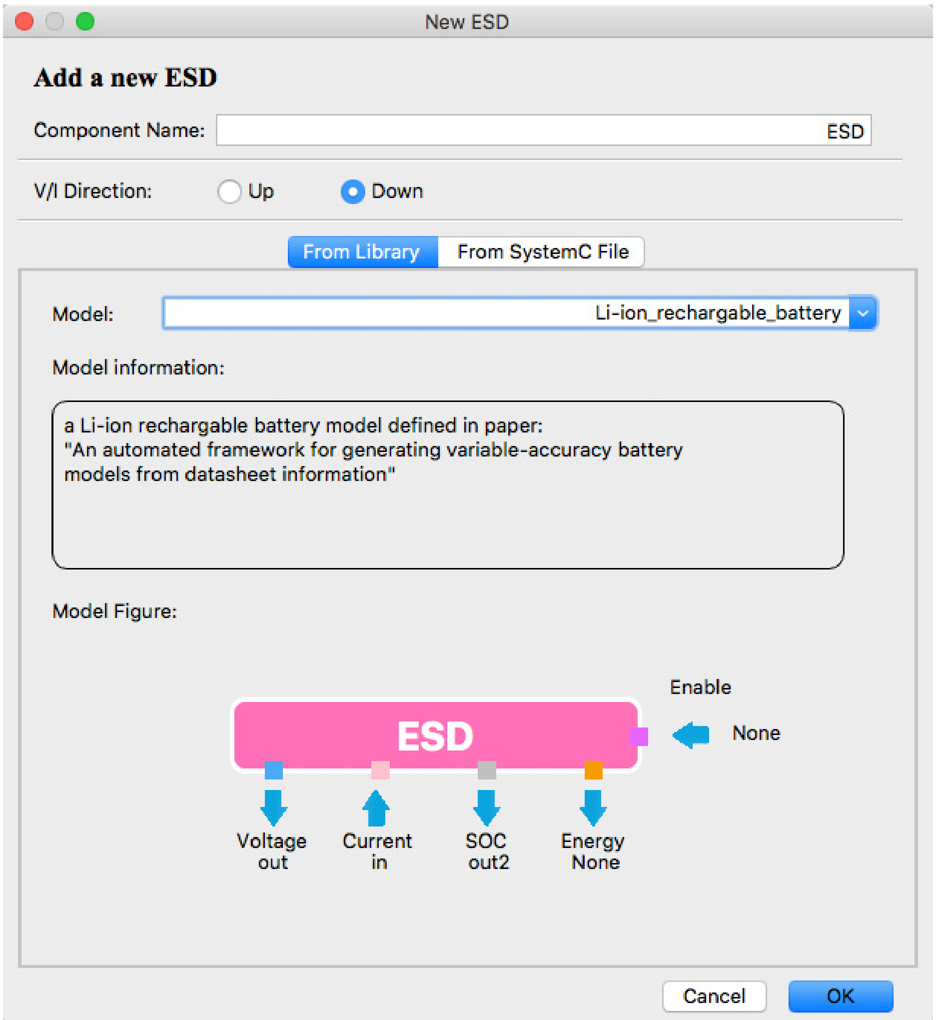


Figure 4.6 An example ESD component created by a model from library

The second approach is "load from file approach" which allows users to import SystemC files as implementation of their components. Sometimes user need to use models from file, *e.g.* , to reuse existing models or since, they may use these models only once, so they do not need to import them to model library. This approach would give user flexibility to import their own SystemC files to current design by user specified file path directly to their design without importing these files as models in model library first.

A full process of creating a converter component by loading model from file is shown in Figure 4.7. First step is to locate the required file position in operating system and click "Load" button as shown in Figure 4.7a. Then target file would be loaded and analyzed. Model information like model name and input/output port would be extracted from file and displayed in corresponding field of dialog as shown in Figure 4.7b. Finally, users assign all the ports required as shown in Figure 4.7 and click "OK", and a converter following selected file implementation will be created in design space.

Create components by customized approach

As I mentioned before, the load component is a special component comparing with other components, since it supports a third configuration approach which allows users to build their customized model, based on information on the typical load behavior. In the following part, I will introduce how to create load components by customized approach.

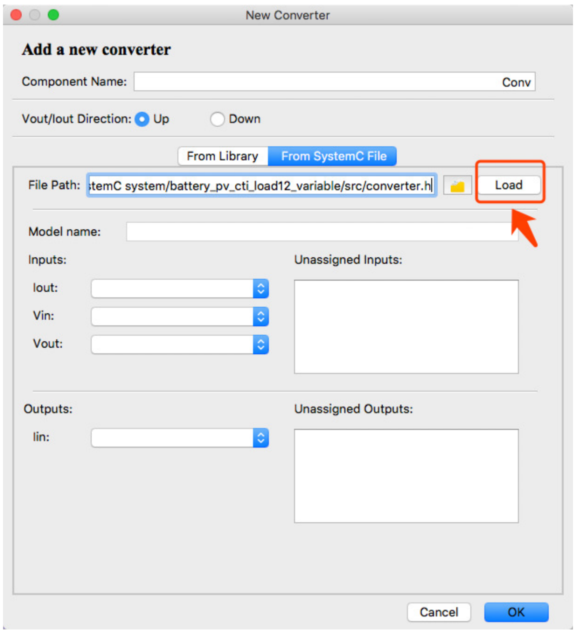
Customized approach allows users to customize load components by specifying a constant output value of voltage and current separately. There are six different ways to specifying the output of load components. Here is the different ways to specify the load component outputs:

- *Constant*

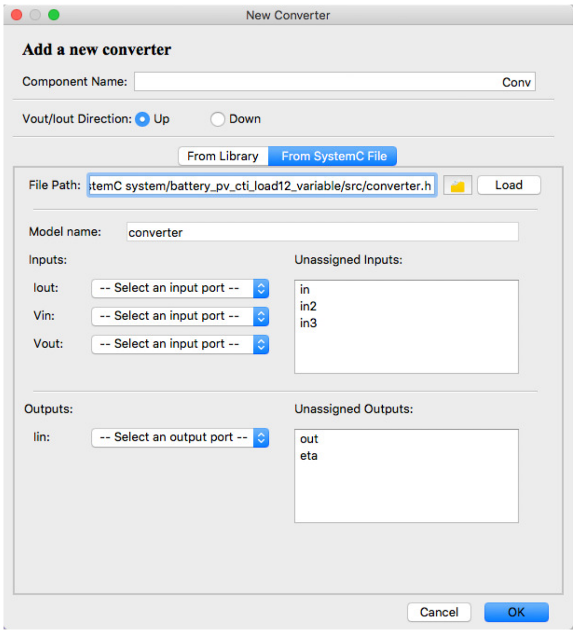
  When user would like to create a load component, they firstly need to click the blue button in button bar of graphics view. Then a an dialog which allows users to choose the approach in which they will build that load component will appear.

  An example of the dialog for creating a load with constant current and voltage output is shown in Figure4.8a. The key parameter for creating a load with constant voltage and/or current output value is the constant value which corresponding output will be. You can see in that dialog, users are able to configure the constant value separately among current and voltage output. The corresponding output value will keep a constant value as specified in the dialog.

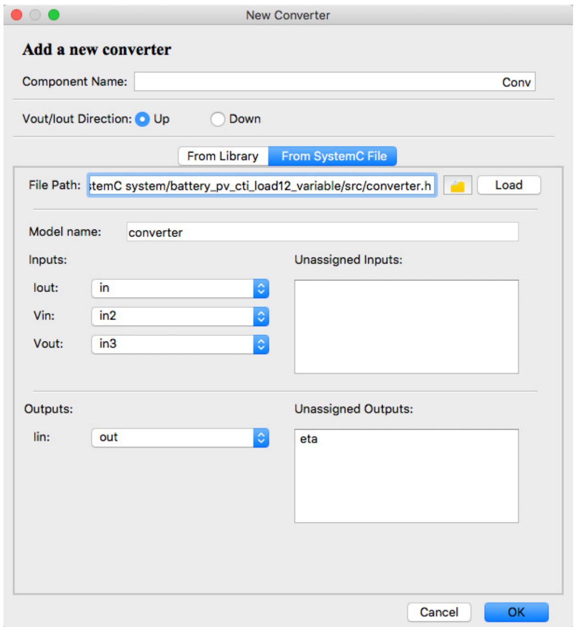
  In Figure 4.8b, the upper part shows the generated of a load component in Figure 4.8a. You can see in the method *processing()* the output of both voltage and current are fixed



(a) Step1: Select file and click "Load" button



(b) Step2: Load model from file

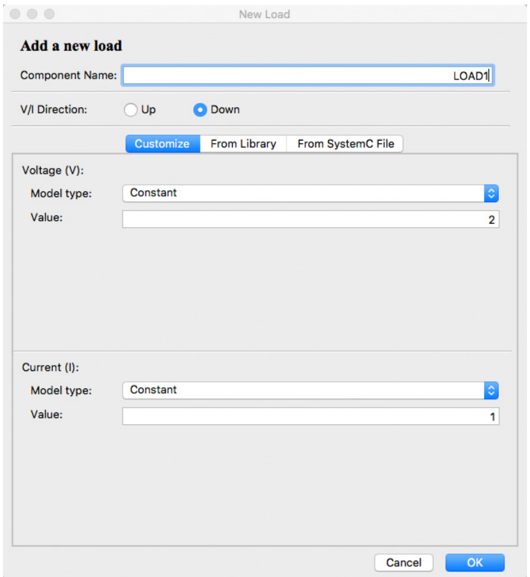


(c) Step3: Port map

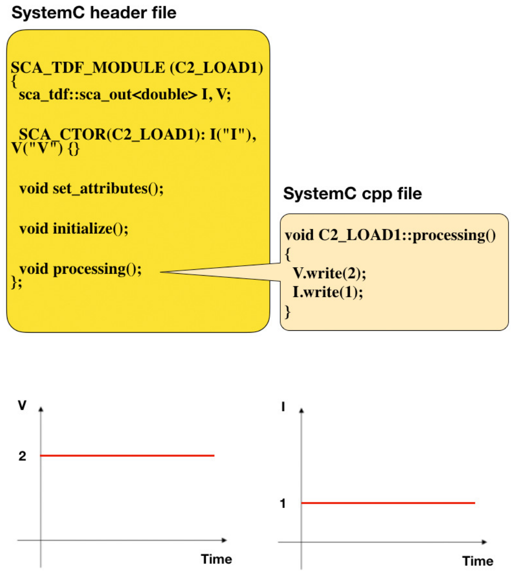
Figure 4.7 Create a converter component by loading model from file

at corresponding value. The lower part in that figure shows the trace of voltage and current output.

- *Load from file*



(a) Dialog of creating a load with constant current and voltage output



(b) Generated code and output trace of the load in Figure 4.8a

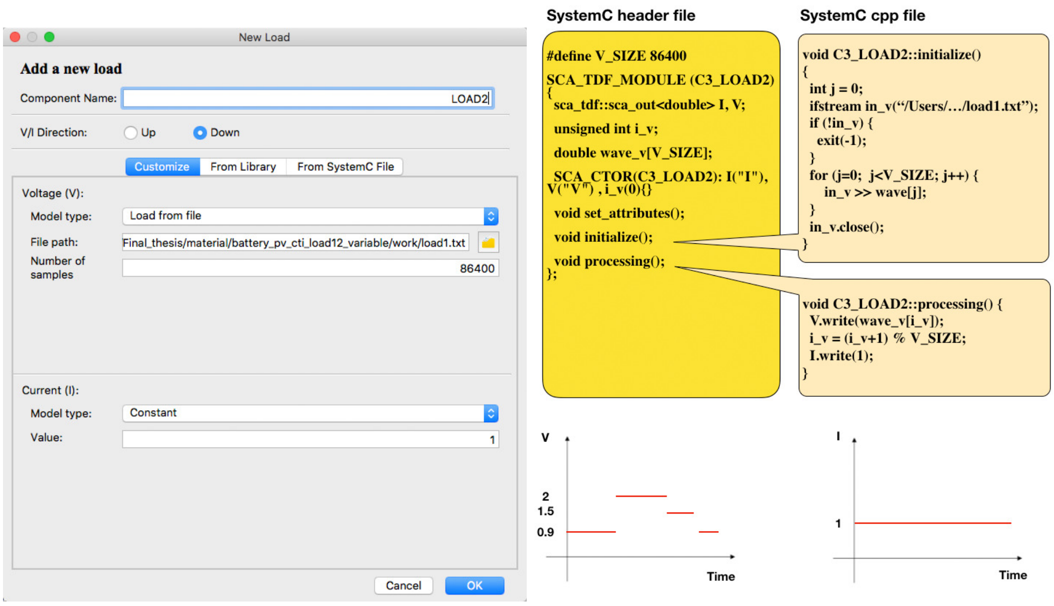
Figure 4.8 Create a constant output load component

In some cases, files which describe output trace is already available, *e.g.* , as output of a power simulator or of physical measurements on the device. . In this situation, users are able to import those files directly by the "Load from file" function. There are key parameters which user should give: trace file path and the number of samples. Corresponding output will perform as the value in trace file. In Figure 4.9a, there is an example load component with a constant current output and a voltage output load from a trace file. The generated code and trace figures are displayed in Figure 4.9b.

- *Sinusoidal*

  Sinusoidal wave is a typical load output waveform is EES design. So sinusoidal is supported by default in IDE I provided. To create a load component with sinusoidal output waveform, users need to choose *Model type* as "Sinusoidal" in the dialog for creating load component. There are there key parameters need to be configured: *maximum output value*, *minimum output value* and *number of samples*. Corresponding output will perform as a sinusoidal between *maximum output value* and *minimum output value* with *number of samples*. Figure 4.10 demonstrates the meaning of each parameter.

  The function of creating sinusoidal load is based on the function of creating load from trace file. It firstly creates a trace file with data for describing a sinusoidal wave.



(a) Dialog of creating a load with a trace file

(b) Generated code and output trace of the load in Figure 4.9a

Figure 4.9 Create a load component from trace file

Then load this trace file to design. In Figure 4.11, a load example with a constant 2V voltage output and a sinusoidal current output between 0A and 1A is demonstrated. The generated code, trace file and trace figures are displayed in Figure 4.11b.

- *PSM*

  A power state machine is a (Finite) State Machine whose states (called power states) represent the operating modes of a system as shown in Figure 4.12. For load components like microprocessors, power state machine is a normal approach provided for dynamic power management[6]. To model load components with PSM, users can choose model type as PSM in the dialog for creating load component. Key configuration parameters for PSM are number of states (N) and the output value of corresponding state. The corresponding output will act as a power state machine with N states with corresponding output value.

  In Figure 4.13, it describes an example load with the PSM shown in Figure 4.12. There are two power states which are active state with 2A/1V output and idle state with 1mA/10mV output.

- *Equation*

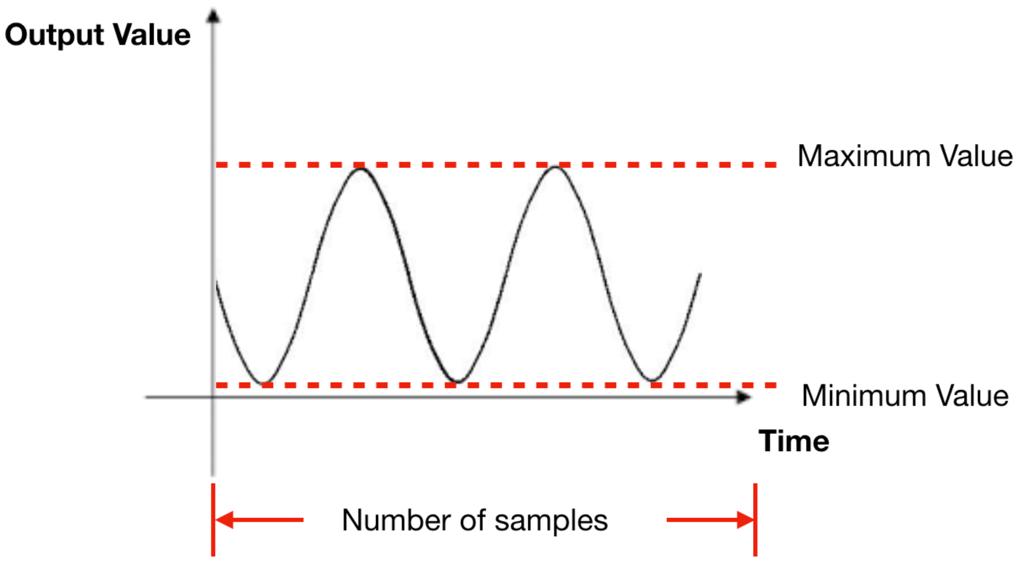


Figure 4.10 Key parameters of sinusoidal wave

For increasing flexibility, creating load component by equations is supported in this IDE. Seven kinds of math functions are supported:

- Linear expression: $y = ax + b$
- Sine expression: $y = asin(x) + b$
- Cosine expression: $y = acos(x) + b$
- Exponential expression: $y = ae^x + b$
- Natural logarithm expression: $y = aln(x) + b$
- Square: $y = ax^2 + b$
- Square root: $y = a\sqrt{x} + b$

By specifying number of samples and equation type, trace file for selected math function expression would be generated after clicking "Generate SystemC files" button. By loading that generated trace file, during simulation, the corresponding output will perform as a wave of selected equation type.

- *Distribution*

  It is also a normal case that workload of load component follows certain distribution. By giving number of samples, precision, min output value, max output value and distribution type, users are able to create a load which follows target distribution type. The corresponding output will be discrete values with selected distribution in the range between min output value and max output value. In Figure 4.15, it demonstrates a load component example with voltage output following tri-modal distribution with deviation 0.3V and three expectation which are 1V, 2V and 3V. The voltage output value distribution in generated trace file is shown in Figure 4.16.

Modify components

When users would like to modify component information, the operation is very simple users just need to double-click on any component they would like to modify in the operational area of graphics view in main window. A window like shown in figure4.17 will be displayed and users will thus be able to modify the component as they want.

Delete components

For the delete operation for EES components, users need to click "Delete component button" and select the component for removing from current design. By a single click, the selected component could be removed from current design as shown in figure4.18.

4.4.2 Operations related to connections

Once user have created a certain amount of components, there is one more step they need to do to finish constructing the desired system: inserting connections between components such that the program will know how these components are organized and the relationship between each components.

Create connections

There are two different approaches to add connections: the former is Fast connection which allows user to insert connections between two components in a rapid and friendly way, the latter is Normal connection which allows user to connect the connection point in each component one by one. It is less efficient than Fast connection approach but it allows more flexibility.

In order to demonstrate the difference between these two approaches, figure 4.19 shows a simple system instance with an ESD and a load. In the following part, we will take the system instance in figure 4.19 to show what is the different between Fast connection approach and Normal connection approach.

Fast connection

Fast connection offers an efficient way to establish connections between two components by drag-drop arrows. As shown in Figure 4.20, when users would like build connections between load and converter in Figure 4.20a, they need to click "fast connection button" then all components which are ready to connect to other component would be emphasized. Click on the load component and drag mouse, you will see an arrow appears under the mouse. In this time, only the component which current selected component could be connected to will be emphasized, *i.e.* in Figure 4.20b two converters are available. The last step to accomplish connection creating is to drag mouse to one connect-able component and release the mouse. The collection between load and converter is established as shown in Figure 4.20c.

Normal connection

Normal connection function is the first connection function I developed for this problem. Different from fast connection function, normal connection function connects ports of components instead of directly connects two components. It is less efficient and convenient than fast connection function. But it gives user more flexibility than fast connection function by the feature that connections are established between component port. In most cases, fast connection is the preferred approach.

Delete connections

When users make some mistake or change their mind on design, the delete connection function is required. There is a button between fast connection button and normal connection button which is delete connection button. By clicking this button, users are allowed to delete connections between components.

4.5 Library Management

With more and more people notice the efficiency and convenience of SystemC-AMS in energy simulation domain, the number of models built by SystemC-AMS is increasing. To get the benefit from this trend, an extendable library of some predefined EES models is introduced in my work. By assigning existing models to the components in user design from an library, designers are able to rapidly build their design. To maintain that library, a well-designed user interface is needed. In this section, the mechanism of library management will be introduced.

4.5.1 Mechanism

Inside the graphics interface, library management is accomplished by a library configuration file and files storing corresponding codes of each model. The library configuration file stores information about all the models in the library. In details, model information are separated by different kinds of EES components. For example, as shown in Figure 4.21, in the library configuration file there are 2 load models. The top level header file of one model is stored in path: `Library/load1_LOAD/variable_load1.h` and the other model top level header file is stored in path: `Library/load1_LOAD/variable_load2.h`. With the information in library configuration file, the main program is able to find the location where the code of the corresponding model is stored.

The codes and configuration files of one model are stored in the same directory with the directory name combined with model name and model type. As shown in Figure 4.22, there are four files inside the directory of the example model **load1**. Two of them, *i.e.* `info.txt` and `port_map.txt`, are model configuration files and the other two are model implementation files.

- Normally there are two **Model configuration files** for each model:
  - `info.txt` used for storing customized information of each model.
  - `port_map.txt` stores port map information of each model. As described in Chapter 3, each EES components has its own template. By this reason, port map need to be done when corresponding model is inserted into library, to map the actual ports of the SystemC implementation to the EES interface defined in Section 3.3. Figure 4.23 shows an example of port map file which is a simple load component with two out-direction ports.

Users are able to manage models in library by maintaining the content of library configuration file and files storing corresponding codes of each model. My program offers a user-friendly interface which allow users to manage library graphically.

4.5.2 Import model

When users would like to insert new models into library, they can enter "Library settings" function by clicking "Options" button and choosing "Library settings" tab. As shown in Figure 4.24, users are able to check library information including library quantity and already-imported models in current library.

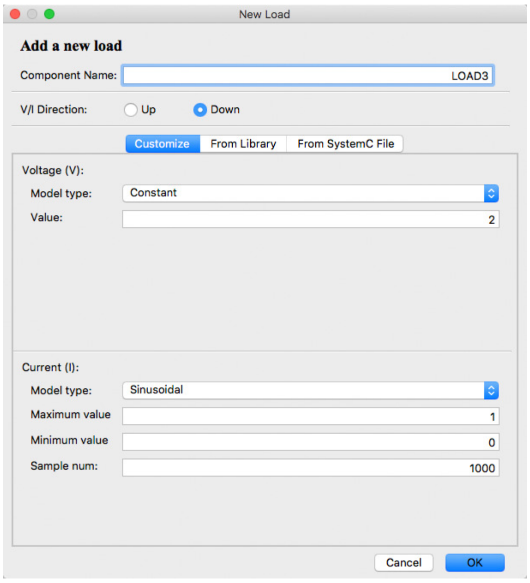
To import new model, click "Add new model" button. A new window used to collect model information appears. After inserting all the information about the new model, user need to manually map all the ports of that component. Then the new model will appear in "Library settings".

4.5.3 Modify model

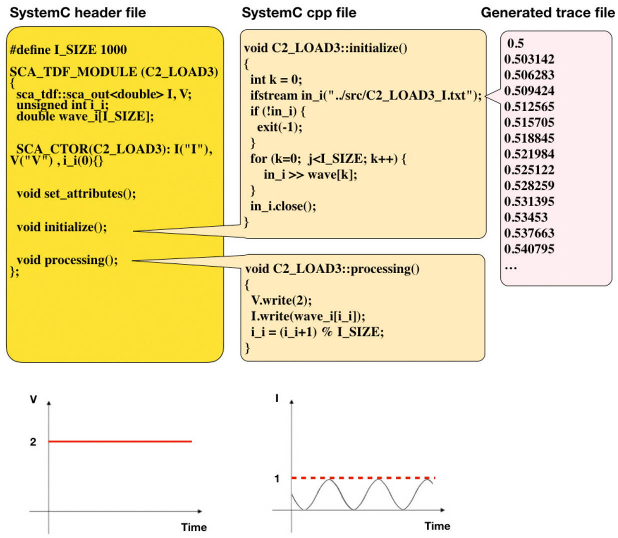
When modify certain model, user should select the model they want to modify first. Then click "Modify model" button. A window similar to the one we see during create the new model appears. User can make modification about this model as they want.

4.5.4 Delete model

When deleting a certain model, user need to choose the model they want to delete and click the button "Delete model", then corresponding model will be deleted from library configuration file and the model directory will be removed.



(a) Dialog of creating a sinusoidal load



(b) Generated code, trace file and output trace of the load in Figure 4.11a

Figure 4.11 Create a sinusoidal load component

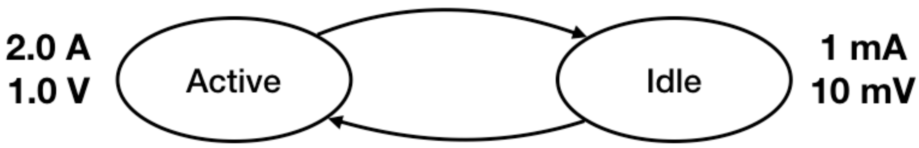
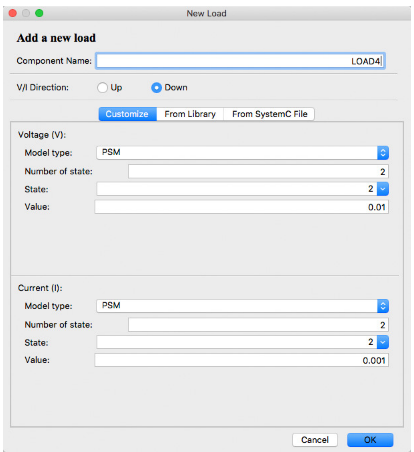
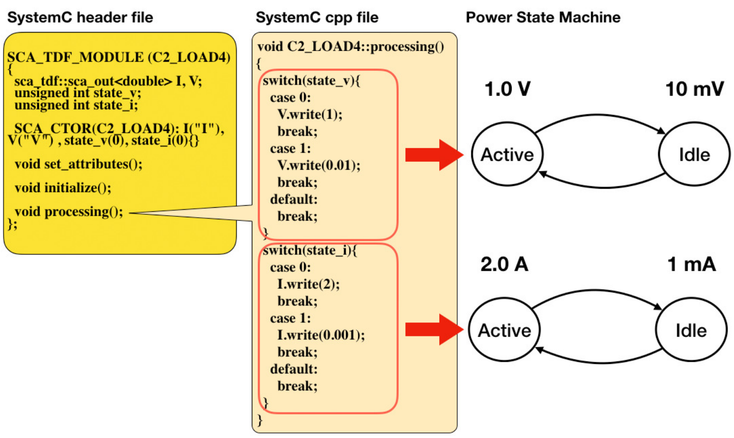


Figure 4.12 Power State Machine

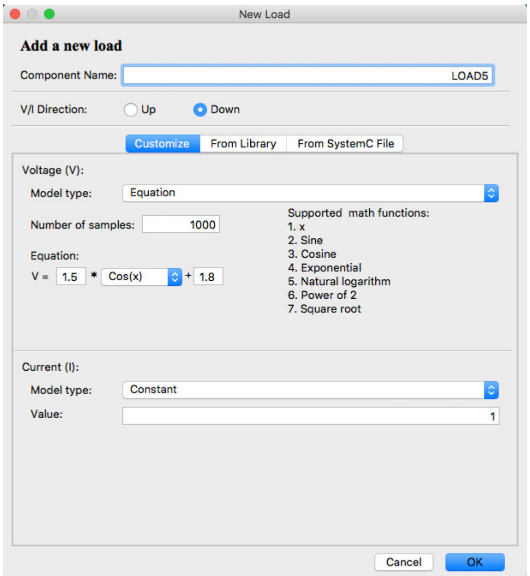


(a) Dialog of creating a load with PSM

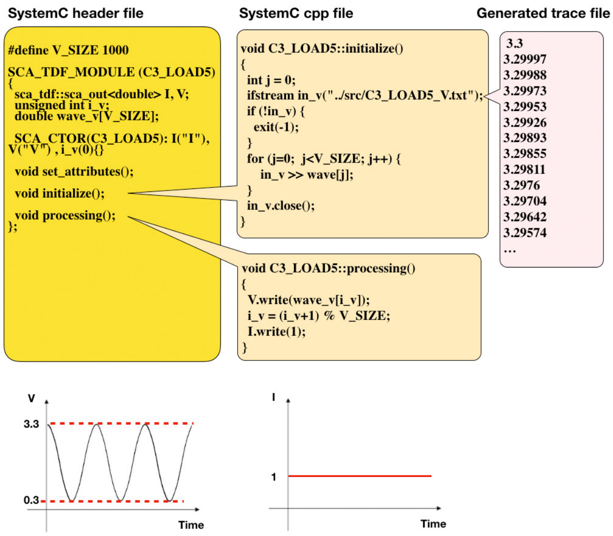


(b) Generated code of the load in Figure 4.13a

Figure 4.13 Create a load component with PSM

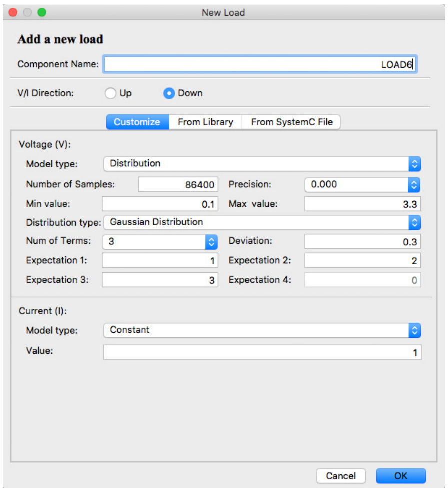


(a) Dialog of creating a load by cosine expression

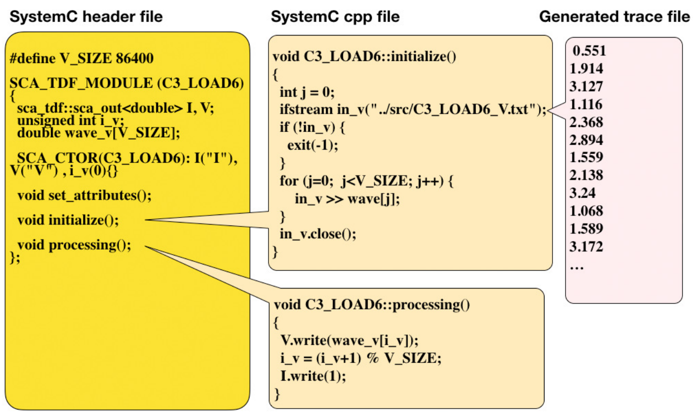


(b) Generated code of the load in Figure 4.14a

Figure 4.14 Create a load component by cosine expression



(a) Dialog for creating a load component with voltage output following tri-modal distribution



(b) Generated code of the load in Figure 4.15a

Figure 4.15 Create a load component with voltage output following tri-modal distribution

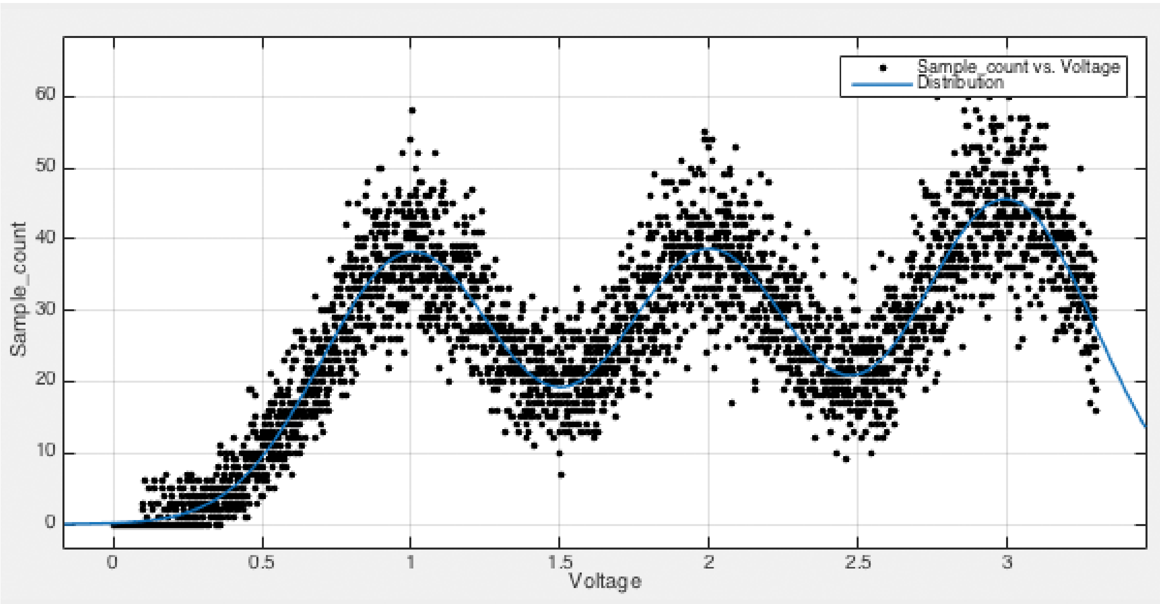


Figure 4.16 Voltage distribution of the load in Figure 4.15a

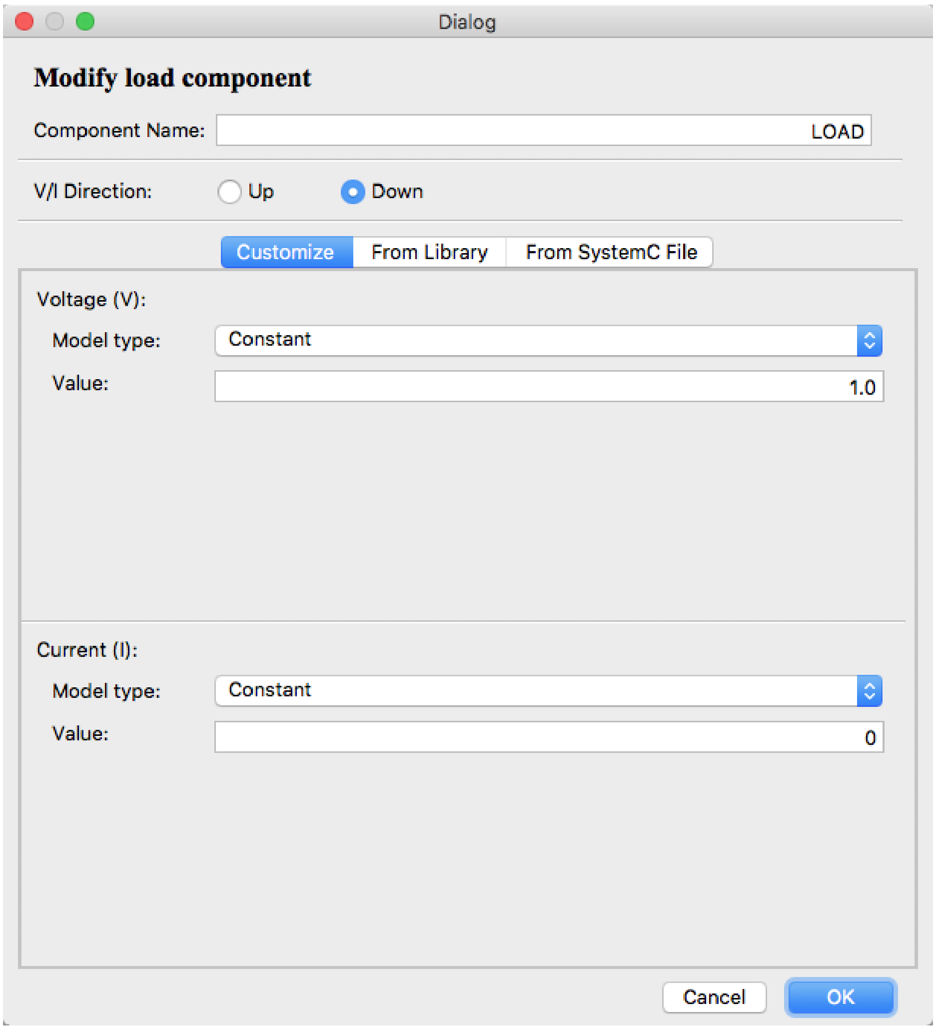
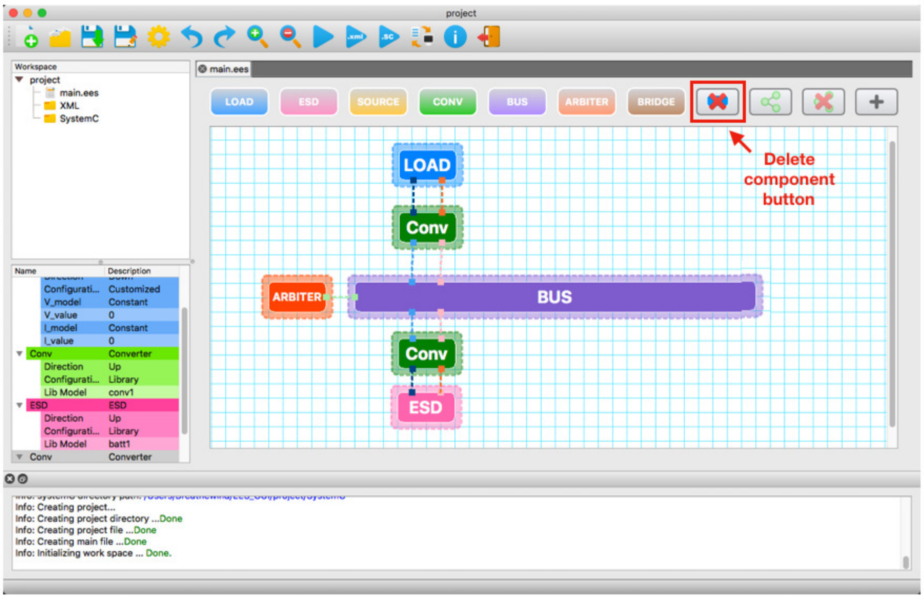
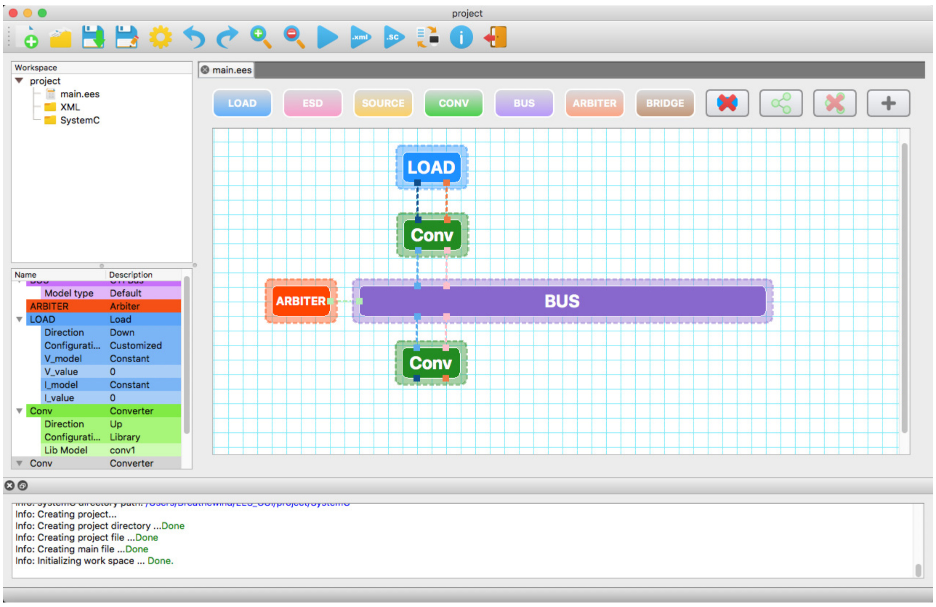


Figure 4.17 Window for modifying load component



(a) Click "Delete component button" and select the component for removing



(b) After deleting one ESD from current design

Figure 4.18 Delete one component from current design

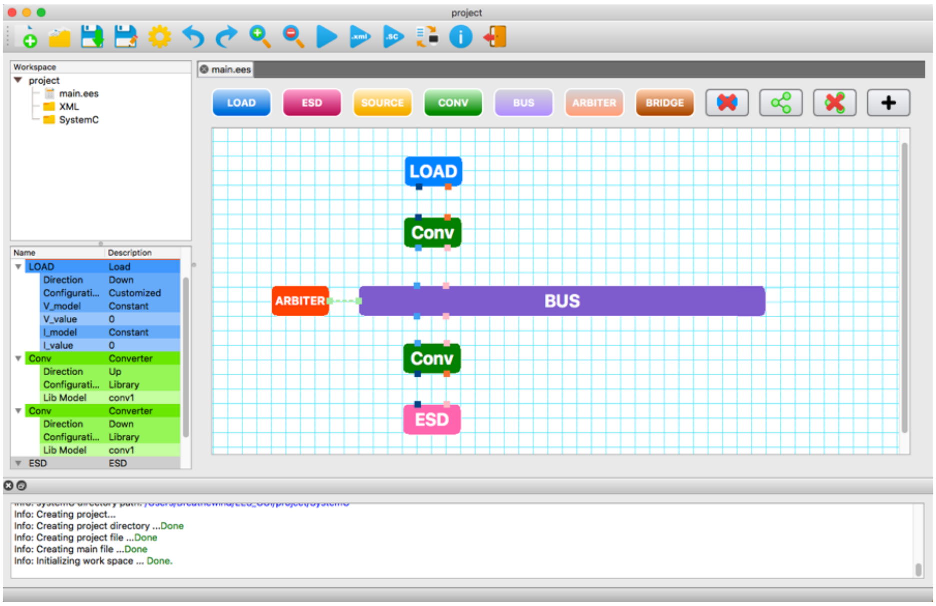
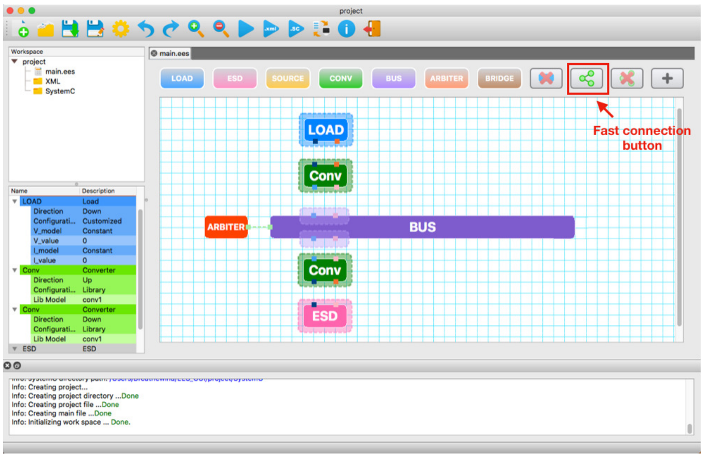
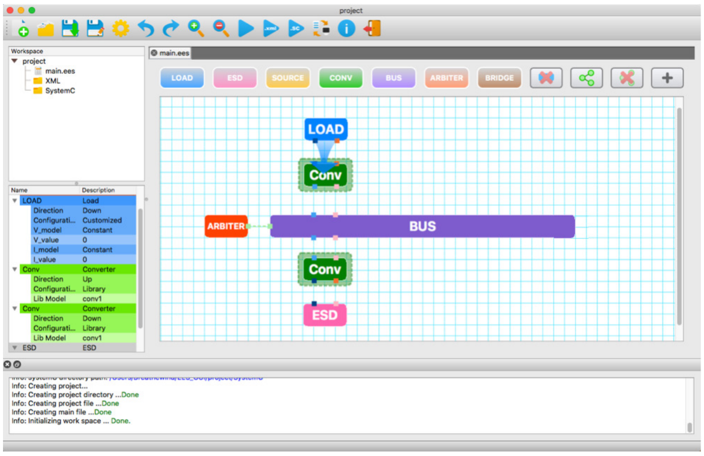


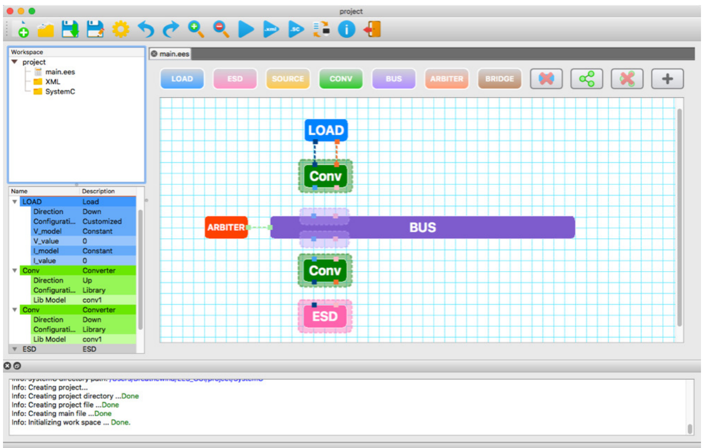
Figure 4.19 System Instance



(a) Fast connection button clicked and corresponding reminders are shown in operation area



(b) Drag the arrow from one component to another



(c) Release mouse and collections between two components are established

Figure 4.20 Create connection by fast connection function

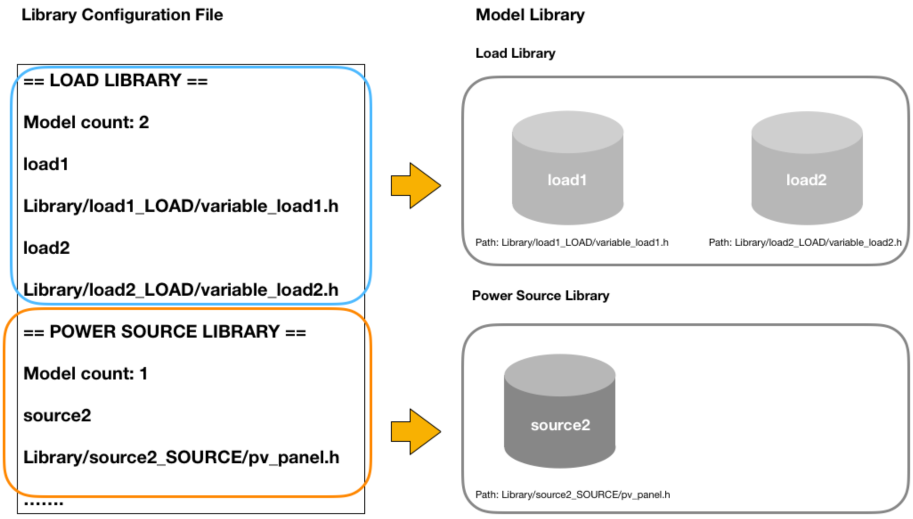


Figure 4.21 Library Configuration File Example

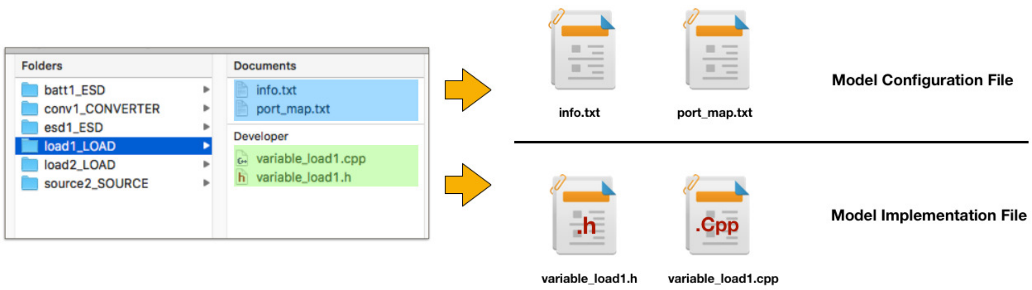


Figure 4.22 Model Files Example

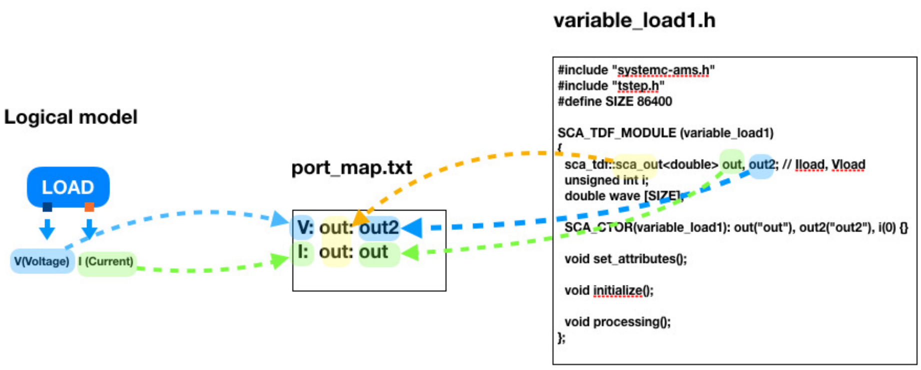


Figure 4.23 Port Map Example

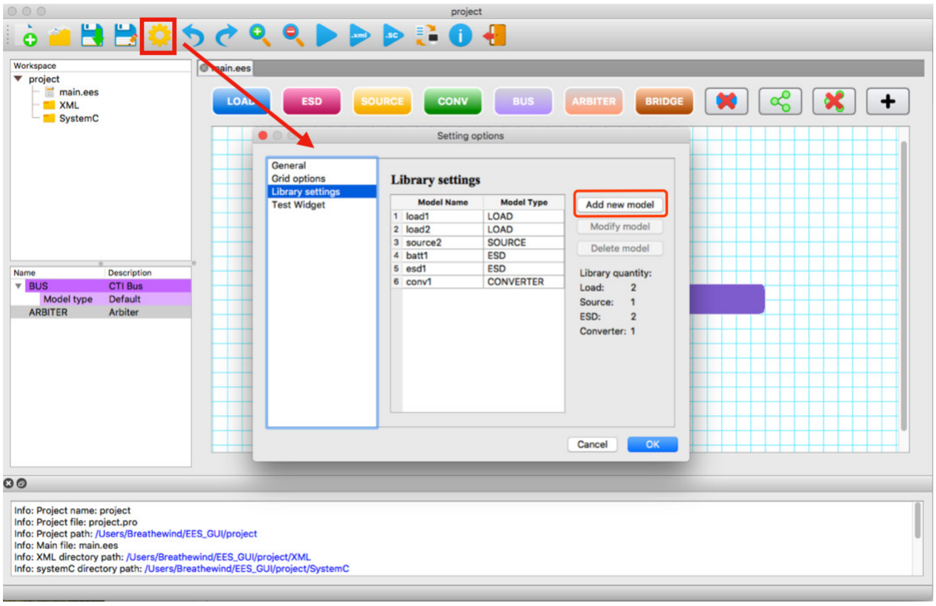
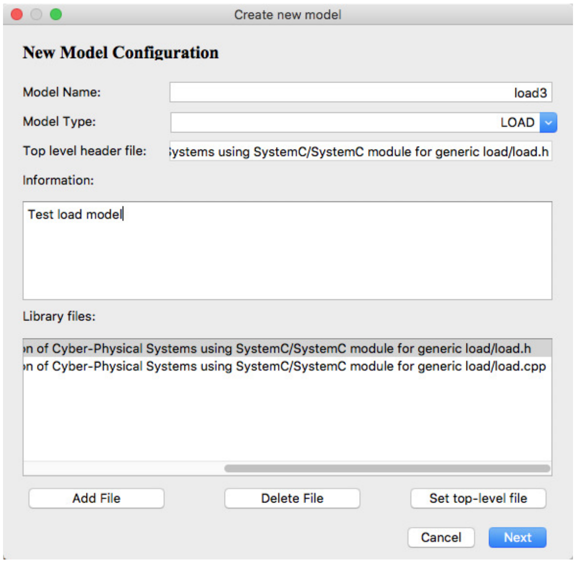
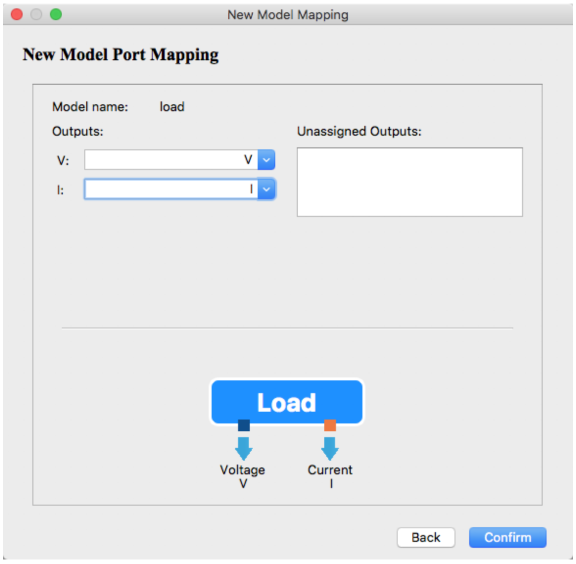


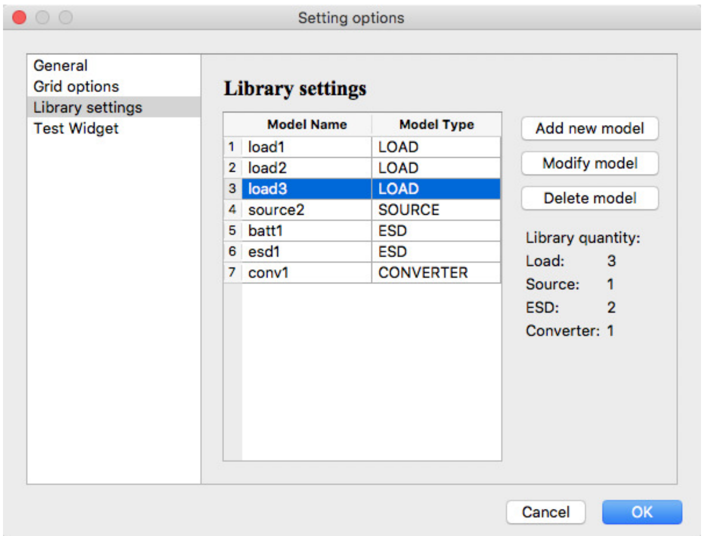
Figure 4.24 Enter "Library settings" function



(a) Import Model Step 1: Insert model information

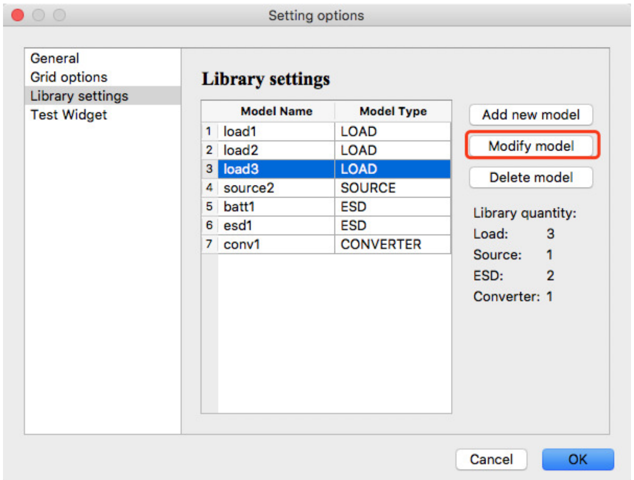


(b) Import Model Step 2: Port map

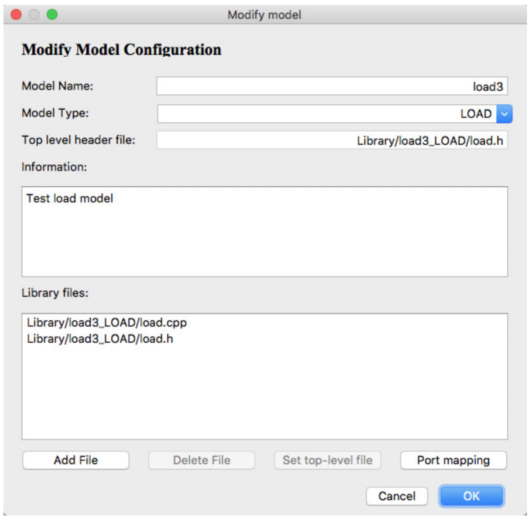


(c) Import Model Result

Figure 4.25 Import Model



(a) Modify Model Step 1: Select the model to be modified



(b) Modify Model Step 2: Modify model

Figure 4.26 Modify Model

Chapter 5

SystemC Code Generation

After finish building their design, users are able to generate SystemC code for future simulation by SystemC code generation function provided by IDE. In this chapter, I will consider a simple system shown in figure5.1 as an example to demonstrate the mechanism of code generation.

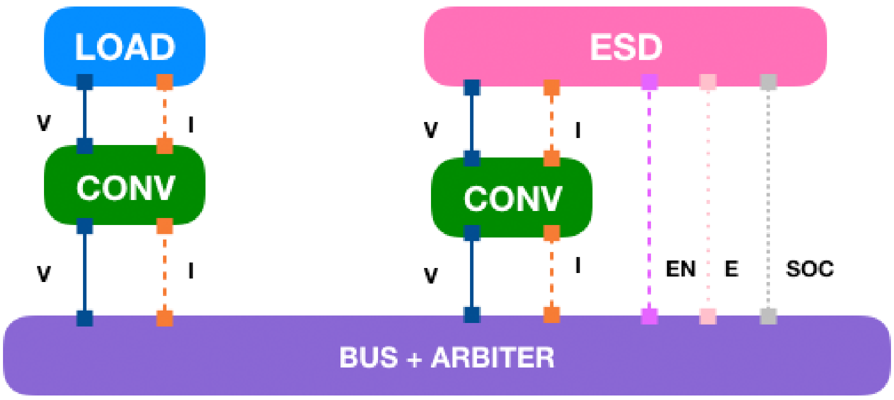


Figure 5.1 Example system for code generation

In Section 5.1, the data structure of storing component information will be introduced to give a general knowledge of how the data is organized and processed in IDE. Then the approach to collect information from data structure will be described in Section 5.3. With the knowledge of first two section, the mechanism of code generation will be demonstrated in Section 5.5. Finally, since the code generated is prepared for simulation, a brief introduction of SystemC-AMS will be giving in Section 5.4.

5.1 EES component implementation

Since the language adopted for developing this project is Qt which is based on C++, I could take advantage of objective oriented programming to define a class with attributes and methods of corresponding object. In data structure design of IDE, a class named *EES_Item* is defined to contain basic information of EES components, such as component name, component type, header file path, and methods to retrieve attributes and interact with other components.

EES_Item is the super class for all component classes. Specified components such as loads, power sources, and ESDs inherit from class *EES_Item* with some additional attributes and methods for specific usage of corresponding component as shown in Figure 5.2. In Figure5.3, it shows some example attributes in *EES_Item* class. These attributes will be used in following section to describe code generation of each component.

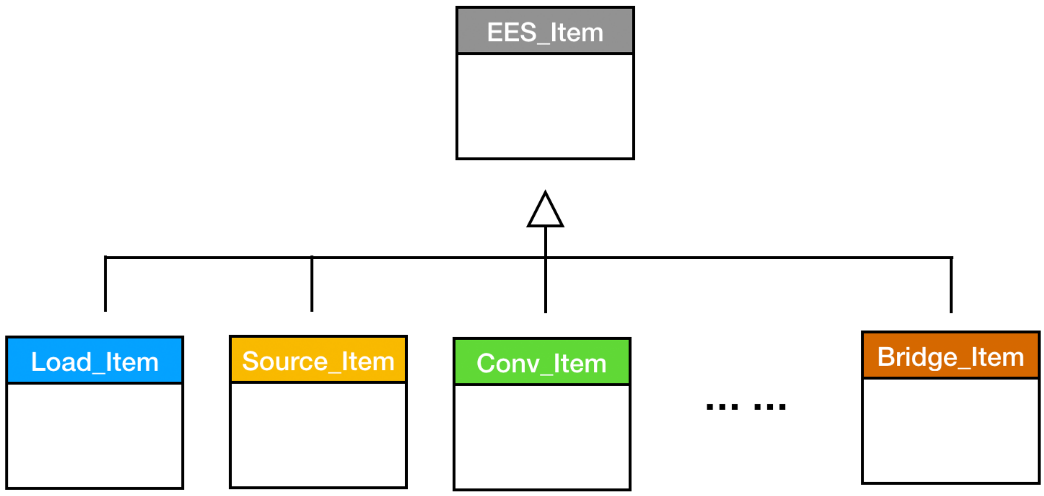


Figure 5.2 Relationship among classes

Seven kinds of EES component are gathered in three types in the view of data structure in my IDE, which are functional components, adapter components and connection components. Functional components are the components which perform functional effect on power, *e.g.* loads consume power, power sources provide power. Among EES components, load, ESD and power source are functional components. Adapter components containing two kinds of component, *i.e.* converter and bridge, are the components adapt voltage between components. Different from previous two types are real electronic elements in design, connection components comprise two logical components, *i.e.* arbiter and bus, which are logical components in design as described in Chapter 3.

5.2 Connection implementation

In Figure 5.3, it shows a general view how *EES_Item* class is inherited by specific components, *i.e. Load_Item*, *ESD_Item* and *Conv_Item* in the figure. Comparing with super class *EES_Item*, attributes related to connection information is extended in these component classes. For example, in class *Load_Item* and *ESD_Item*, an attribute named *Conv_id* which indicates the id of the converter which it connects to is extended. An attribute named *bus_port_id* used to describe port numbers on CTI bus is extended in class *ESD_Item* and *Conv_Item*. Finally, in class *Conv_Item*, there exists an attribute which can be only found in *Conv_Item* named *Component_id* which is used for identifying the functional component connected to it. These attributes will be used for collecting connection information during code generation phase.

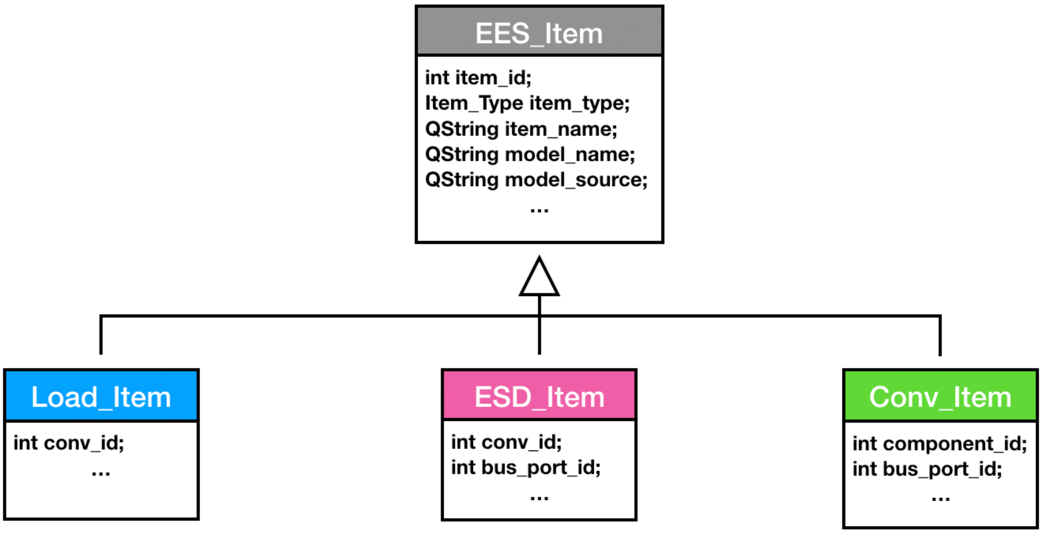


Figure 5.3 Data structure used for storing component and connection information

5.3 Information collection

To generate SystemC code, component and connection information stored in corresponding attributes need to be collected. Methods for collecting information are developed in corresponding classes to help retrieving attributes. During code generation, IDE invokes methods from each component class to collect information. In figure 5.4, it shows an example on information collection from a EES component class.

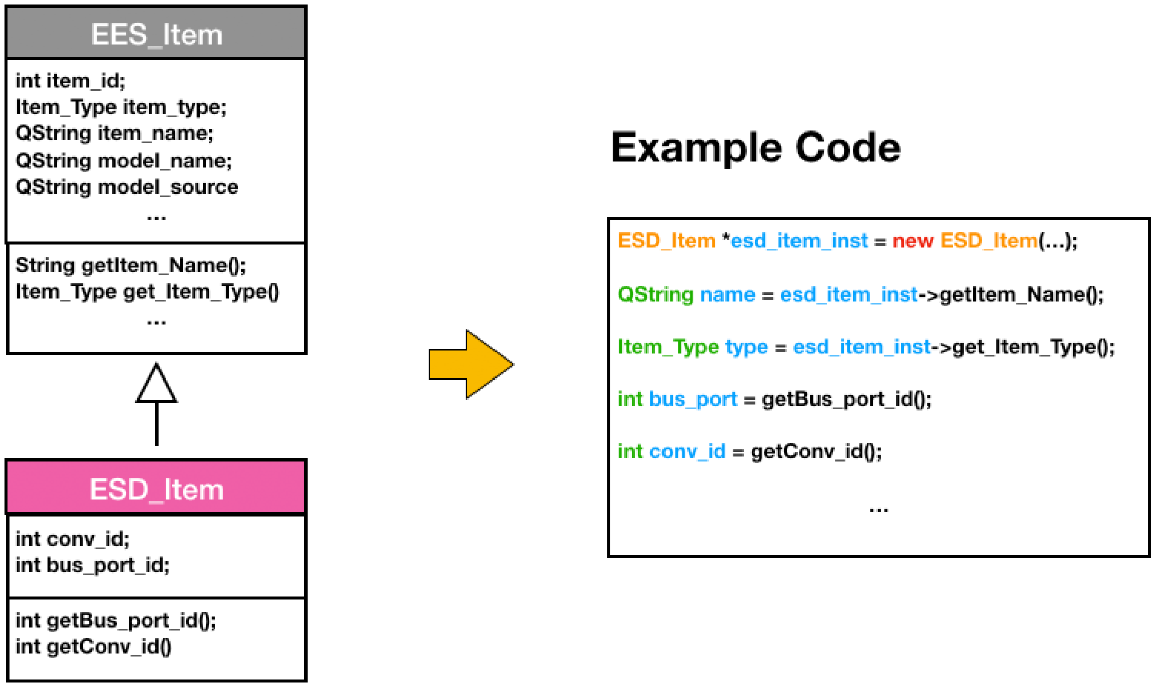


Figure 5.4 Example on information collecting

5.4 Power Simulation with SystemC-AMS

The implementation of the power perspective of a cyber-physical system requires the simultaneous simulation of models at different levels of abstraction, *e.g.*, waveforms and circuit-equivalent models. This may impact on synchronization: low level models may introduce too many synchronization points, and it would thus be difficult to determine a correct synchronization mechanism with higher level models. To this extent, we adopted the TDF level of abstraction for all interfaces and connections. This implies that synchronization between components happens at predefined fixed time steps, *e.g.* , once every millisecond (of simulated time). On the contrary, component models can be implemented with the most suitable level of abstraction, ranging from SystemC TLM up to SystemC-AMS TDF, or ELN itself. This allows to determine a good tradeoff between accuracy of the model and effectiveness of the synchronization mechanism.

As presented in Chapter 3, there are two main kinds of models usually used during EES design. One is circuit model, which can be constructed by connecting ELN blocks. Another is functional models, which can be implemented as TDF functions.

To this extent, each EES component is implemented as a SystemC module (*i.e.* a instance of `SC_MODULE`) if the component model is circuit model, to leave freedom to adopt any level of abstraction. If the component model is not circuit model, SystemC-AMS TDF module (*i.e.*

a instances of `SCA_TDF_MODULE`) would be adopted to implement that component. Functional and analytical models are implemented as TDF modules (`SCA_TDF_MODULE`) to exploit the efficient scheduling of TDF. The interface adopts TDF ports (`sca_tdf::sca_in` and `sca_tdf::sca_out`), usually of type double, to make overall simulation more efficient, and to enforce an efficient interaction with components at any level of abstraction. An example of component implementation is provided in Figure 5.5, that shows an excerpt of code for the implementation of a battery.

Component evolution is handled differently, depending on the kind of adopted model. Figure 5.5 outlines this idea by comparing two different implementations of a battery. In case of functional models (*e.g.*, waveforms, state machines or equations), the evolution is handled by the processing() function as a SystemC process, executed at fixed time steps. As an example, the left-hand side of Figure 5.5 models battery dynamics with Peukert's model. If else the adopted models is at circuit-level, it is implemented by describing the circuit as a network of SystemC-AMS ELN components, instantiated and connected in a way that reproduces the circuit specification (right-hand side of Figure 5.5). Wrapping the ELN subsystem through ELN-TDF converters allows to preserve synchronization with the rest of the system.

Figure 5.5 allows to highlight some of the advantages produced by the adoption of SystemC-AMS as a target language. First of all, a single language allows to cover two very *different levels of abstraction*, *i.e.*, functional and circuit level. Different types of models can thus be simulated simultaneously, leaving the interaction to the underlying simulation kernel. This is advantageous *w.r.t.* co-simulation frameworks.

Furthermore, SystemC-AMS separates the implementation of each component's interface and behavior. This *modularity* allows to preserve the interface when varying the adopted model. In the Figure, both implementations of the battery have the same interface (lines 1–5), even if the implemented models are at very different levels of details. This allows to adopt different implementations for the same component, depending on the target (*i.e.*, accuracy or performance), or with the goal of comparing their behavior and characteristics, without affecting the connection to other system components.

Finally, SystemC-AMS modules can be easily *configured and reused* for modeling components with different characteristics. As an example, the SystemC-AMS module can be adopted for modeling two different batteries by setting the different capacity level and all the circuit parameters according to the methodology in Section 3.7. Thus, the proposed methodology can be enhanced with the definition of a library of models for the components, that can be easily instantiated and configured at later times.

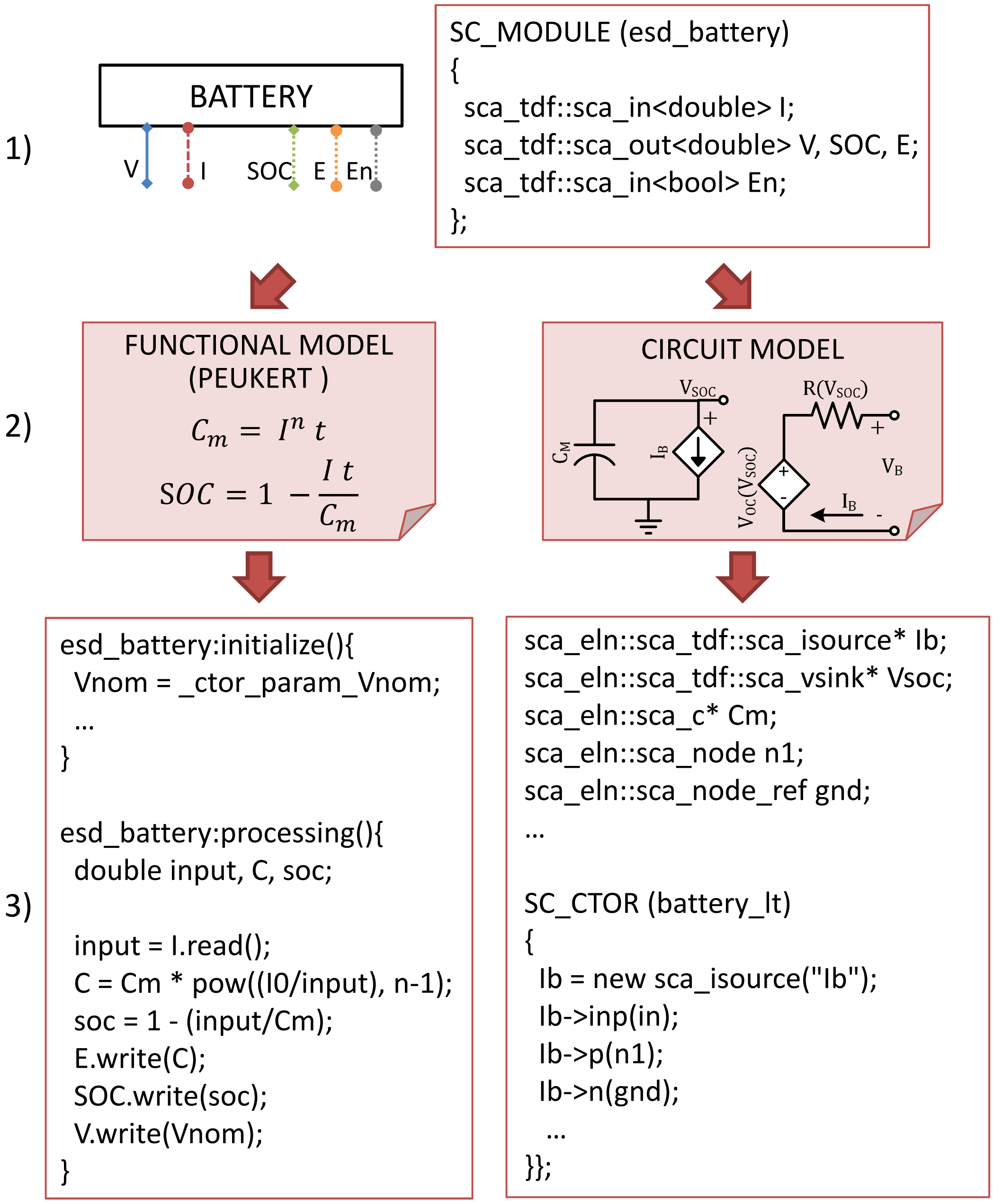


Figure 5.5 Example of SystemC-AMS Implementation a Battery by Adopting Two Different Models: Peukert's Law (left) and a Circuit-Equivalent Model(right).

5.5 Code Generation

Code generation for entire system consists of two sub code generation phase: component code generation phase and connection code generation phase.

During component code generation phase, the program retrieves information from component classes and recognizes corresponding component type. For example, a power source instance is generated by certain model in library. During component code generation, program invokes methods used to collect component information and retrieves component information from that power source instance. Then program knows this source power instance is implemented by certain model in library. All files need to describe this model will be imported to current design and in the generated top-level main file of current project an instance of that power source model will be instantiated.

After all components in current design are imported, the program then invokes methods to collect connection information. By the component connection attributes described in Section 5.1 and Section 5.3, topology information will be retrieved and included into the generated top-level main file.

Let us consider a simple system as shown in Figure 5.6 with one load, one ESD and two converters as an example to demonstrate the mechanism of code generation.

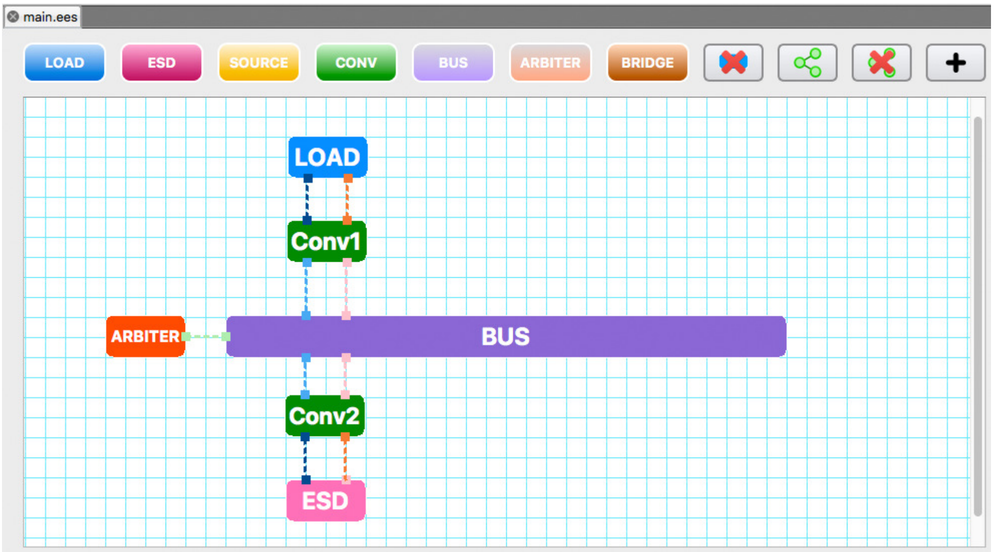


Figure 5.6 Design of a simple system with one load, one ESD and two converters

In the creating process of all functional components and corresponding code generation examples are demonstrated. The creating process of all functional components and corresponding code generation examples are demonstrated in section 4.4.2. Figure 5.7, Figure 5.8 and Figure 5.9 show the generated top-level header files for functional components and adapter components in this design. In Figure 5.7, there is a load with two constant outputs: voltage output on 2V and current output on 1A. IDE reads the name, model type and corresponding voltage and current value of that load by methods provided by Load_Item class. Then, the IDE processes the retrieved information to generate code for that load component.

In Figure 5.8, it shows that both of two converters in this design are implemented by a switching converter model in model library. For simplicity, only top-level header file is

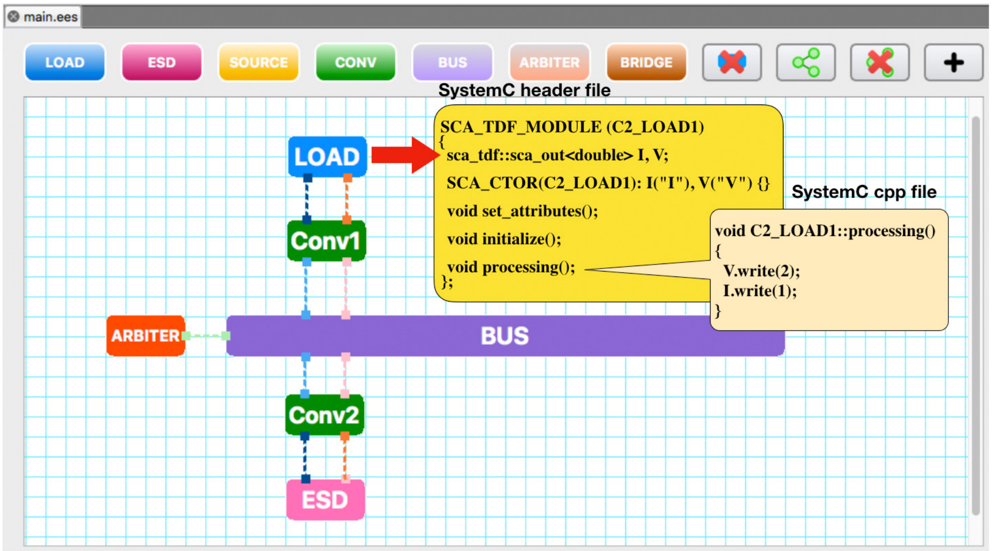


Figure 5.7 Code generation of load in example system

displayed in that figure. The ESD in this design is implemented by a Li-ion rechargeable battery model from model library. Its generated top-level header file is shown in Figure 5.9. The mechanism of the code generation of converters and ESD in this design is more or less the same. As described in section 4.5, the code of target models of each EES component will be copied from model library to current project. IDE creates an instance of corresponding model for each component in generated main file. By this approach, it can be guaranteed that implementation of created component will coincide with that in model library.

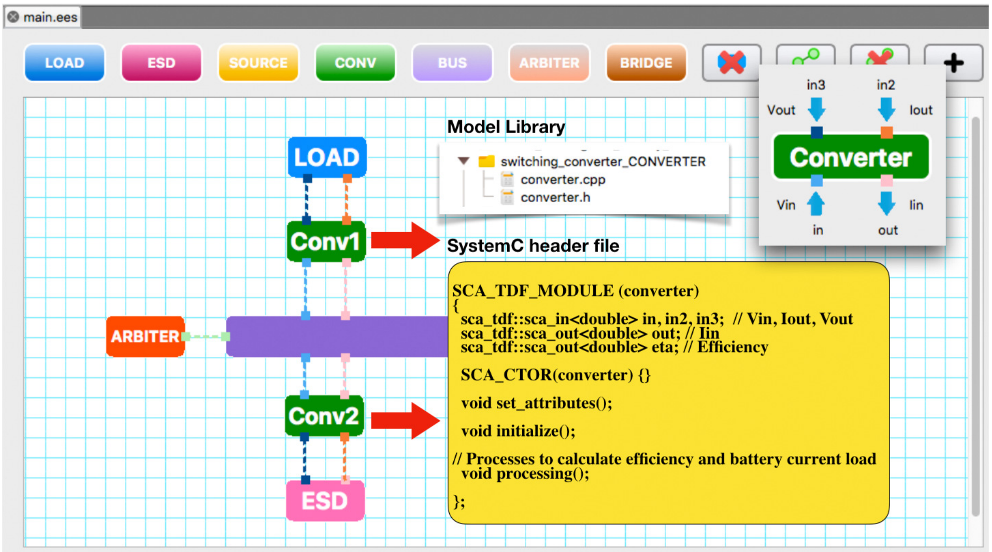


Figure 5.8 Code generation of converters in example system

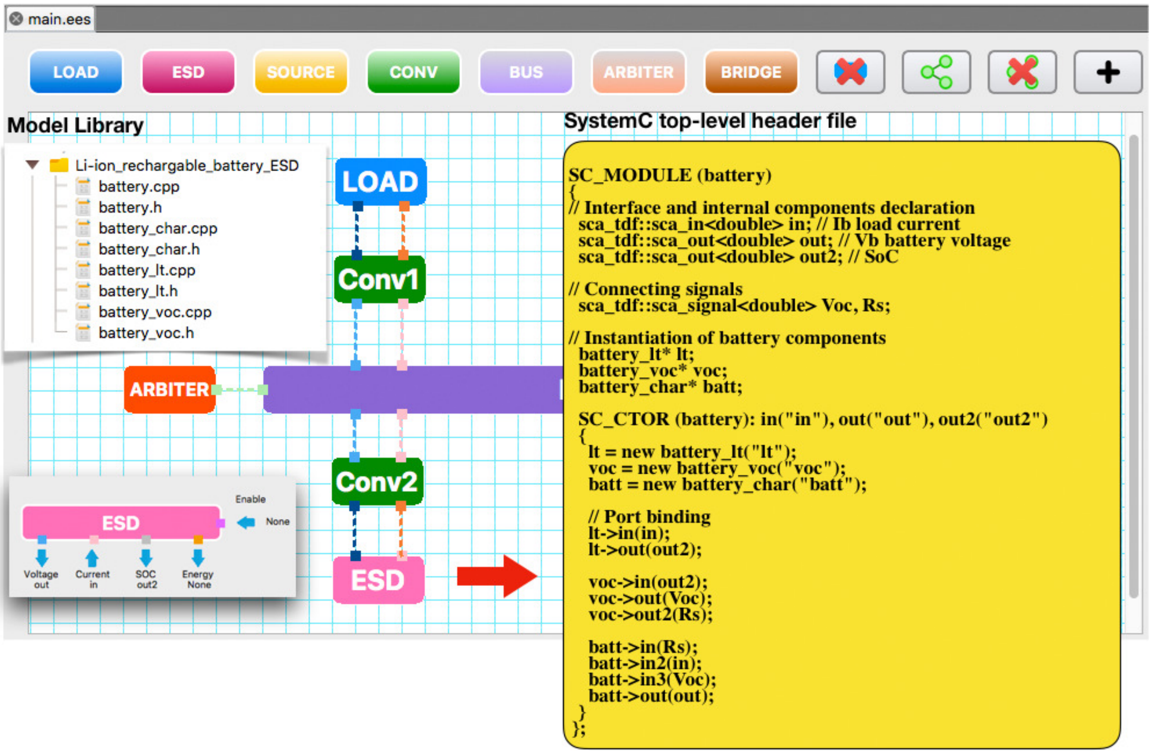


Figure 5.9 Code generation of ESD in example system

All the works done in Section 4.4.1, *i.e.* operations related to components, will be adopted for collecting information for components. In Section 4.4.2, it describe operations related to connection. All information collected in this part will concluded as connection information and processed to a cpp file. For each project, a top-level main file named main.cpp will be generated. It works in a higher level than other header and cpp files. It contains the topology information about entire design rather than model information in other files.

The file main.cpp consists of three main parts: included files code generation, component instantiation and port binding.

- During included files code generation, IDE generates codes for including required libraries and component header files into current design. As described in Section 5.1, all of the seven kinds EES component class inherit from a super class EES_Item. In that class, there is an attribute which stores the information of the header file of corresponding EES file. This attribute will be configured when corresponding component is created or modified. During code generation phase, IDE reads the information of this attribute and processes it to fit main file format.

- IDE instantiates all the components in current design during component instantiation phase. In EES_Item class, there are also attributes storing the name of corresponding model class and EES component name. Similar as what IDE dose during included files code generation phase, it reads attributes from each component and processes it with respect to main file format.

- The last phase of code generation is the port binding phase which describes the topology information of current design. The topology information stored between different kinds of EES components is sightly different depending on role of component. Functional components such as loads, ESDs and power sources, there is only information about the converter they connected to stored in class. For converters, since they are connected to both bus and functional components, in the class of converters both the component id of the function component it connected to and the the bus port id will be stored. In bus class, there is a bus port id in for each port such that IDE could identify different bus ports. When generating the codes for port mapping, IDE will read topology information from each component class and process to fit SystemC-AMS syntax.

After these three steps described above, main.cpp will be generated. Now generated codes are ready for simulation. In Figure 5.10, there is the generated main.cpp code of the example mentioned in this section.

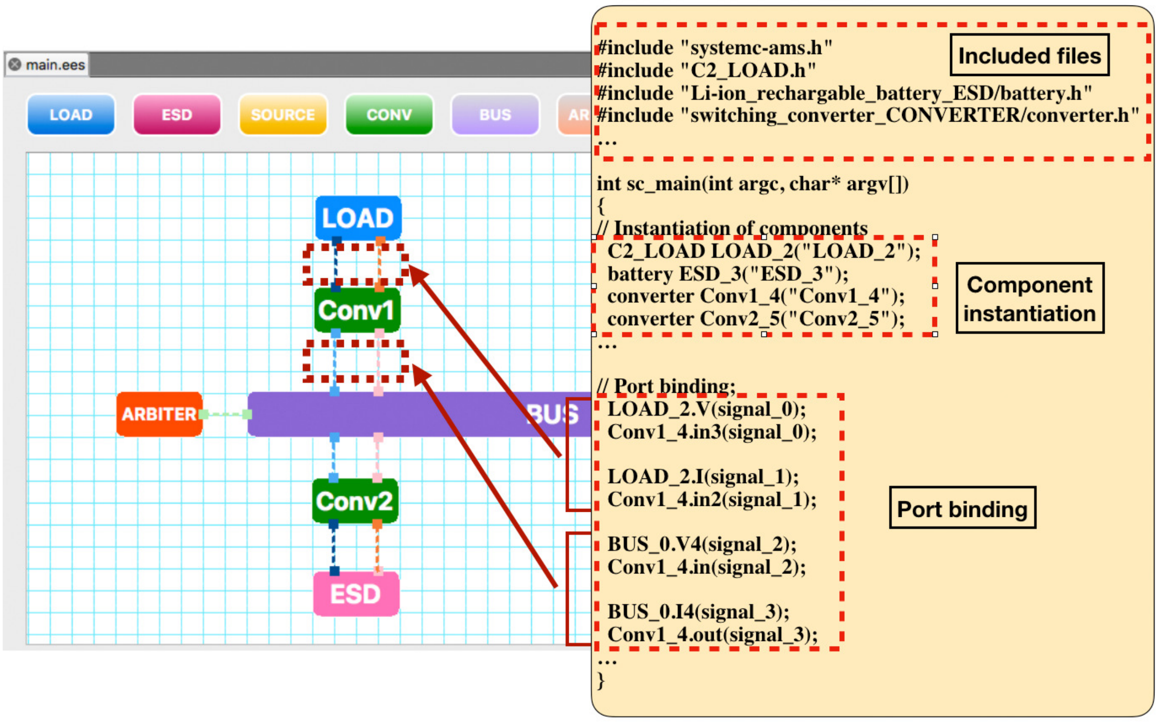


Figure 5.10 Code generation of main.cpp in example system

Chapter 6

Application to EES case studies

In this chapter, three cases will be introduced. Each example contains one customized EES system design. The design of each example including component and connection information will be firstly described. Then, how corresponding design could be implemented graphically by my program will be demonstrated. Thirdly, I will show the code generated by the code generation function introduced in previous chapter. Finally, the results of simulation will be reported to prove the correctness of the code generated for the corresponding designs with all the previous steps.

6.1 Case study 1

In this section a design with two loads, a photo-voltaic panel , a battery and four converters will be considered. They are organized as shown in Figure 6.1. In details, the design is composed of:

- a Li-ion rechargeable battery by Qinetiq (capacity of 5.8Ah, nominal voltage of 3.69V) modeled as in [14];
- a power source, *i.e.*, a photovoltaic (PV) panel composed of 5 Sunpower A300 PV cells connected to a module performing maximum power transfer tracking (MPTT) [8];
- two load devices;
- four DC-DC converters modeled as in [13];
- a CTI bus modeled as an ideal current conductor with constant reference voltage of 3.0V.

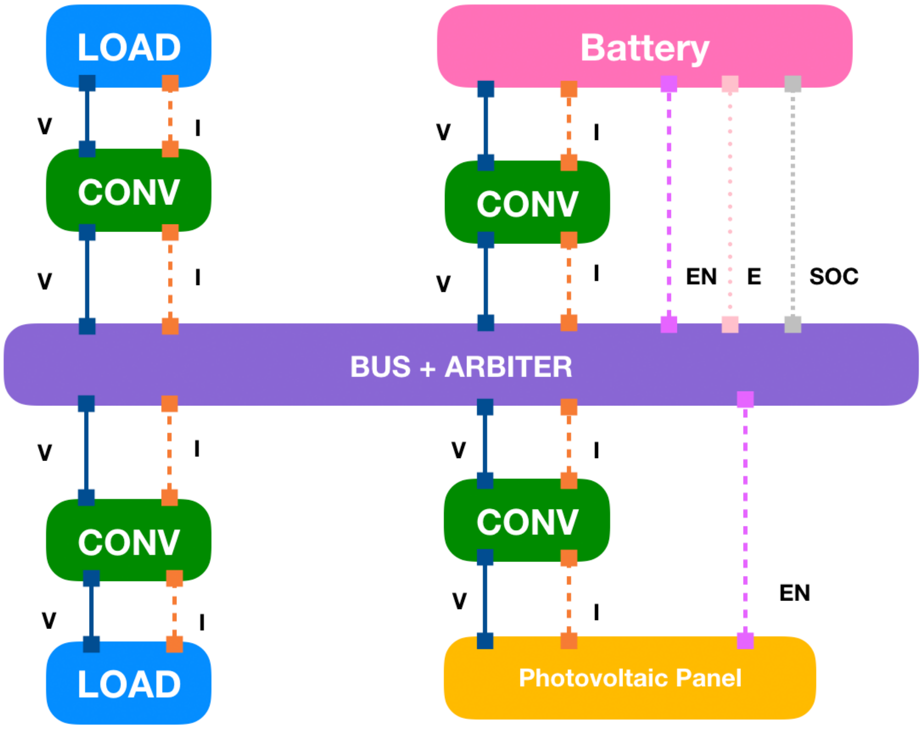


Figure 6.1 Design of case study 1

The CTI arbiter is augmented with an implementation of a “charge allocation” policy, which is as follows: As long as the power drawn from the PV panel satisfies the power demand of the loads, loads are supplied by the power source. Otherwise, the loads are supplied by the battery, until the SOc is below 10%. When SOC is below 10%, the entire system terminates. Although this policy is quite simplistic, the purpose here is not to develop sophisticated policies but rather to show the availability of the my program.

6.1.1 Implementation

In this section, I will demonstrate how to build the system described above step by step using my IDE.

Create a new project

Firstly, a new project need to be created as shown in Figure 6.2. User need to fill in information about project, such as project name, project path, main file name and main file path. Notice that main file here is not main.cpp introduced in Section 5.5. It is the file stores information about current design with a file name "main.ees". With these information, program is able to locate the new project. After "OK" button clicked, program is prepared for creating new design as shown in Figure 6.3.

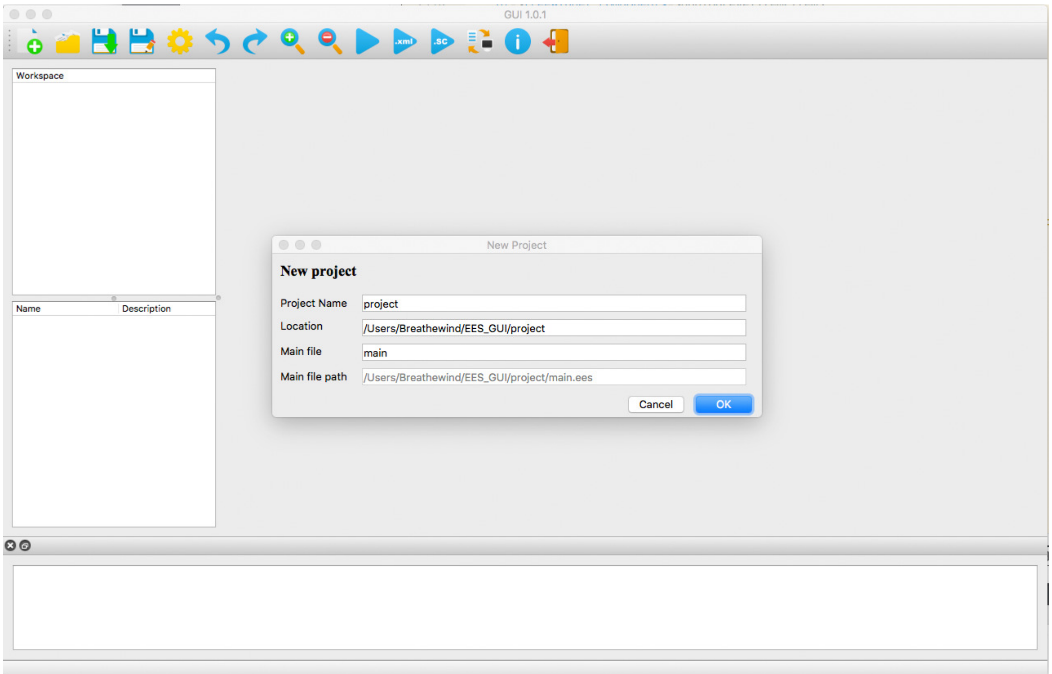


Figure 6.2 Case study 1: Creating new project

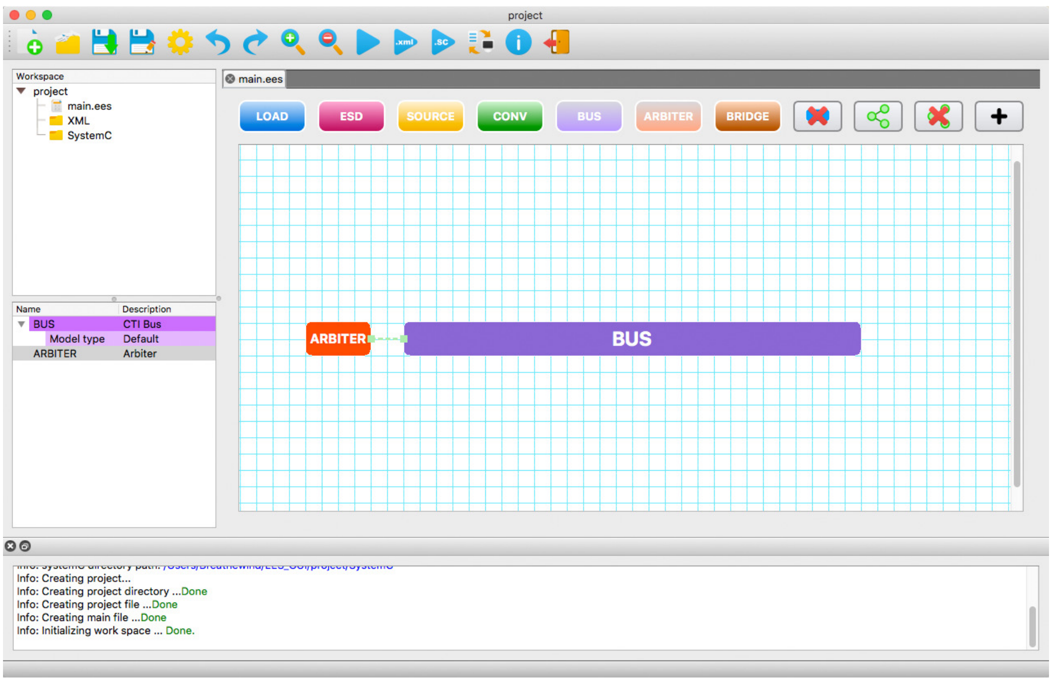
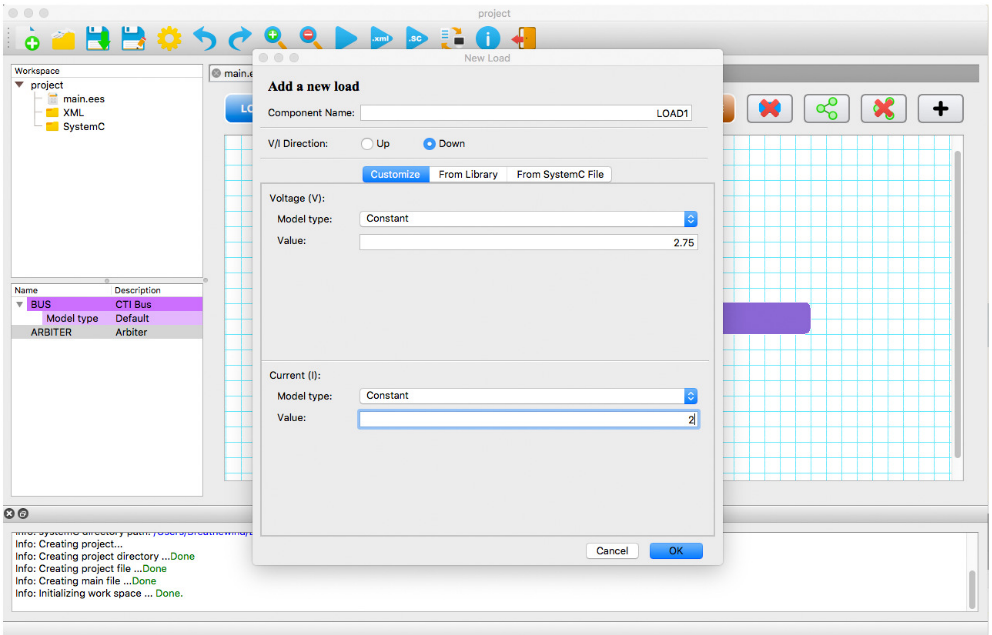


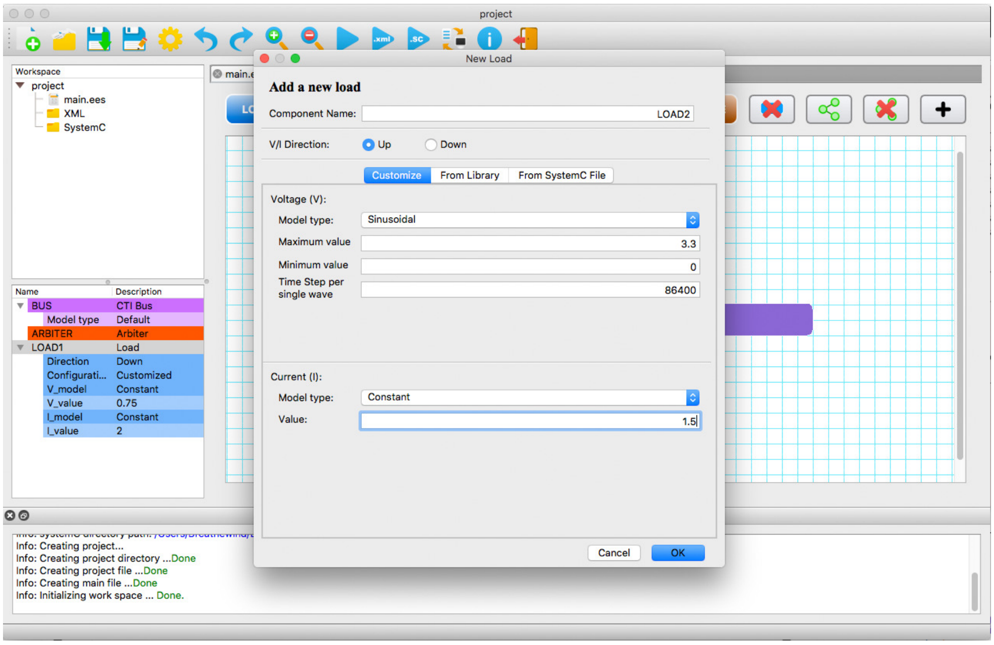
Figure 6.3 Case study 1: Main window of program

Create components

Next step is creating components of user customized design. Users can create components they need in any order as they want. In this case, load components will be created first.



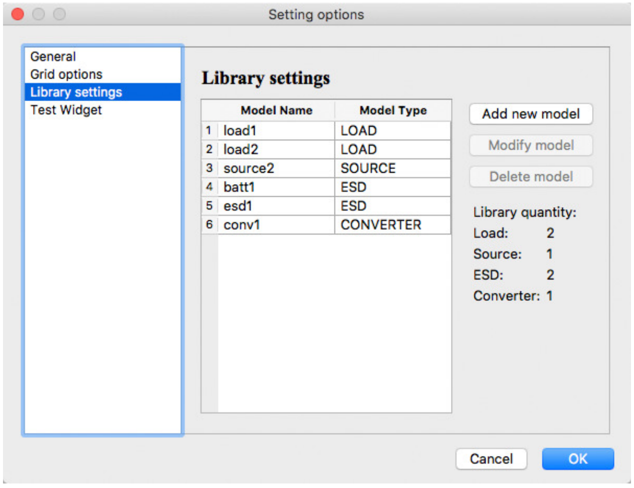
(a) Case study 1: Creating model for the first load component



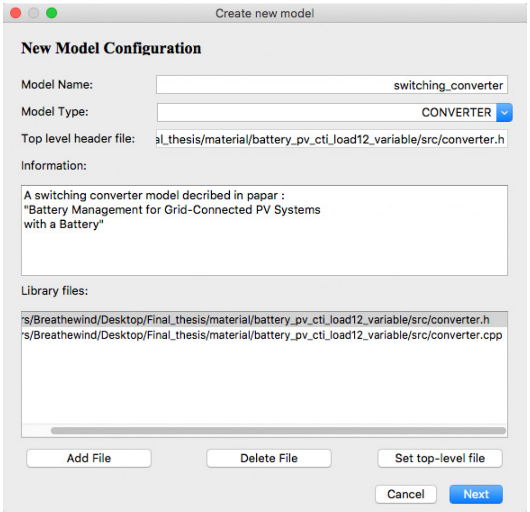
(b) Case study 1: Creating model for the second load component

Figure 6.4 Case study 1: Creating model for load components

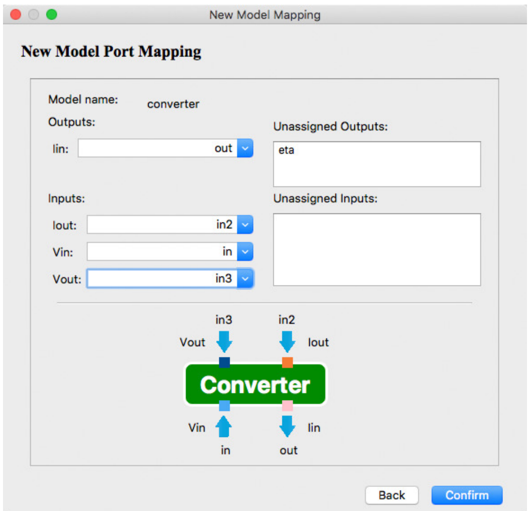
There are two load components in this case. To demonstrate the function for creating load components well, two different model creation approaches are used. For the first load component, *i.e.* the load component named "LOAD1" in Figure 6.4a, a constant model with 2.75V and 2A output will be selected as shown in Figure 6.4a. LOAD2 is created following a model with a sinusoidal waveform voltage output between 0V to 3.3V and a 1.5A constant current output as shown in Figure 6.4b.



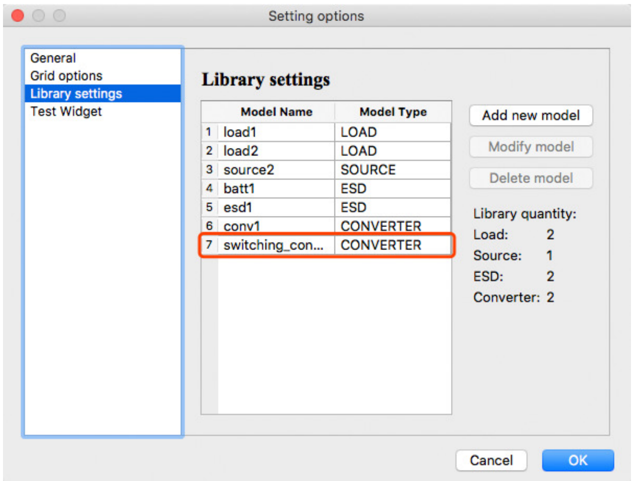
(a) Case study 1: Model library before import switching converter model



(b) Case study 1: Import model to library - basic information



(c) Case study 1: Import model to library - port mapping



(d) Case study 1: Model library after importing new model

Figure 6.5 Case study 1: Import switching converter model into model library

After load components, converters are the next component to be created. In case study 1, converters are switching converters modeled as in [13]. To demonstrate more functionality of IDE, I assume that there is no such model in IDE model library. By this reason, before creating converter components, the model of switching converters should be imported into model library first. Figure 6.5 shows the process how a new switching converter model is imported to model library.

The first step is to open setting option dialog from main window as shown in Figure 6.5a. We can see there is no switching converter in current model library. To import that model, click "Add new model" button in this dialog. A dialog for importing new model to library as shown in Figure 6.5b will appear. After inserting all the basic information, such as model name, model type, top level header file and etc, click "Next button". Then the dialog for port mapping appears as shown in Figure 6.5c, all the unassigned ports are displayed in the right side field. Users are able to map them to the EES template of converter component in the left side. After all work done, click "Confirm" button. As shown in Figure 6.5d, now in the setting dialog, we could see a model named "switching_converter" is already there ready for being selected during the process of creating new converter components.

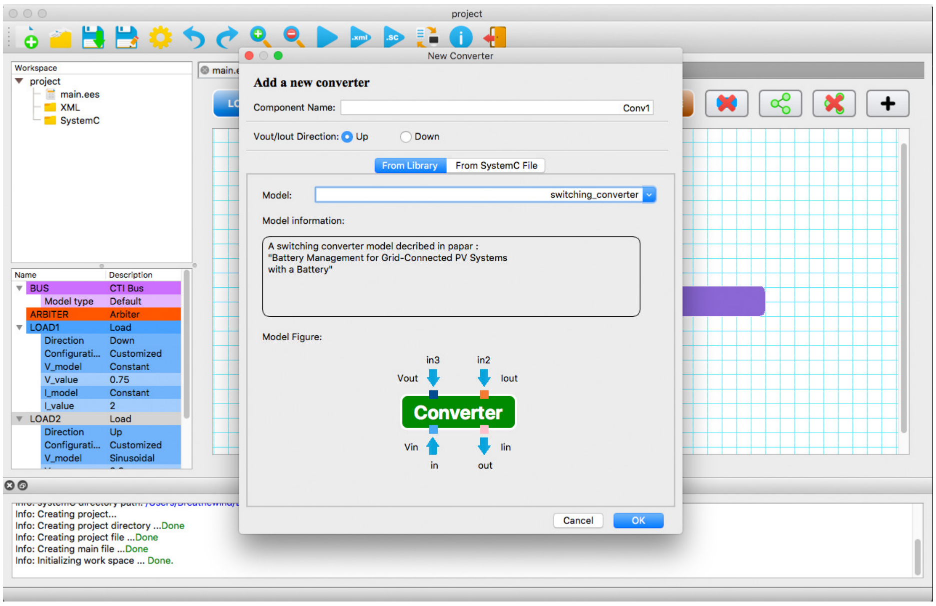


Figure 6.6 Case study 1: Creating converter components by switching converter model

After confirming there exists switching converter model in the model library, users are able to create converter components with switching converter model by selecting corresponding model name in the dialog for creating new converter component. There are four converters in this case 1. All of them are implemented by the same switching converter model. To avoid reiteration, here, only the creating process of the first converter, *i.e.* the converter with name "Conv1", is demonstrated. After configuring all the required fields as shown in Figure 6.6, click "OK" button. A converter with target model will be created in design space. The other converters are able to be created following the same process.

By the same process as creating converters by the model existed in library, the battery and PV panel model can be imported into model library. The corresponding ESD and power source component are created as shown in Figure 6.7 and Figure 6.8.

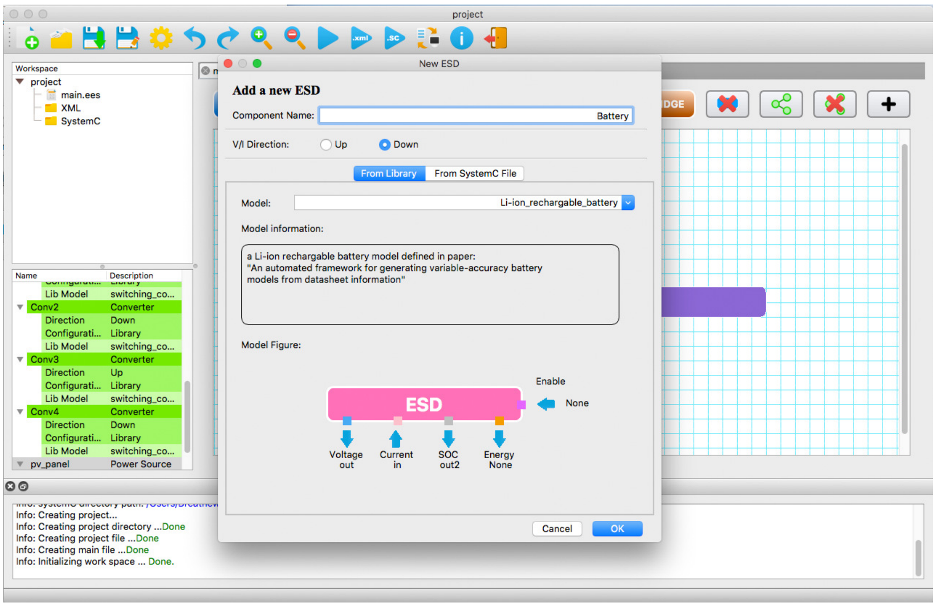


Figure 6.7 Case study 1: Creating ESD component by battery model

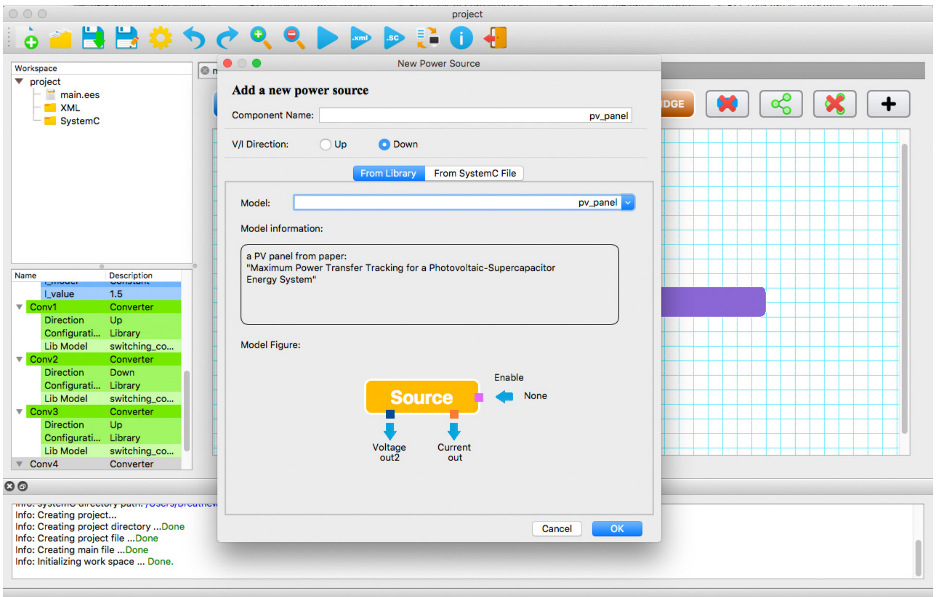


Figure 6.8 Case study 1: Creating power source components by PV panel model

After all the steps above, a EES as shown in Figure 6.9 is accomplished.

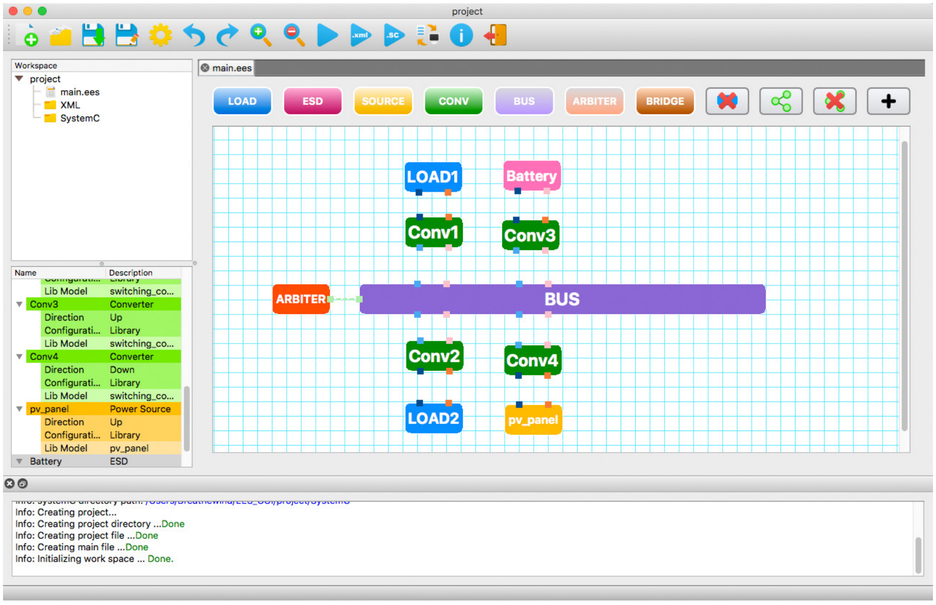


Figure 6.9 Case study 1: System with all components created

Create connections

Now a system design with all EES components required is accomplished. Then connections should be created between each two components such that topology information would be insert to current design.

The fast connection approach introduced in Section 4.4.2 to create connections between components would be adopted as shown in Figure 6.10. Finally a system which is ready for code generation is achieved as shown in Figure 6.11.

6.1.2 Code generation

Since we have already built the entire system for code generation, after clicking the code generation button, all the codes of current design will be generated automatically. In Figure 6.12, it shows the generated header files for each component.

6.1.3 Simulation results

In this section, the simulation result with 1 second time step will be demonstrated to prove the correctness of the code generated by IDE. Since the trace of LOAD1 is constant value with 2.75V and 2A, the total power consumed by LOAD1 and LOAD2 is mainly depended

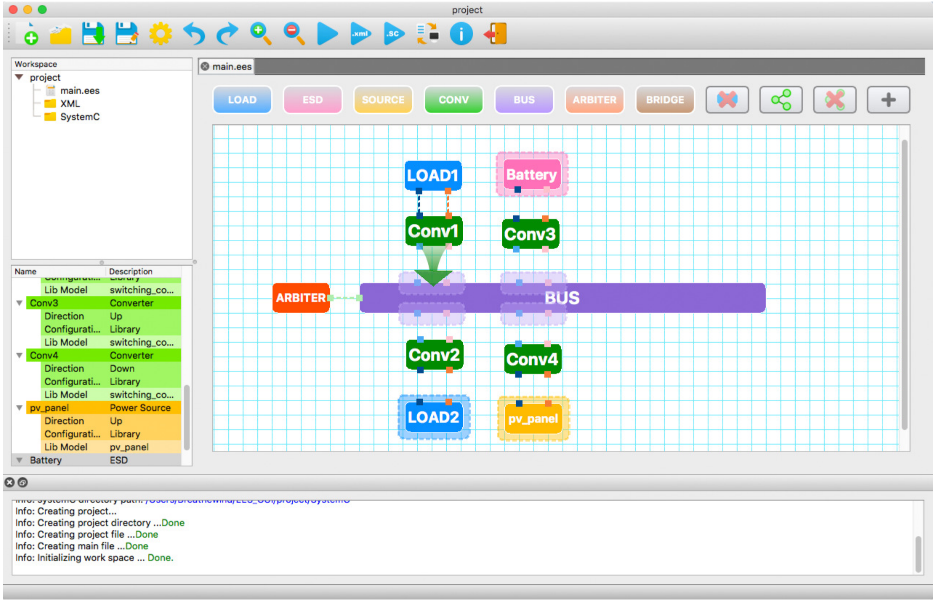


Figure 6.10 Case study 1: Creating connection between each two components

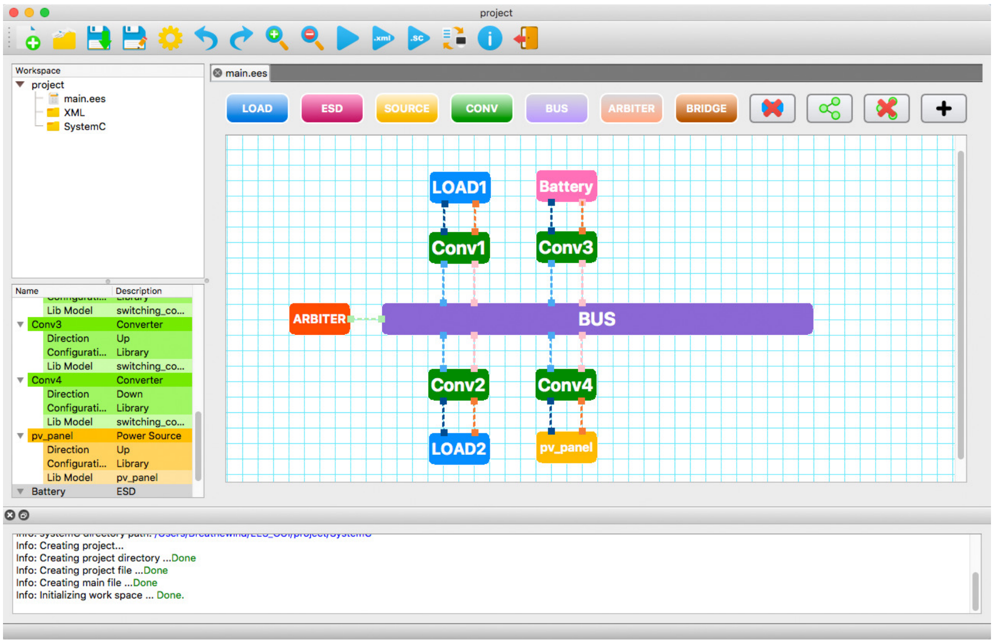


Figure 6.11 Case study 1: Final system

on the voltage value of LOAD2. In Figure 6.13a.1, the voltage trace of LOAD2 is displayed. It is the rising part of a sin wave. that means the total power consumed by LOAD1 and LOAD2 is increasing. In Figure 6.13a.2, it shows power trace of loads and power source. The dotted line demonstrates power consumed by loads and the solid line demonstrates power produced by power source. The top figure in Figure 6.13b shows the battery SOC. The figure

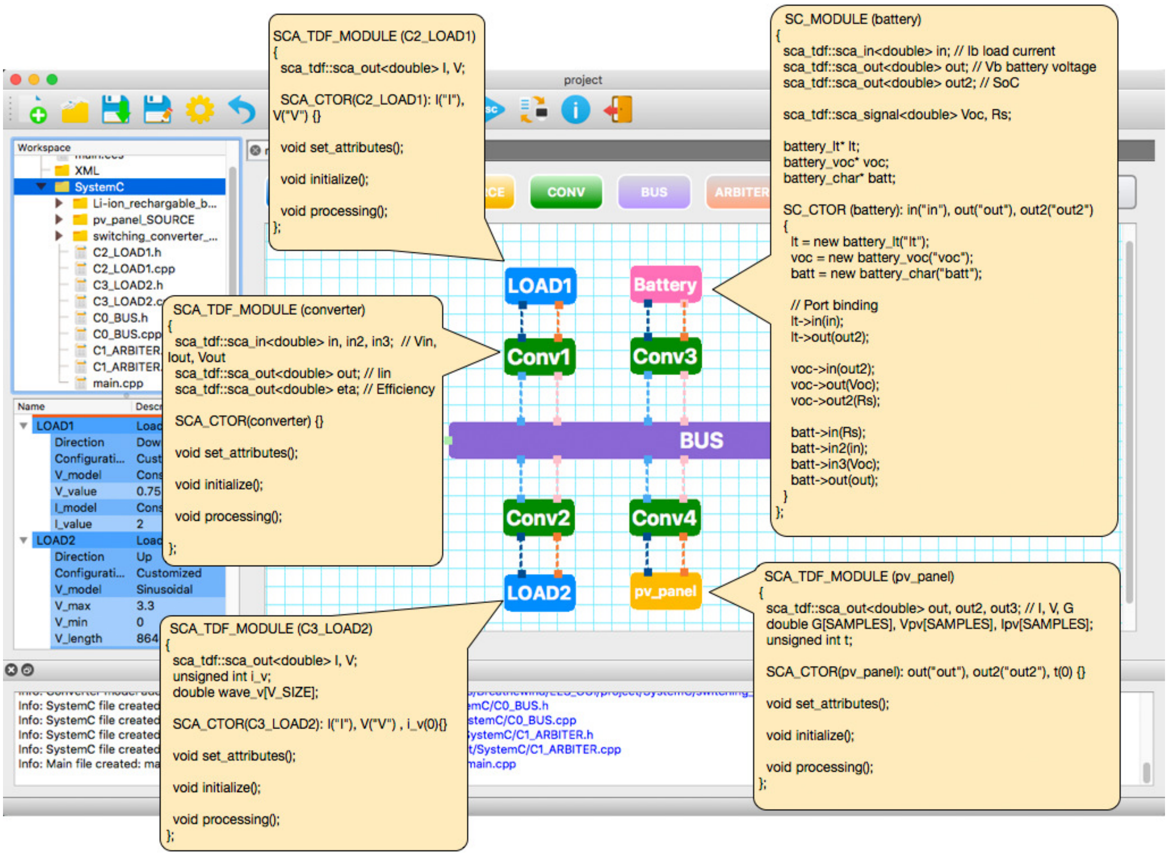
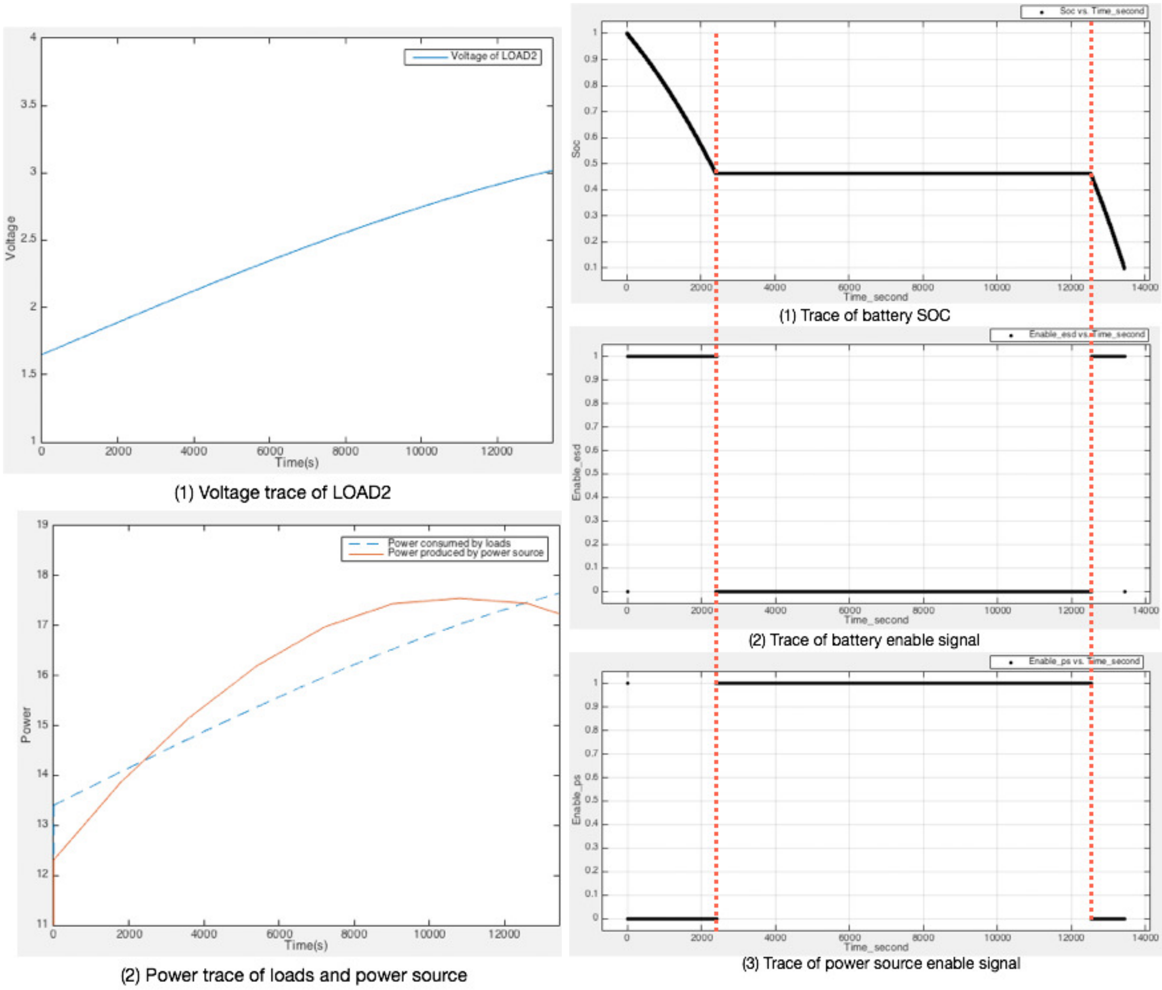


Figure 6.12 Case study 1: Generated code

in the middle of Figure 6.13b shows the trace of battery enable signal. In the bottom figure of Figure 6.13b, the trace of power source enable signal is demonstrated.

We can see, in initial part, the power consumed by loads is higher than the power produced by power source in Figure 6.13a.2. Depending on the policy, during this time, the loads are supplied by the battery. It corresponds to the part on the right hand of the first dot line in Figure 6.13b which battery SOC decreases, the battery is enabled and the power source is not working. After the point around 2500s, the power produced by power source is higher than the power consumed by loads which means power source is able to produce enough power to supply the loads. As shown in the area between two dot lines in Figure 6.13b, the power source is enabled and the battery stops working. Since the battery is not working during this time period, the battery SOC stops decreasing also. In the area after around 12500s in Figure 6.13a, the power consumed by loads is higher than the power produced by power source again. Loads stop being supplied by power source. The battery works and battery SOC decreases again. Finally, in the point around 13700s, the battery SOC reaches 10%. Then the system stops working.

All the data captured by simulation is consistent with the policy described before. That means the codes generated by IDE are correct.



(a) Case study 1: Data trace

(b) Case study 1: Trace of battery SOC and enable signals

Figure 6.13 Case study 1: Simulation result of case study 1

6.2 Case study 2

The second example is a system which consists with two loads, a supercapacitor, a battery and four converters. They are organized as shown in Figure 6.14. In details, the design is composed of:

- two load devices, *i.e.*: a STM8L processor and a Wifi Transceiver, for sending and receiving data to other network nodes every 10s [7, 17];

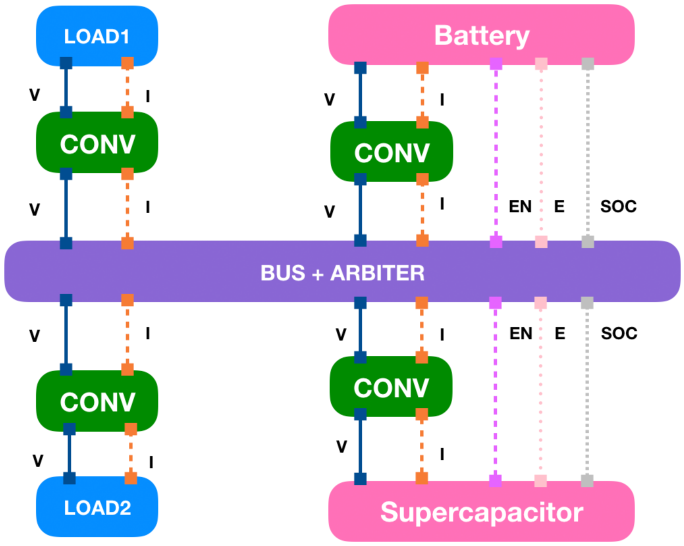


Figure 6.14 Design of case study 2

- a solid state lithium thin film battery by ST, with nominal capacity of 700 μAh and nominal voltage 3.9 V [1];
- a NessCap supercapacitor with 10 F capacitance [12];
- a DC-DC converter modeled as in [13], connected to the thin film battery, where conversion efficiency is function of input voltage, output voltage and current;
- three Texas Instruments TPS63060 DC-DC converters [18] whose conversion efficiency is function of input and output voltage, connected to the supercapacitor and the loads;
- a CTI bus modeled as an ideal current conductor from the energy providers (*i.e.*, battery and supercapacitor) to the load devices, with a constant reference voltage of 3.0V. The CTI arbiter is augmented with a "charge allocation" policy, that uses the supercapacitor for high load demand intervals and the battery for low load demand intervals.

This example highlights that Peukert's model may be adopted as battery model in an initial design phase, when fewer details about the battery characteristics may be available (e.g., it does not require to model the battery internal resistance). The model library of my program allows to easily replace the Peukert's model with a more accurate one at later phases of the design process, to get a more precise estimation of system behavior as shown in Figure 6.15.

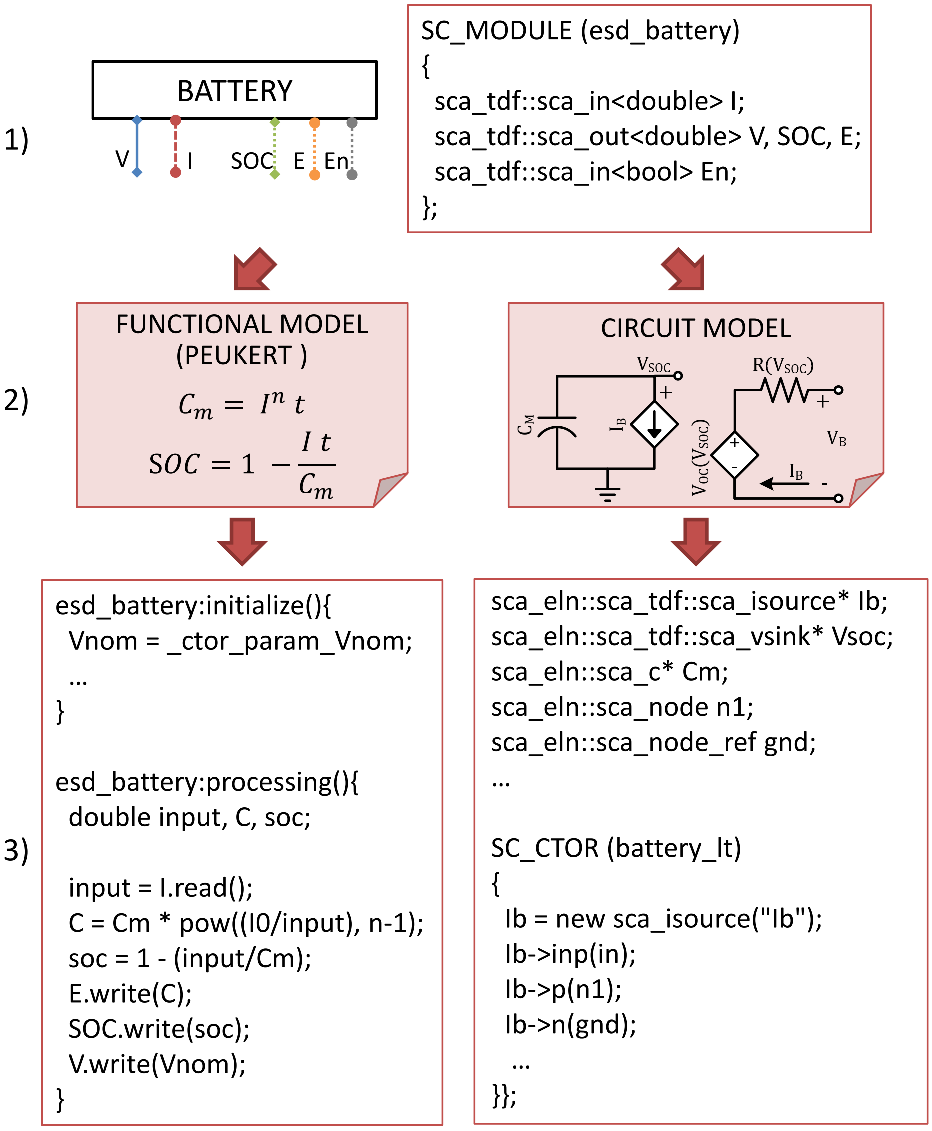


Figure 6.15 Application of the methodology to a battery. Interface is as defined for ESDs in Figure 3.7. Implementation adopts either Puekert's law (left-hand side) or a circuital model (right-hand side).

6.2.1 Implementation

As GUI implementation section of Example1, users need to create a new project, create components, create connections and generate codes step by step. To avoid useless reiteration, all the models needed in this case have already been imported to model library before creating the project.

Create a new project

A new project named "Example2" need to be created as shown in Figure 6.16.

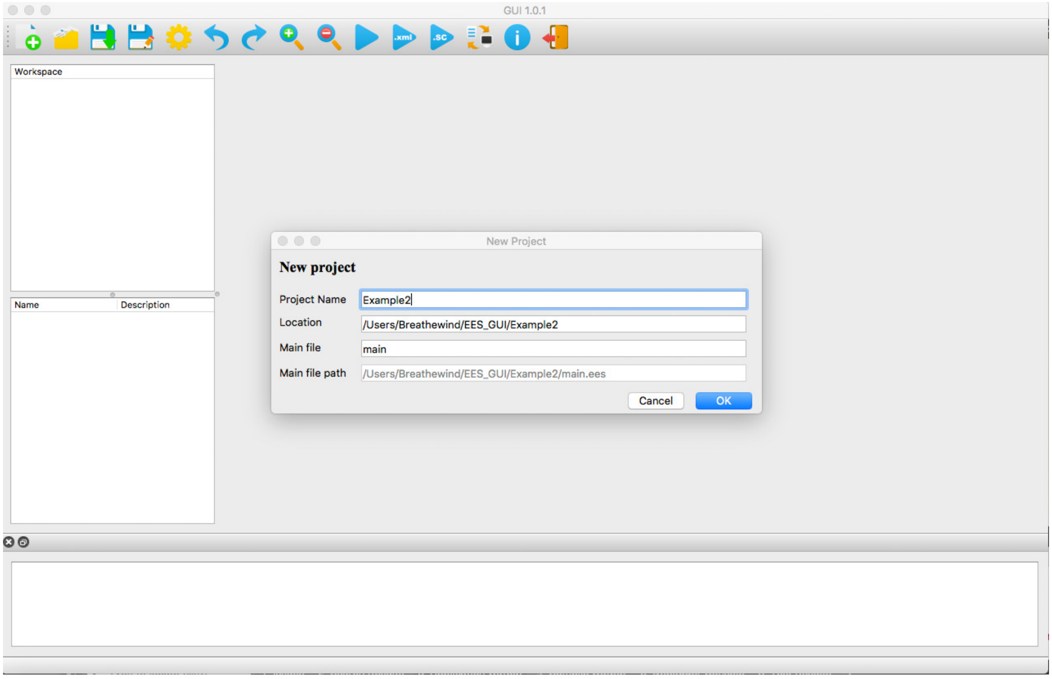


Figure 6.16 Case study 2: Creating new project

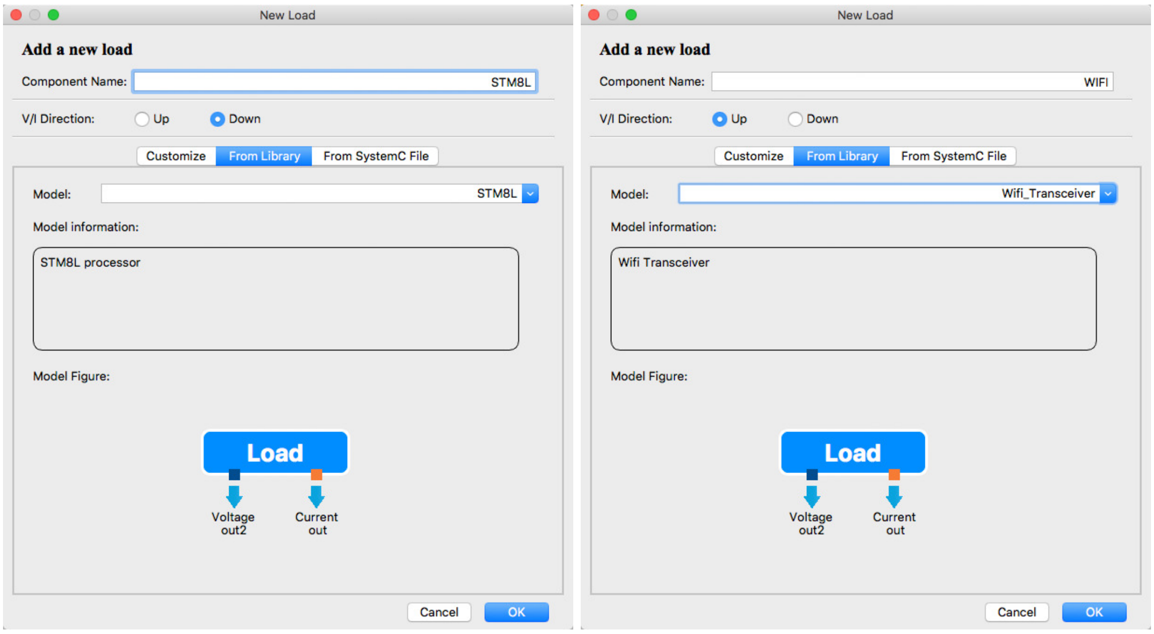
Create components

Next step is creating components of user customized design. There are two load components in this case. One is created as a STM8L processor as shown in Figure 6.17a. Another is created as a WIFI transceiver as shown in Figure 6.17b.

There are also two ESD components in case study 2. One is a NessCap supercapacitor as shown in Figure 6.18.

Another is a solid state lithium thin film battery. It will be firstly built by Peukert model as shown in Figure 6.19a. The code of that model would be generated and simulated. Then the model will be changed to circuit model as shown in Figure 6.19b. Here we can see, by my IDE, users are able to switch the model of EES component easily. It would be a great help during EES design.

For converter components in this case, the converter connected to battery adopts different converter model comparing with converters connected to other functional components as shown in Figure 6.20b. Since the converters connected to loads and supercapacitor are using the same converter model, for simplicity, only the process to create converter for



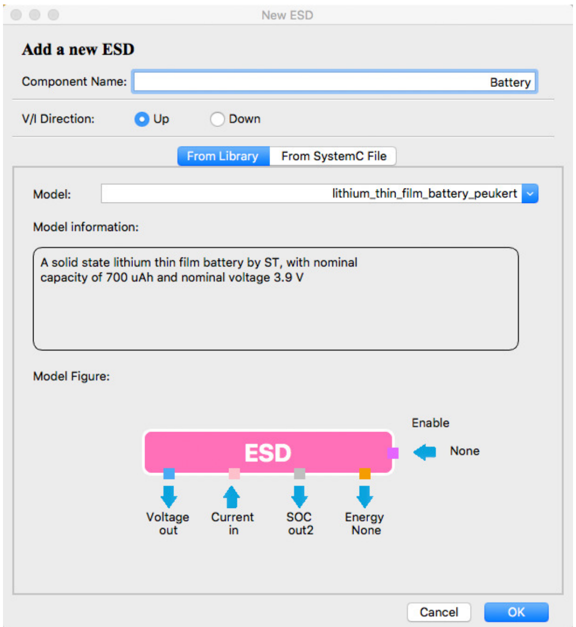
(a) Case study 2: Create a load component as a STM8L processor

(b) Case study 2: Create a load component as a WIFI transceiver

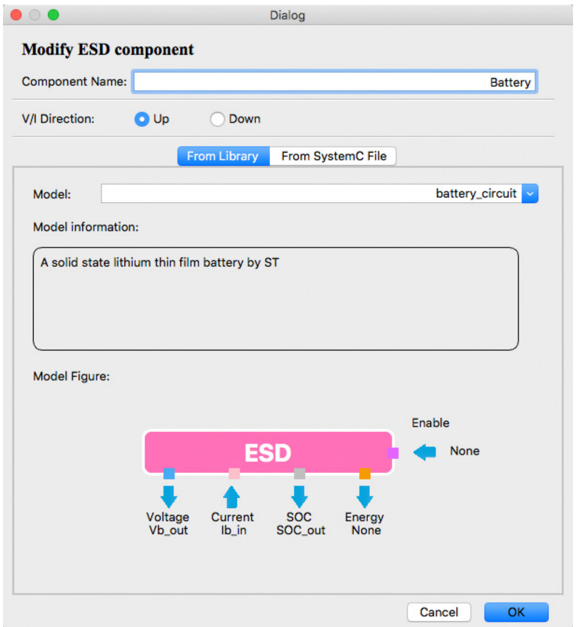
Figure 6.17 Case study 2: Create load components

Figure 6.18 Case study 2: Create an ESD component as a supercapacitor

load component STM8L processor is demonstrated as shown in Figure 6.20a. Then all the components are already inserted to IDE as shown in Figure 6.21.

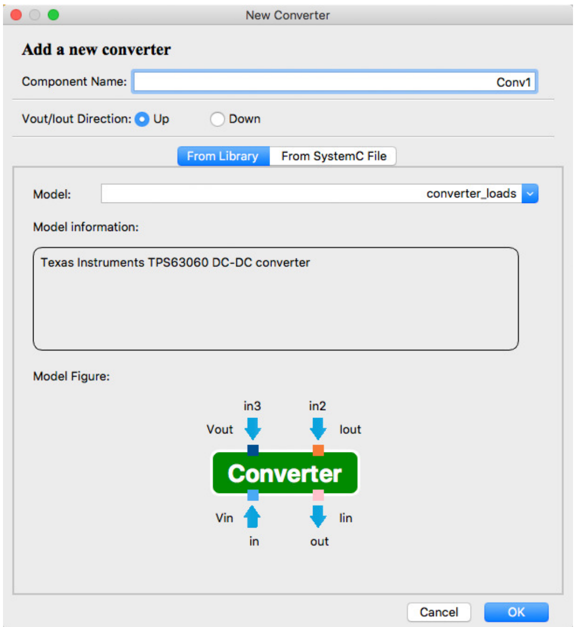


(a) Case study 2: Create a battery by Peukert model.

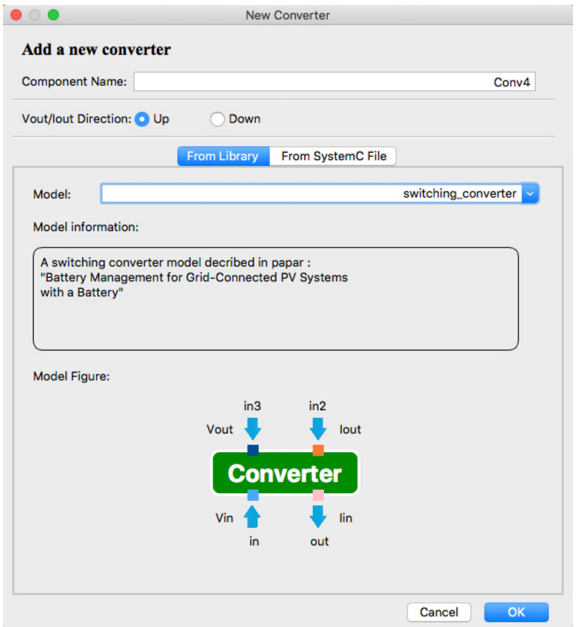


(b) Case study 2: Create a battery by circuit model.

Figure 6.19 Case study 2: Create an ESD component as a battery



(a) Case study 2: Create an converter component as a TPS63060 DC-DC converter



(b) Case study 2: Create an converter component as a switching converter

Figure 6.20 Case study 2: Create converter components

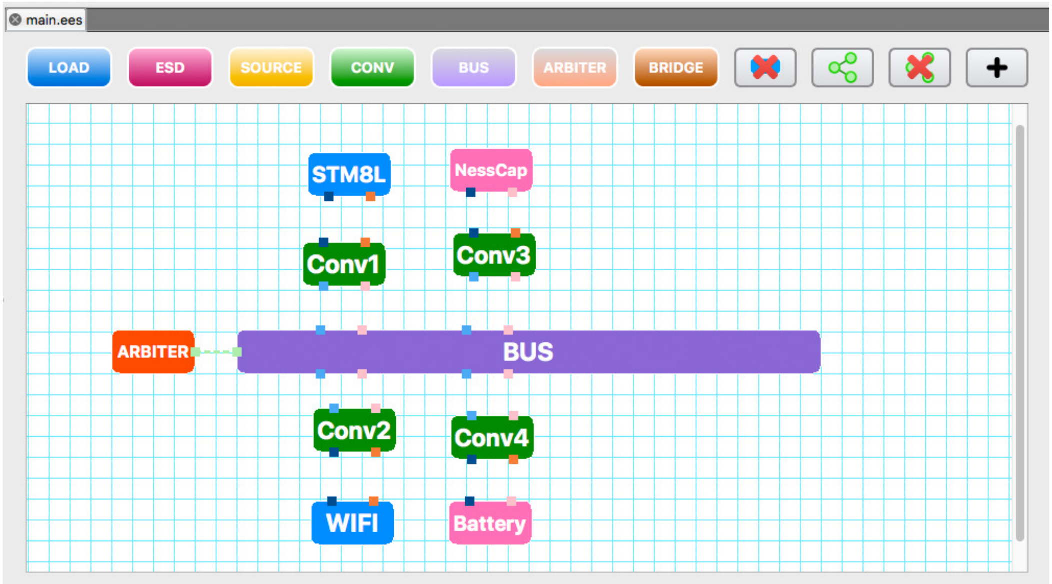


Figure 6.21 Case study 2: System with all components created

Create connections

Then connections need to be created by either one of the approaches introduced before. The final system is shown in Figure 6.22.

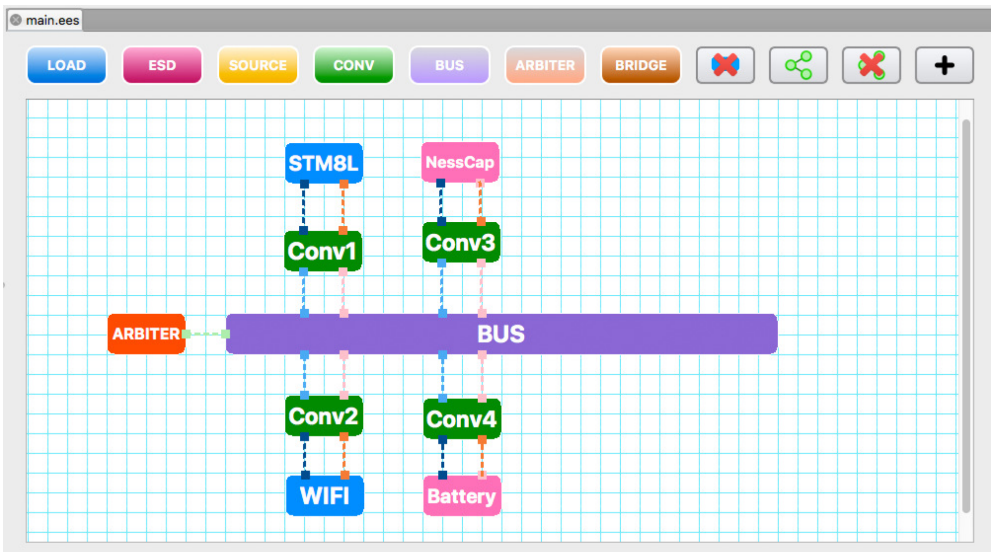


Figure 6.22 Case study 2: Final system

6.2.2 Code generation

We have already built the entire system for code generation by previous operations. After clicking the code generation button, all the codes of current design will be generated automatically. In Figure 6.23, it shows the generated header files for all the components except battery.

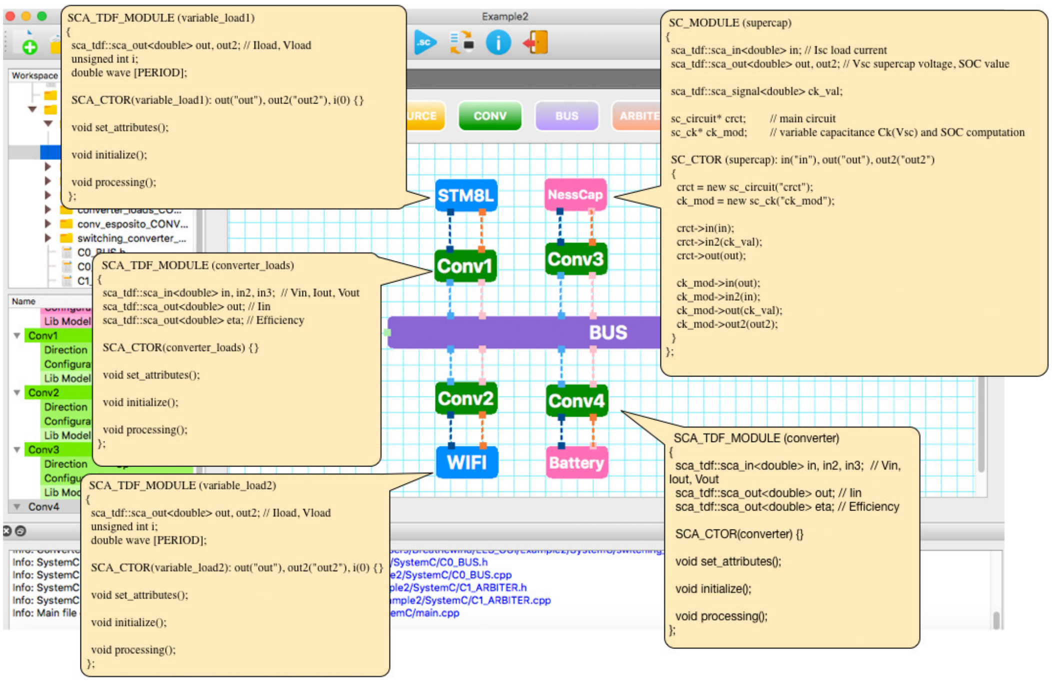


Figure 6.23 Case study 2: Generated code for all the components except battery.

We generated the code for each of the versions of case study two, i.e., once for each model adopted for the battery. This is done to compare the behavior of the system depending on the accuracy of the model for the battery, i.e., when adopting Peukert's model and a circuit model. Thus, Figure 6.24 displays the two generated header files of Peukert model and circuit model.

6.2.3 Simulation results

In this section, the simulation result with 0.1 second interval will be demonstrated to prove the codes generated by IDE could work. In this case, two battery models are adopted. In Figure 6.25, it shows the data trace of both battery models. The upper one is the trace of Peukert model, and the lower one is the trace of circuit model. Since the circuit model is more precise than the Peukert model, we can see there exists sightly difference between Figure 6.25.(1) and Figure 6.25.(2).

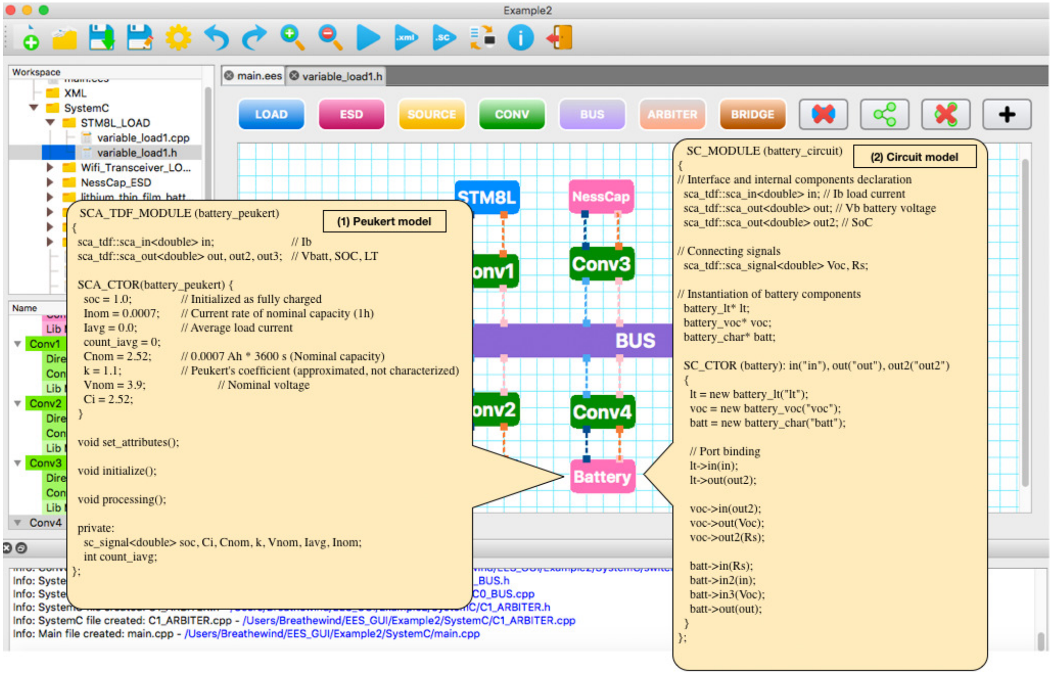
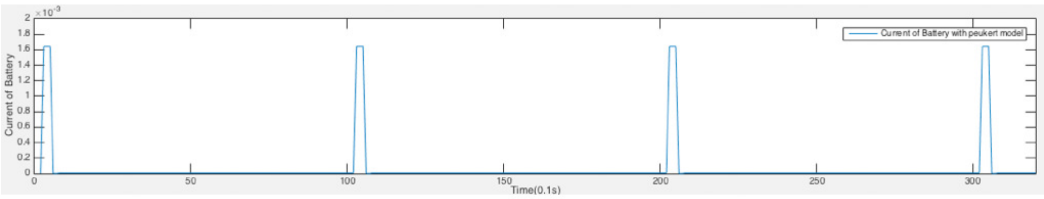
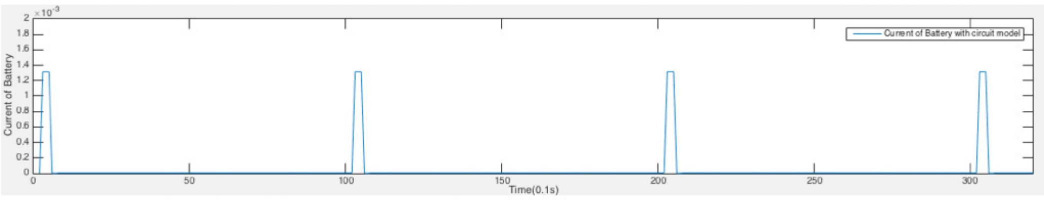


Figure 6.24 Case study 2: Generated code for battery.



(1) Current of Battery with peukert model



(2) Current of Battery with circuit model

Figure 6.25 Case study 2: The data trace of battery current

The trace of battery SOC in both models is shown in Figure 6.26. Since the current of Peukert model is larger than it of circuit model, the SOC of Peukert model decreases faster. It proves the generated codes work properly.

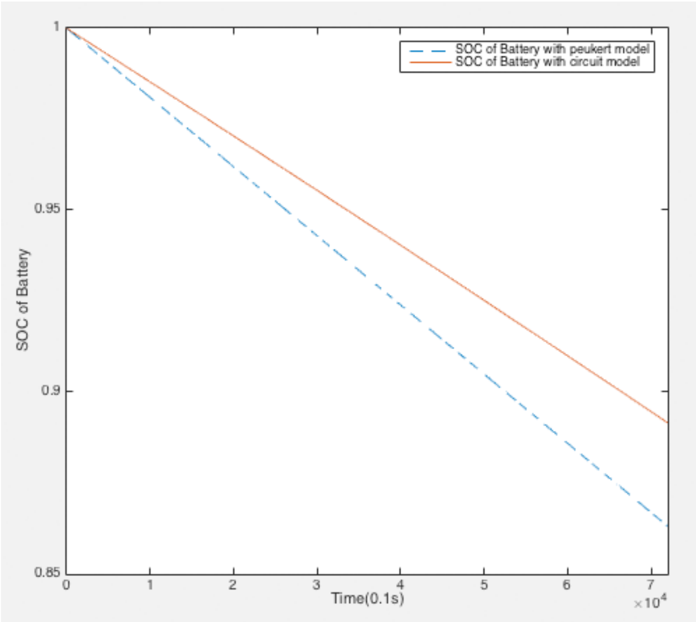


Figure 6.26 Case study 2: The trace of battery SOC

6.3 Case study 3

The last example is a more complicated system which consists with three loads, a supercapacitor, a battery, a PV cell and six converters. They are organized as shown in Figure 6.27. In details, the design is composed of:

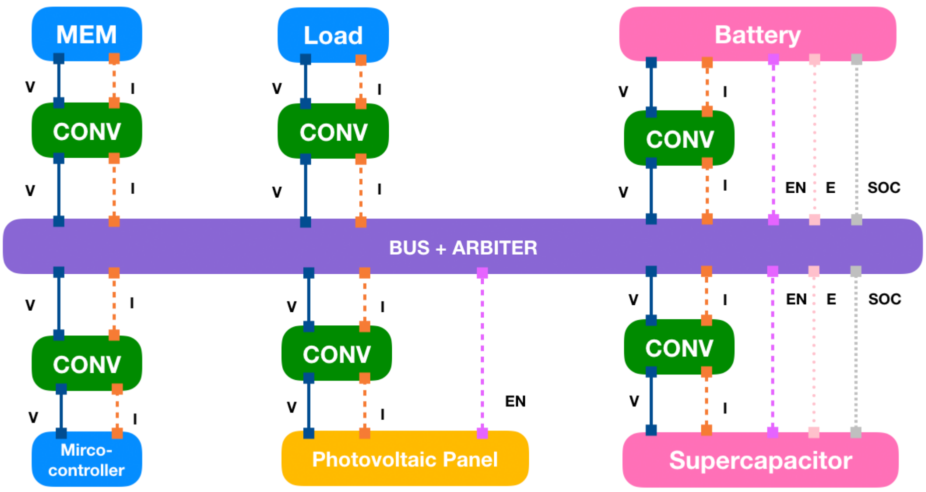


Figure 6.27 Design of case study 3

- Three load devices including: (1) a set of MEMS sensors such as a 3-axis accelerometer, a MEMS pressure sensor and temperature sensors; (2) a 32-bit ARM-based ST microcontroller used as computing unit to perform data acquisition and control over all the operation of the system; (3) an ST low-power wireless radio for performing wireless data transmission. All these devices are modeled with their execution traces for current and voltage obtained with experimental measurements, as shown in Figure 3.2;

- A solid state lithium thin film battery by ST, with nominal capacity of 700 μAh and nominal voltage 3.9 V [1];
- A Panasonic stacked coin type supercapacitor with 0.33F capacitance;
- A DC-DC converter modeled as in [13], connected to the thin film battery, where conversion efficiency is function of input voltage, output voltage and current;
- Five Texas Instruments TPS63060 DC-DC converters [18] whose conversion efficiency is function of input and output voltage, connected to the supercapacitor and the loads;
- The power bus modeled as an idea DC bus consisting of an idea wire and a voltage generator at the constant reference voltage of 3.0V. Moreover, the power bus is enhanced with an arbiter that abstracts the power behaviors of the overall system. As long as the power drawn from the PV panel satisfies the power demand of the loads, these are supplied by the power source. Otherwise, the loads are supplied by the supercapacitor during high load power demand intervals, while, during low load power demand intervals, the arbiter activates the thin file battery. When the power supplied by PV panel is larger than the power consumed by loads, the extra power will be used to recharge battery and supercapacitor.

6.3.1 Implementation

Case 3 is a more complicated case comparing with previous two cases. I will show how my program works when dealing with a relative large system.

Create a new project

Same as previous cases, an new project named "Example3" need to be created as shown in Figure 6.28.

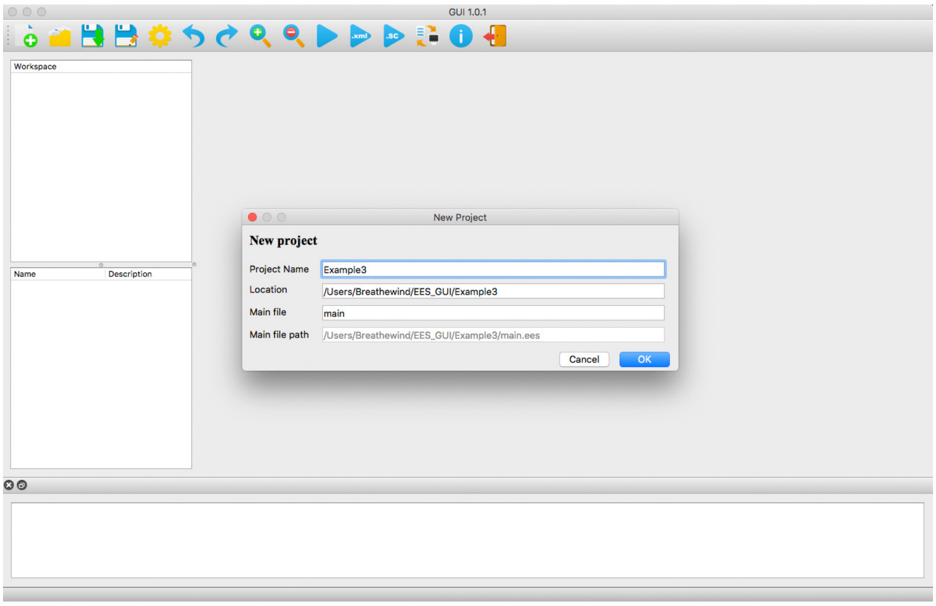


Figure 6.28 Case study 3: Creating new project

Create components

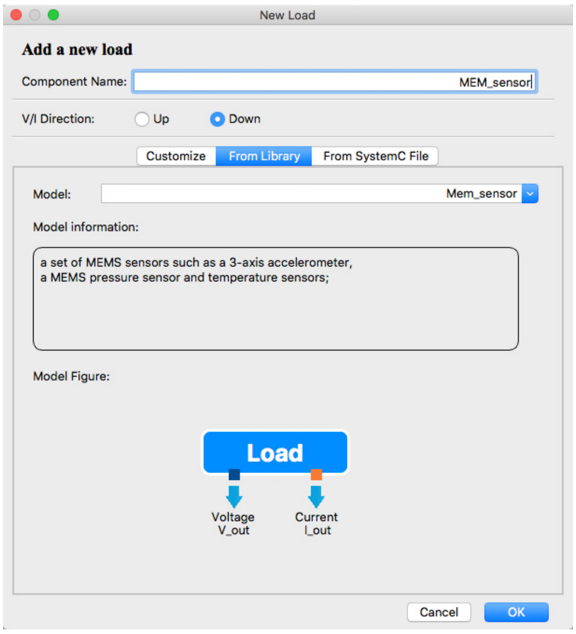
There are three load components in this case. To speed up the building process, their models are already imported to model library. Here, I just need to build components with corresponding model as shown in Figure 6.29.

There are two ESD components in this case. One is a supercapacitor as shown in Figure 6.30a. Another is a solid state lithium thin film battery as shown in Figure 6.30b.

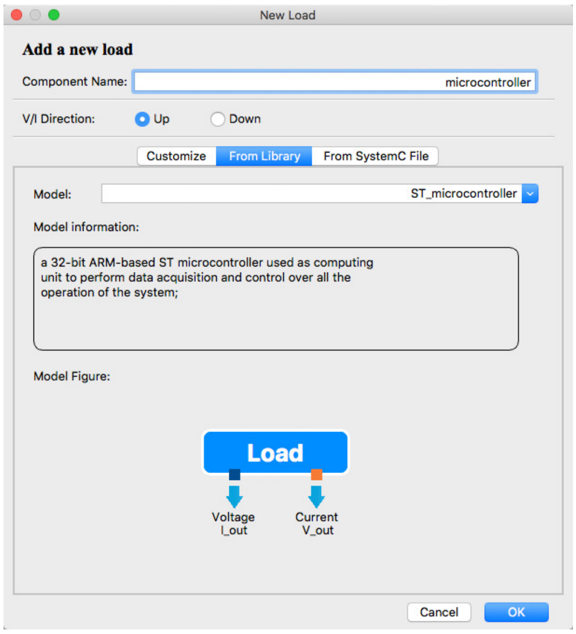
Among all converter components in this case. the converter connected to battery adopts different converter model comparing with other converters. The dialog for creating that converter is displayed in Figure 6.31b. Other components are connected to the converters with small converter model. For simplicity, only the process to create the first converter is demonstrated as shown in Figure6.31a. Then all the components are already inserted to IDE as shown in Figure 6.32.

Create connections

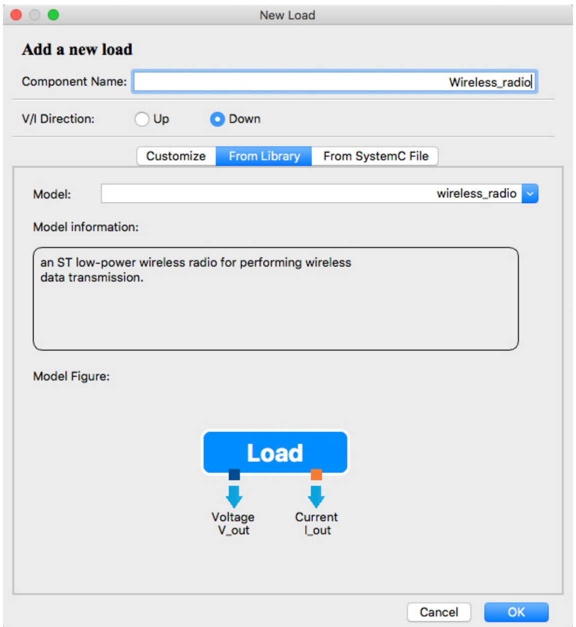
Then connections need to be created with preferred approach. The final system is shown in Figure 6.33.



(a) Case study 3: Create a load component as a set of MEMS sensors

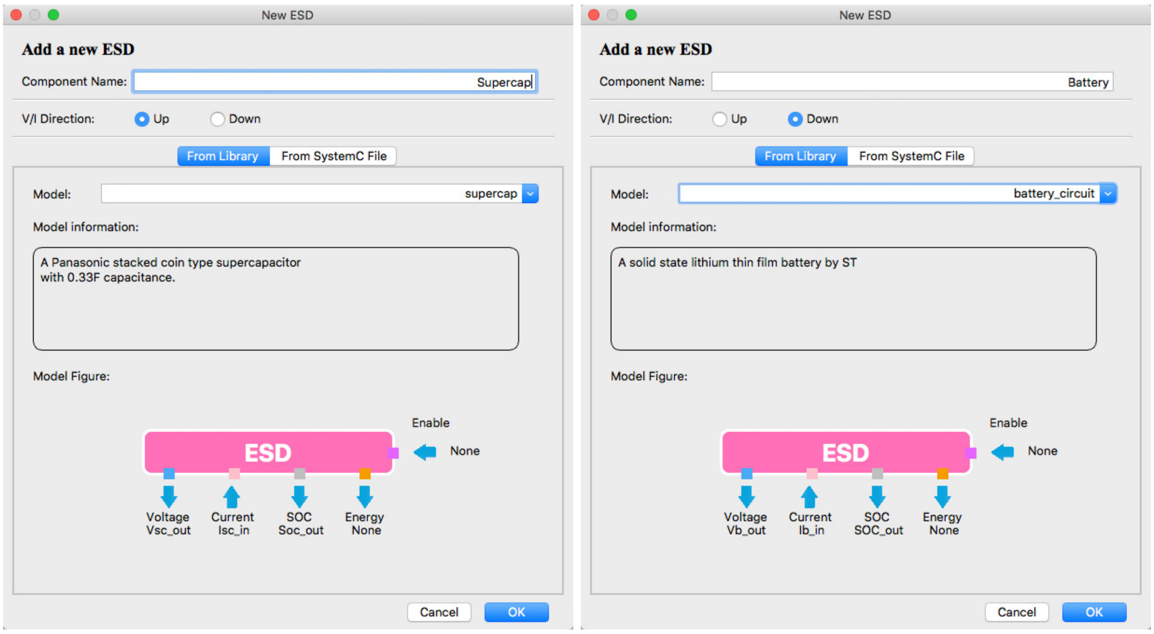


(b) Case study 3: Create a load component as a 32-bit ARM-based ST microcontroller



(c) Case study 3: Create a load component as a wireless radio

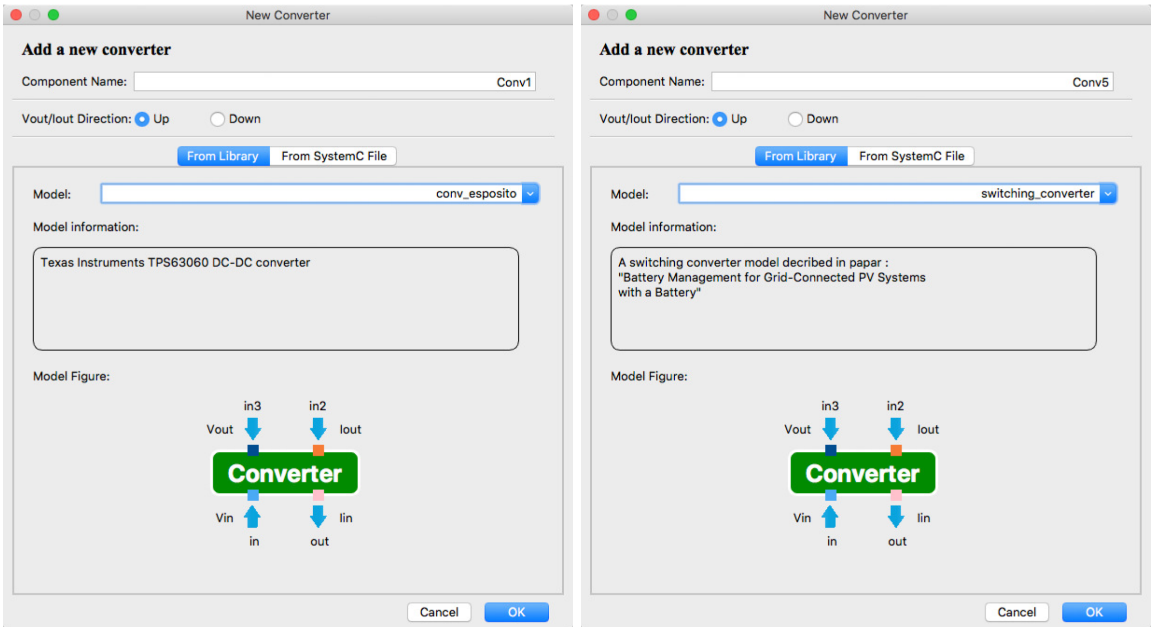
Figure 6.29 Case study 3: Create load components



(a) Case study 3: Create an ESD component as a supercapacitor

(b) Case study 3: Create an ESD component as a battery

Figure 6.30 Case study 3: Create ESD components



(a) Case study 3: Create an converter component as a TPS63060 DC-DC converter

(b) Case study 3: Create an converter component as a switching converter

Figure 6.31 Case study 3: Create converter components

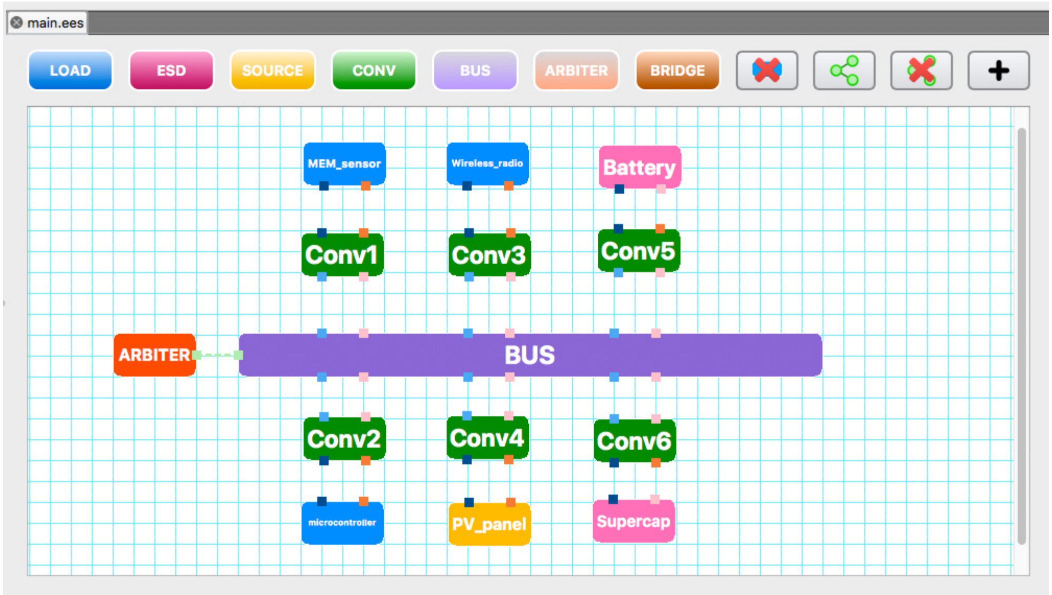


Figure 6.32 Case study 3: System with all components created

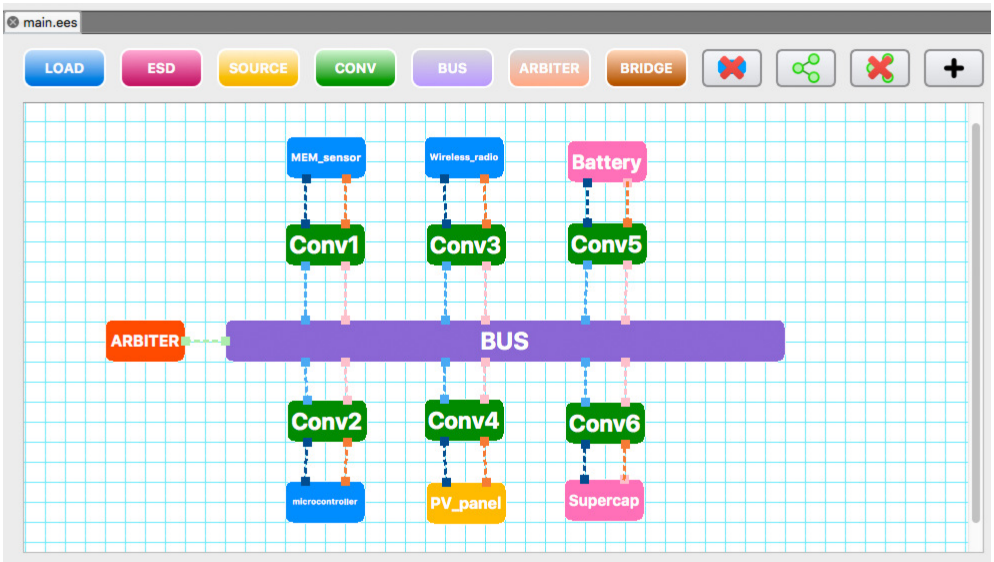


Figure 6.33 Case study 3: Final system

6.3.2 Code generation

As previous cases, the entire system is accomplished, by clicking the code generation button, IDE will generate all the codes of current design automatically. In Figure 6.34, it shows the generated header files for each component.

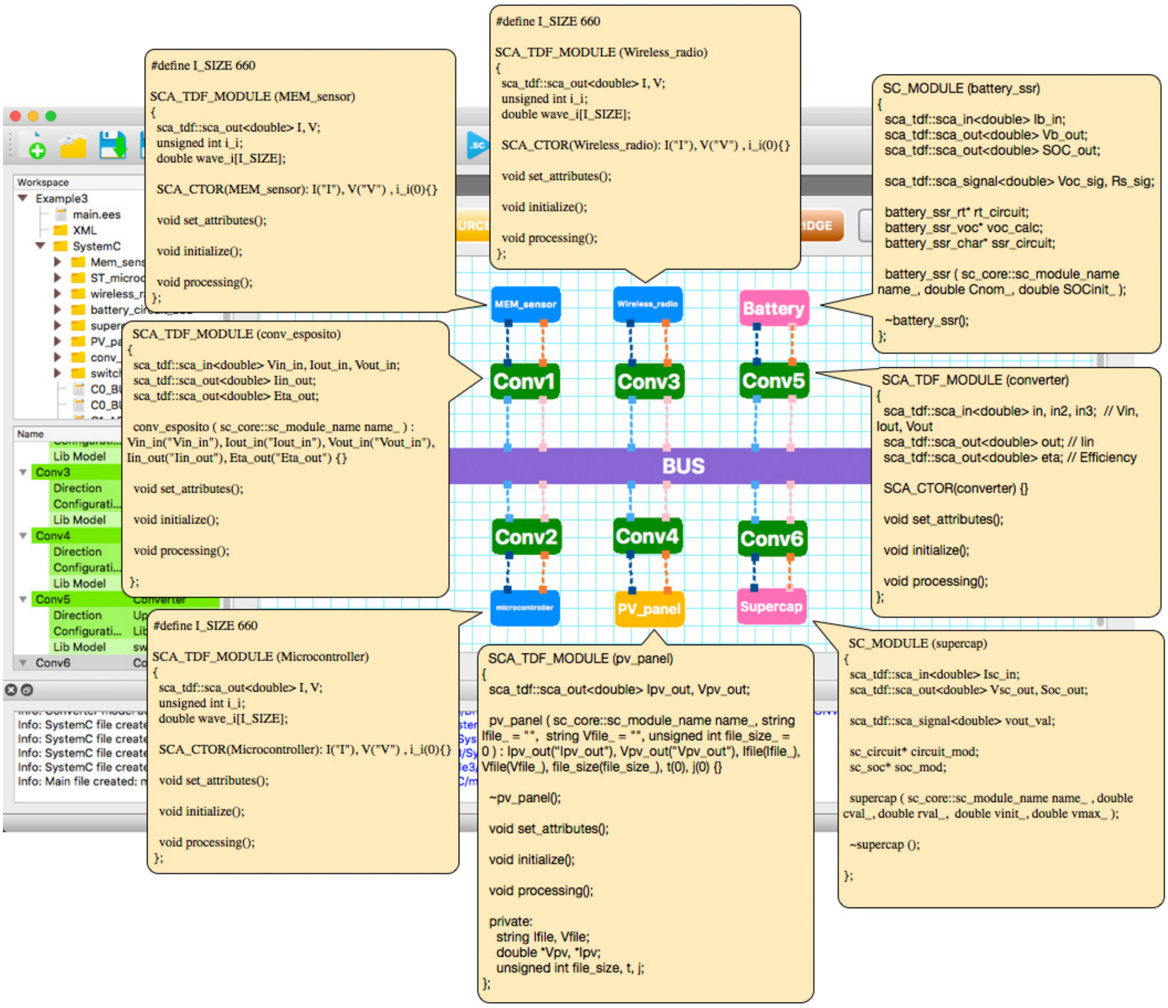
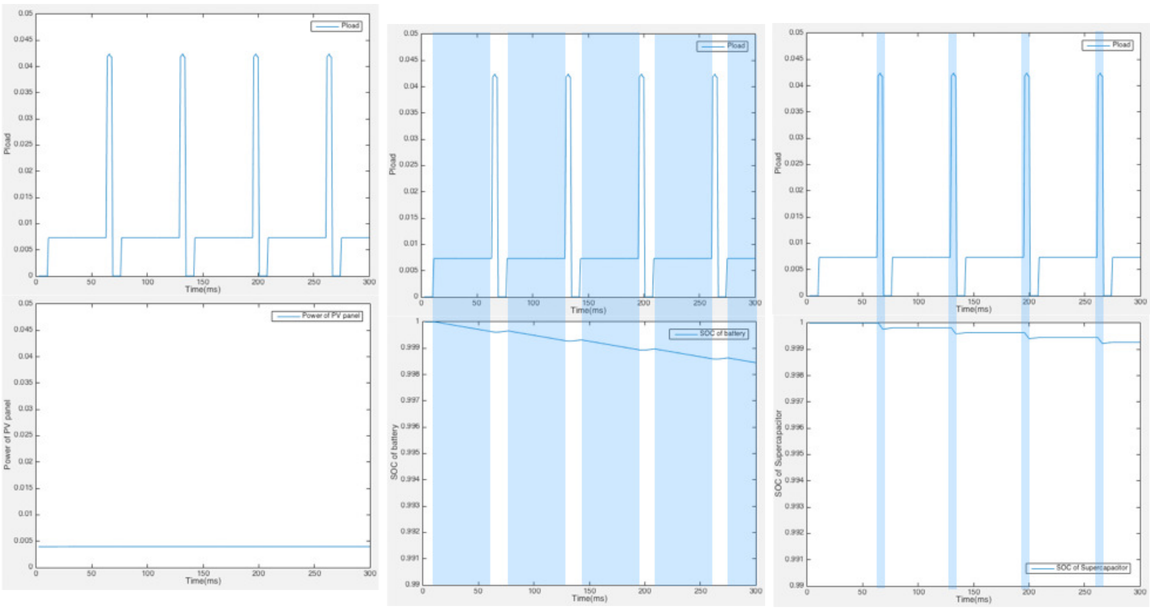


Figure 6.34 Case study 3: Generated code

6.3.3 Simulation results

In this section, the simulation result with 1 millisecond time step will be demonstrated to prove the codes generated by IDE could work. In Figure 6.35, there are data trace of Case 3 from 0ms to 300ms. In Figure 6.35a, the upper figure shows that the total power consumed by loads and the lower figure shows that the power provided by PV panel. We can see that, during the time period between 0ms and 300ms, the power trace of loads are regular waveforms which is firstly close to 0, *i.e.* smaller than the power provided by PV panel. Then it switches to low power demand which is higher than the power provided by PV panel. Finally, after it switches to high power demand for a short time, it goes back to the value close to 0.

In the shadowed part of Figure 6.35b, there are the low power demand intervals. We can see that in the lower figure, the SOC of battery decreases. It is consistent with the policy which the arbiter activates the thin file battery during low power demand intervals. And in the shadowed region in Figure 6.35c, the SOC of supercapacitor decreases when the loads are in high power demand intervals. Therefore, it is also consistent with the policy which loads are supplied by the supercapacitor during high power demand intervals. According to the policy, the PV panel supplies the loads and recharges battery and supercapacitor when the power provided by the PV panel is larger than the power consumed by loads. During the intervals when load workload is close to 0, we can see both the SOC of battery and supercapacitor are slightly increased, since at these time period, the power provided by the PV panel is large enough to supply loads.



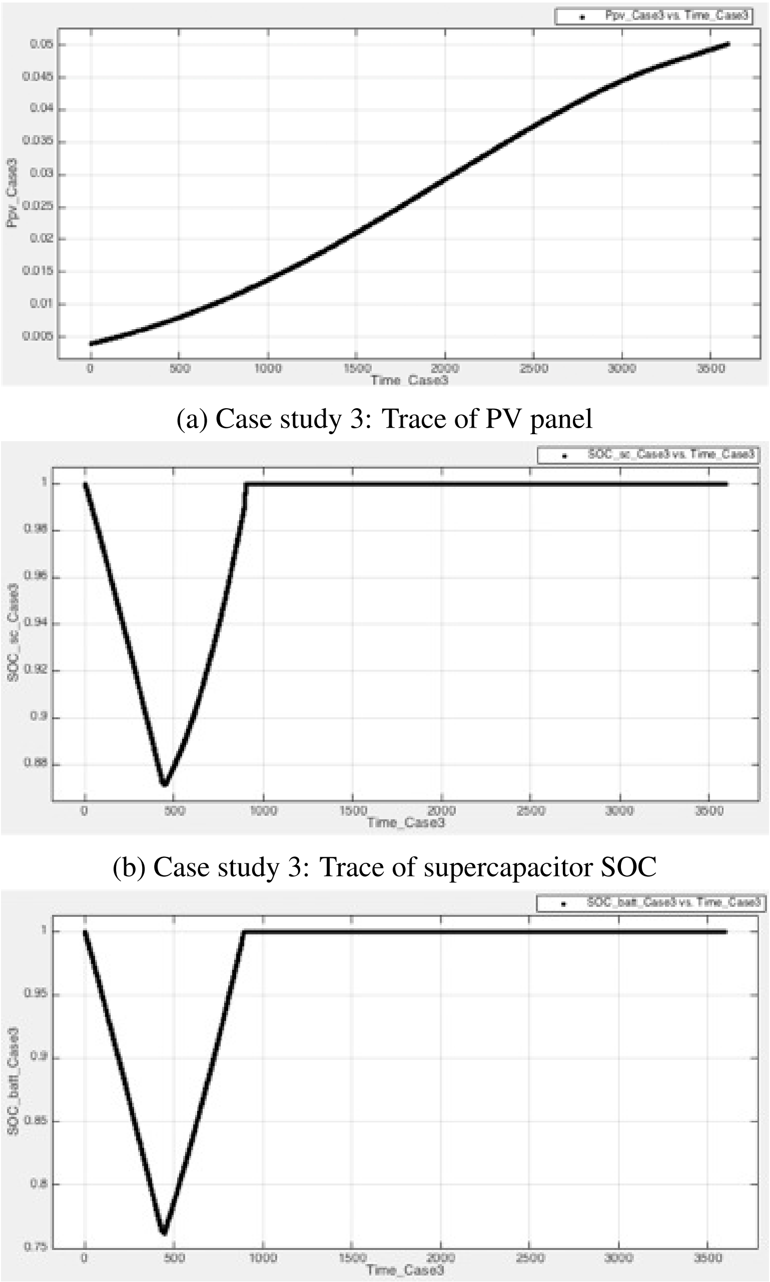
(a) Case study 3: Power consumed vs power provided by PV panel

(b) Case study 3: Power consumed vs SOC of battery

(c) Case study 3: Power consumed vs SOC of supercapacitor

Figure 6.35 Case study 3: Simulation result of Case 1 from 0ms to 300ms

From the simulation results between 0s to 3200s as shown in Figure 6.36, we can see in the beginning both the SOC of battery and supercapacitor are decreasing until the point around 490s which the power provided by PV panel is larger than the power consumed by loads during low power demands intervals. After that time the battery is slowly recharged until full-recharged. The supercapacior is also slowly recharged, but before the power provided by PV panel is larger than the power consumed by loads during high power demands intervals, the supercapacitor still need to supply loads during high power demands intervals.



(a) Case study 3: Trace of PV panel

(b) Case study 3: Trace of supercapacitor SOC

(c) Case study 3: Trace of battery SOC

Figure 6.36 Case study 3: Simulation result of Case 3

After all the content above described, we can see the simulation result of the codes generated for Case 3 are correct. That means the IDE works well during building the design of case study 3.

Chapter 7

Conclusions

My thesis proposed a possible solution to speed-up EES design and generate SystemC code for target system according to user design. By this approach, designers are able to build their EES design in a high level without a good knowledge in programming and have the flexibility to switch different models during system implementation. It makes EES design architectural. When a certain EES component design is accomplished and stable enough, if necessary, that design could be imported to model library of my program as a model. When that design is needed in the future, designers could directly initiate an EES component by that model from library and include it to their system. The progress could be very convenient and fast. It will significantly speed-up the process of EES design. Another contribution of my work is that it does not only provide a possible approach for users to build their system but also has the ability to convert logical design to SystemC code which could be used in future simulation. The IDE I provided in this thesis automates the EES design flow. And the IDE is extendable, there maybe some more works could be done in the future. For instance, I leave the possibility to add functions to simulate the generated codes inside the IDE by invoking SystemC-AMS simulator. It would be a useful tool for the ones who work in EES design domain.

Bibliography

[1] (2013). ST EFL700A39 EnFilm rechargeable solid state lithium thin film battery datasheet.

[2] (2018). Qt description. https://www.qt.io/.

[3] (2018). Systemc ams extensions. http://accellera.org/community/systemc/about-systemc-ams.

[4] Al Faruque, M. A. and Ahourai, F. (2014). A model-based design of cyber-physical energy systems. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 97–104. IEEE.

[5] Alnejaili, T., Mehdi, D., Drid, S., and Chrifi-Alaoui, L. (2015). Advanced supervisor control for a stand-alone photovoltaic super capacitor battery hybrid energy system for remote building. In *2015 4th International Conference on Systems and Control (ICSC)*, pages 278–283.

[6] Benini, L., Hodgson, R., and Siegel, P. (1998). System-level power estimation and optimization. In *ACM International Symposium on Low Power Electronics and Design (ISLPED)*, ISLPED '98, pages 173–178.

[7] Dementyev, A., Hodges, S., Taylor, S., and Smith, J. (2013). Power Consumption Analysis of Bluetooth Low Energy, ZigBee, and ANT Sensor Nodes in a Cyclic Sleep Scenario. In *IEEE International wireless symposium (IWS)*, pages 1–4.

[8] Kim, Y., Chang, N., et al. (2010). Maximum power transfer tracking for a photovoltaic-supercapacitor energy system. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 307–312. ACM.

[9] Kim, Y., Shin, D., Petricca, M., Park, S., Poncino, M., and Chang, N. (2013). Computer-aided design of electrical energy systems. In *International Conference on Computer-Aided Design*, pages 194–201. IEEE Press.

[10] Molina, J. M., Pan, X., Grimm, C., and Damm, M. (2013). A framework for model-based design of embedded systems for energy management. In *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2013 Workshop on*, pages 1–6. IEEE.

[11] Nassif, S., Nam, G.-J., Hayes, J., and Fakhouri, S. (2014). Applying vlsi eda to energy distribution system design. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 91–96. IEEE.

[12] NessCap (2014). NessCap ESHSR-0010C0-002R7 ultracapacitor datasheet.

[13] Park, S., Wang, Y., et al. (2012). Battery Management for Grid-connected PV Systems with a Battery. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 115–120.

[14] Petricca, M., Shin, D., Bocca, A., Macii, A., Macii, E., and Poncino, M. (2013). An automated framework for generating variable-accuracy battery models from datasheet information. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 365–370.

[15] Peukert, W. (1897). Über die Abhängigkeit der Kapazität von der Entladestromstärke bei Bleiakkumulatoren. In *Elektrotechnische Zeitschrift*, page 20.

[16] Shrivastava, A. and Calhoun, B. H. (2012). Modeling DC-DC converter efficiency and power management in ultra low power systems. In *2012 IEEE Subthreshold Microelectronics Conference (SubVT)*, pages 1–3.

[17] STMicroelectronics (2014). STM8L ultra low power series. www.st.com/web/en/catalog/mmc/FM141/SC1244/SS1336.

[18] Texas Instruments (2012). Texas Instruments TPS63060 DC-DC converter. www.ti.com/product/tps63060.

[19] Valenciaga, F. and Puleston, P. F. (2005). Supervisor control for a stand-alone hybrid generation system using wind and photovoltaic energy. *IEEE Transactions on Energy Conversion*, 20(2):398–405.

[20] Vinco, S., Chen, Y., Macii, E., and Poncino, M. (2016). A unified model of power sources for the simulation of electrical energy systems. In *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, pages 281–286.

[21] Yue, S., Zhu, D., Wang, Y., Pedram, M., Kim, Y., and Chang, N. (2013). Simes: A simulator for hybrid electrical energy storage systems. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 33–38. IEEE Press.