MULTI-DOMAIN MODELING THROUGH SPECIFICATION OF A DOMAIN SPECIFIC MODELING LANGUAGE FOR CYBER-PHYSICAL SYSTEMS DEVELOPMENT

By

Alexander L. Mendes

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

August, 2013

Nashville, Tennessee

DEDICATION

To my loving parents, Mary Ellen and Lloyd, for their unfaltering encouragement and all of their imparted wisdom. I would never have been able to achieve all that I have without their continued love and support.

To my beloved Avery, for her years of steadfast patience and comforting support. Your belief in me kept me on my feet even through the toughest times, and saw me through to the finish.

ACKNOWLEDGMENTS

I am truly indebted and grateful to my advisor Dr. Theodore Bapty, who gave me the instruction and support I required throughout my thesis writing. It would have been next to impossible to write this thesis without the help and guidance of Dr. Joseph Porter, who helped me in selecting the right research methodologies. Dr. Porter's expertise and invaluable teachings allowed my ideas to flourish, and will not be forgotten as I embrace the challenges that lay ahead.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

Recent and continued advancements in the field of information technology have permitted more and more physical devices to become ingrained with computers. Such devices are called Embedded Systems, and the term encompasses the large number of computers whose job is not primarily information processing, but is instead interacting with processes of the physical world [12]. Embedded Systems−which are typically miniaturized and of specialized design−employ their computational power towards interacting with the physical environment to achieve some specialized purpose. Such systems are typically reactive to their environment which places elevated importance on their timing constraints. For many years designers have sought to develop these systems as efficiently, and inexpensively as possible by pursuing minimalistic designs, and thereby cutting the cost of unnecessary computational features [12].

Traditionally Embedded Systems design has been an industrial problem, and mostly a problem of resource limitations. To cope with the small memory, small data word sizes, and slow clocks of embedded processors, engineers have historically strived to write low level software, to avoid feature-rich operating systems, and to develop minimalistic, specialized architectures and networks [12]. Within this traditional industrial setting a typical, modern design process for Embedded Systems is as follows:

1. Write the embedded code
2. Build the embedded system
3. Test for correct timing and function

This process is highly inefficient and insufficient for the development of comprehensive systems. The unavoidable problem with this process is the inevitability that functional and temporal constraints will not be met upon initial deployment. This would suggest an iterative design approach, meaning that the software engineers would have to go back to earlier phases of design and reconfigure the embedded code, component mapping, and hardware mapping. Then an entire redeployment of the embedded system would be necessary before the system could be tested for correctness again. For a vehicular system this iterative process of design, deployment, and verification can take years, and often requires several deployments before all requirements are met [22].

Poor understanding of design decisions and implementation details causes defects to surface during system deployment. These inconsistencies arise because designers must utilize multiple specialized tool suites that provide disparate representations of a system's computational behavior and physical dynamics. Software engineers must program by making educated guesses about the physical environment's reaction to the system's computational commands. Without a mathematical model to translate the software commands onto the reaction of the physical system, this process of trial-and-error and of lengthy design cycles is necessary so that the software engineers can slowly begin to understand how their device reacts once fully integrated and deployed in a non-ideal environment. This traditional design process does not lend itself to modularity, take for example the design of an electronic airbag: After the designers configure the inertial sensors—which trigger airbag inflation—for a particular vehicle, deploying this same airbag in a heavier vehicle would compromise the correctness of the system. The reality in designing these safety critical functions is that engineers typically tweak parameters and scheduling priorities in an iterative, ad hoc manner until the system tests correctly during deployment. A system designed using this process would be considered brittle, meaning that small changes in operating conditions or design would cause big changes in behavior [12]. Reconciling these changes may require complete reprogramming, rebuilding and recertification.

On the other hand, if we had a model to govern the physical processes of an airbag during crash, we could tweak the parameters of the vehicle in our model and then simulate how the system would react—all prior to deployment. The resulting design decisions would be founded in calculated physics rather than in trial-and-error. This moves the successive iterations inherent in the design process back into the early drafting phases, and away from the costly phases of componentization and deployment. Model-based design embraces the idea of evaluation and analysis starting from the beginning of the design process. A typical model-integrative design process is as follows:

1. Model the environment & dynamic behavior
2. Write the embedded code
3. Formally verify function and timing via simulation
4. Build the embedded system

The motivation behind this process is to drastically reduce the timelines for Cyber-Physical System deployment, by a factor of five [22]. From this academic viewpoint of system design, we see that extra effort is devoted towards accurately modeling the end-system, yet diverted away from time-intensive redeployments. The ability to formally verify the system through simulation is what allows us to move the building of the embedded system to later in the design process [4]. This design process is feasible because certain deployment-related phenomena appear during the extensive modeling process. With the knowledge gained from a model's simulations the final device can be built with more confidence, and with fewer deployments to achieve correctness of design as compared to the ad hoc approach.

This paper discusses model-integrating-computing (MIC) technology along with CyPhyML, a modeling language, and then presents a hybrid-engine controller application created using a specialized MIC toolchain. The paper concludes with an evaluation and a discussion of future research directions.

I.1 Motivation and Problem

I.1.1 Cyber-Physical Systems

Emerging from the abundance of interconnected embedded systems and their underlying networks is a new classification of devices, called Cyber-Physical Systems. These next generation systems are similar to Embedded Systems in that their primary focus is not solely information processing; however, the defining characteristic of Cyber-Physical Systems is that these systems maintain a tightly coupled relationship with their environment [12]. In this way, Embedded Systems comprise a topical subset of Cyber-Physical Systems with a stronger focus on real-time computational elements, as opposed to physical dynamics and cyber-physical interactions. Many Cyber-Physical systems are safety critical–such as fly-by-wire, adaptive cruise control, and nuclear engineering systems–and, therefore, must be able to react properly to physical conditions, even under fault circumstances. Because of their great potential, Cyber-Physical systems were classified by the National Science Foundations as a crucial area of emerging technology and research [27]. Like Embedded Systems, Cyber-Physical systems are fundamentally difficult to design and typically need to be deployed in order to test how the control software responds to the system's dynamics. For this reason, the design process for Cyber-Physical Systems is routinely iterative, and costly with many deployments.

I.1.2 Dynamic Simulations

It behooves the developers of Cyber-Physical Systems to employ dynamic simulations to test system behavior. Dynamic simulations allow analysis of the time evolution of a system due to the forces applied by the system's actuators [28]. In comparison, a static simulation would represent the solution to only the first timestep of a modeled system. The physical systems and engineering designs within a dynamic model are represented visually using block diagram notations [10]. Within the Modelica modeling language, the physical constraints imposed by the physical connections of the system are represented as signals which provide interconnections between the ports of equation-based models and blocks. Characteristic of dynamic modeling languages like Modelica is the support of acausal power ports at component interfaces. A well-formed acausal model corresponds to a similarly well-formed system of dynamic equations [24]. Acausal ports can represent the complex relations between physical components because such ports are bidirectional and carry a two-fold definition of the power of a signal, known separately as effort and flow variables. This type of connection, unlike a functional one-way data flow, is well suited for expressing the physical constraints imposed by the interactions within a dynamic system. For example, an electrical power port shares the variables of voltage (effort) and current (flow) along its connections with other electrical components.

Once a model has been composed and checked for structural correctness, a dynamic modeling environment is able to translate the subsystems and connections within the model into equations which can be simulated and analyzed [24]. Simulations tools perform this task by consistently evaluating and updating the current state of a system starting from its initial state [5]. Typically a dynamic simulation is linked to the system's real-time control software. Within a model, the designer can create realistic use scenarios and then test the consequence of the proposed actions. With these capabilities, designers are able to validate and verify−with reasonable assurance−the behavior of their system in its intended environment, all before system deployment or even componentization [6], [5]. The knowledge feedback from dynamic simulations serves to replace the feedback gained from costly deployment cycles, and leads to faster development timelines.

There are some disadvantages posed by relying on simulation. For example, it can be expensive to measure exactly how one component affects another within a physical system in order to formulate an understanding of the system model. In this way, it takes time to create a system model that instead could have been devoted towards developing the actual system. However, our approach to system design advocates that the extra time spent to comprehend the intricacies of a system is invaluable in terms of the resulting ability to verify system properties prior to full implementation. With simulations, critical situations can be investigated without any risk or danger of faults. Furthermore, simulations can perform lengthy experiments which would be infeasible to test with a physical implementation, by leveraging a sped up notion of time,

I.1.3 Software Modeling

The computational aspects of Cyber-Physical Systems are governed by control loops and data flows, which could cause a system to behave differently depending on how the control elements are implemented within software and hardware. A control loop is the underlying process management scheme of a system that determines how a process variable is maintained at a set point, notwithstanding the destabilizing effect of external disturbances [9]. How this process is allocated within the confines of software architecture is termed the data flow: the abstract functions which passed control signals become instantiated as tasks which send messages between software components [10]. Furthermore, hardware allocation refers to how these software tasks are deployed onto a platform. Allocation details cover the mapping of tasks onto processing nodes, and messages to network interfaces. The entire process of mapping a control loop onto software, then onto hardware is called componentization. It is typically a costly and vaguely specified process that occurs prior to deployment [10].

To avoid the need of ad hoc componentization and deployment prior to system-level testing, system engineers have embraced the idea of software modeling. Software modeling supports educated project planning, allows detection of faults, and provides a rationale for refining code [30]. Most-importantly, as with dynamic simulation, software modeling allows the designers to evaluate and analyze system performance and timing. Software modeling tools, like the proposed ESMoL tool suite, facilitate the process of conforming a control-loop to proper software abstractions. Moreover, complex componentization concepts such as shared memory, partitioning, and distributed processors can be easily specified and simulated by mapping abstracted software components onto hardware platforms. Software simulation can be done with high assurance because tool suites allow the designers to specify crucial timing aspects such as clock speeds, worst case execution times, and network interface delays.

The described process for CPS design falls apart in practice due to inconsistencies across tool suites, which lead to ambiguous definitions of crucial design components. The resolution of ambiguous details between design domains and subsystems in a Cyber-Physical System is the main cause of failures during deployment and, therefore, project overruns [29]. Thus full system-level integration is not automatic or intuitive, and instead requires a specialized design solution for incorporating the signal-flow, software architecture, and hardware architecture aspects of system. In this sense, implementation becomes an extra design phase in which designers must painstakingly adjust integration details until the modeled behavior is exhibited in the integrated system.

I.1.4 Model-Integrated Computing (MIC)

In this paper we describe the META tool suite as well as its accompanying languages, which realize the broader Model-Integrated-Computing (MIC) approach to system design. MIC is emerging as a powerful and efficient means for designing, maintaining, and progressing large-scale computational systems such as Embedded Systems, and Cyber-Physical Systems. The MIC technique employs a model-based approach to system-level software development in which users are able to synthesize deployable program code from the formalized models of META's heterogeneous integration language, CyPhyML [3]. On top of capturing the requirements of the system, models in CyPhyML also explicitly capture details of the system's embedded software and environment in a multi-domain, heterogeneous manner [1]. No longer are models obscure or isolated representations of non-practical system behavior, but instead provide a reliable means for analyzing computational and dynamic behavior wherein model subcomponents correspond to real physical components.

The Model-Integrated approach reconciles the issue of incompatibility and ambiguity across specialized tool-suites by aggregating numerous aspects of system-level design together into a single unified semantics/representation [29]. This means that the crucial yet obscure implementation decisions no longer take place outside of the modeling framework in an ad-hoc fashion during manual deployment. The META approach permits the full integration of multi-domain, specialized models all within a single tool suite so that implementation details can then be carefully modeled, and done so within early phases of design. This capability shortens the iterative design cycle by allowing valuable integration-time simulation analysis, verification, and validation to take place prior to the costly deployment phase [1]. To capitalize on the numerous models which already reside in existing engineering and analysis tools, CyPhyML allows the seamless–yet specialized–exchange of semantic definitions from popular tool suites like Simulink and Modelica [4]. This semantic exchange seeks to foster the adoption of these Model-Integrated techniques by system designers who are already proficient with the current-generation of design tools. The Model-Integrated-Computing approach permits full realization of the aggregate design goals sought by these current tools, to create a next-generation design suite capable of efficient and effective modeling and simulation of Cyber-Physical Systems.

I.2 Approach

I.2.1 Integrated Dataflow Programming

In our attempt to model an electrical Cyber-Physical system, we require a modeling framework that will allow us to compose physical dynamical components alongside control flow components. ESMoL currently lacks a free and open language for representing synchronous control logic. The MIC approach does not warrant switching tool suites without model exchange, thus creating a similar model which is semantically unrelated. Therefore, we require a Domain Specific Modeling Language (DSML) integrated within CyPhyML to support the hybrid composition of control systems with attributes of physical dynamics.

With this said language, it would be possible to use discretely modeled software functions to enact change in continuously evolving continuous-time models, and vice versa. CPS are tightly coupled with their environment, and we need to be able to represent this tight coupling within our design suite so that we can fully embrace the MIC technique: If we are able to compose these different Models of Computation (MoC) together, then we will be able to test embedded CPS software with high assurance by being able to analyze, evaluate, and verify formal system properties from a heterogeneous simulation, without first deploying the system.

Since we desire flexibility and interoperability, such a DSML must be able to exchange models with other dataflow programming suites. Additionally since the ESMoL suite focuses on model-based deployment, atomic components should correspond to C code which can be easily componentized and executed on diverse hardware platforms. Through graph transformation techniques, this underlying C code would allow the system model to directly generate the specified system software. Behavior of the dataflow simulations should accurately match up with the eventual deployed behavior of the system.

I.2.2 Solution Summary

Therefore, we propose the Signal Flow language and block library as a sublanguage of CyPhyML to provide heterogeneous composition of control systems and dynamic models. The Signal Flow metalanguage can interact with non-Signal Flow blocks within CyPhyML, closing the semantic gap between physical components and functional components. Our work extends upon an older Domain Specific Modeling Language (DMSL) for functional modeling which had previously relied on a closed-source backend. Signal Flow models are semantically linked with dynamic models by means of the CyPhyML integration language. The Signal Flow DSML is an open-source dataflow programming language, similar to Simulink or LabVIEW, designed specifically for modeling embedded systems and their underlying computational semantics. Since the continuous-time, high-precision simulation environment of Simulink has gained widespread acceptance in the control and modeling community, we have devised a way of exchanging models between Simulink and Signal Flow. Signal Flow's block library bears many of the same constructs as the Simulink library, so that the large preexisting pool of Simulink models can be easily regraphed as Signal Flow functional blocks. Moreover, Signal Flow models can be transformed back into the Simulink environment which may be desirable if the user wishes to employ TrueTime kernels as a basis for simulation. As desired, the Signal Flow language is capable of rewriting the dataflow model as C snippets, so that its models can be used to directly generate the software expressed by its models.

In this work, our primary goal is to offer a fundamental understanding of the use of MIC towards Cyber-Physical System design. To better demonstrate this process, we propose the design of an electrical Cyber-Physical System: a bidirectional DC/DC power converter. This type of electronic device integrates into a hybrid-electric vehicle and regulates power to and from the vehicle's electric battery. The converter's controller functions by flipping electrical switches to alternate the direction of operation. Within either operating mode, the circuit opens and closes a NMOS switch with high-frequency to regulate the voltage at specific nodes within an RLC circuit. Input signals from the vehicle, such as brake-down or accelerator-down, govern whether or not the converter is driving the electric motor, or charging the battery.

In this proposed design, we will model the power converter circuit as well as the software controller. The MIC framework will allow us to model the dynamics of the converter circuit in conjunction with the software logic of the controller. In order to represent the synchronous semantics of the controller logic, we will employ the SignalFlow language to import the controller design from a Simulink model. The controller dataflow will be paired with the DC/DC converter circuit which it will drive within the CyPhyML environment. The heterogeneous model will allow us to create simulations in the dynamic simulation environment of Modelica. Our goal will be to analyze these simulations in order to verify the stability, and appropriate behavior of our bidirectional converter as it transitions between five different operating modes. The final model will allow us to easily generate deployable code so that the converter can be tested in the context of the physical circuit.

I.2.3 Related Works

Embedded and Cyber-Physical Systems modeling is not new, and there are already numerous and diverse approaches towards Embedded and Cyber-Physical Systems modeling. Similar to the approaches of Ptolemy II [33] and ModHel'X [34], our CyPhyML language is centered on genericity, composition and integration. The compositional approach to heterogeneous modeling means that the ability to "glue" disparate models together is inherent in the underlying language framework. Unlike CyPhyML, however, Ptolemy II provides support for a diverse range of Models of Computation on the condition that the outputs of Ptolemy models (or actors) are functional with respect to their inputs and internal states. The strict definition of input and output ports means that model interfaces are casual and one-directional, limiting the scope of physical components that can be represented. In our approach, ports can be directionless and interconnections between components outline strict mathematical constraints between variables at the interfaces. This allows for more complex equations to govern component interactions, like the sharing of flow and effort variables across interfaces in order to represent linked dynamical quantities such as current and voltage, or position and force.

Hardebolle and Boulanger [31] provide a canonical overview of the current state-of-the-art of heterogeneous modeling approaches. In this summary, the CyPhyML language stands out because of its support for an open set of modeling languages, and also because of its underlying metamodeling framework. Some tools, such as Kermeta [35] and AToM3 [36], also employ a metamodeling framework for model composition and translation. Kermeta seeks to attach strict operational semantics to the elements of a metamodel, however, it provides only the fundamental framework for the future prospect of weaving executability into metalanguages: it would require specialized effort to apply these principles onto each desired Domain Specific Modeling Language. Our ESMoL framework already contains developed behavior models for many metalanguages catered towards Cyber-Physical System modeling, and supports rich semantic exchange between DSMLs. Additionally, the operational semantics of CyPhyML are provided by the Modelica language. The AtoM3 metamodeling language is strikingly similar to CyPhyML in that metamodels are used to describe Domain Specific Languages, and the AtoM3 or CyPhyML environments themselves can automatically function as tool suites for specifying the resulting metalanguages. AtoM3 does not support some key modeling fundamentals such as hierarchical composition, or built-in model exchange. Furthermore, AtoM3 does not support the continuous evolution of a metalanguage as the needs and scope of the project change. Updated metalanguages have difficulty importing designs created in older versions of the same metalanguage in AtoM3. By comparison CyPhyML can update model files from upstream metamodel changes, threby permitting a continously evolving metamodel.

Some approaches have arisen from coming to terms with the fact that tools which are used in different phases of the design cycle are based on different theoretical foundations and, therefore, it is too difficult to maintain consistency of the models between verification tools and implementation tools. For example, the Rosetta [32] specification language implements a multi-aspect framework similar to CyPhyML and is capable of representing different views and domain characteristics of a component. Additionally, the language bears excellent aspect merging capabilities for supporting formal verification and validation. However, the highly-abstracted Rosetta models are not directly executable, meaning that its users must switch to a different tool for the design phases of the development process, and redraw a similar model. This “similar models” approach stands in high contrast with ESMoL approach in which models used for specification and composition are the same models used for analysis, verification, deployment, and execution.

CHAPTER II

Background

II.1 Cyber-Physical Systems

While Embedded Systems have been in existence for decades, a new “revolution” in information technology is mounting due to the increased pervasiveness of Embedded Devices, and their proliferation into everyday life. As stated by the National Research Council, “IT is being embedded into a growing range of physical devices linked together through networks,” and is doing so with continued momentum as technologies become smaller, faster, and cheaper [13]. The Research Council predicts that the collectivization and interoperability of these numerous devices will yield even greater systems, causing a revolutionary shift in how we interact with the physical world: “These networked systems of embedded computers...have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways” [13]. These proclaimed integrative systems have become commonly known as Cyber-Physical Systems.

To approach the problem of modeling these complex Cyber-Physical Systems, we start by introducing three fundamental models of computation. Each of these models is crucial in representing the behvaior of computational and physical processes. Our model-integrated approach aims to unify these mixed domains in order to describe a comprehensive system in a single framework.

II.1.1 Synchronous Reactive Systems

As we have just discussed, Embedded and Cyber-Physical systems mix the hardware and software of digital controllers with the continuous dynamics of physical systems. These systems are intrinsically Heterogeneous, in the sense that such systems combine asynchronous events, periodic timed components, and continuous dynamics [14]. Of these

Models of Computation we will first delve into the one based on asynchronous events: synchronous-reactive systems.

Real-Time Systems are precisely defined synchronous reactive systems, in that they are reactive systems subject to externally defined timing constraints [16]. However, formal synchronous-reactive systems are devoid of the idea of time; therefore, they are not time-triggered. Rather these systems operate only in reaction to their environment, and−in a sense−run at the speed of their environment. In this way synchronous reactive systems maintain a permanent interaction with their environment [15]. It behooves us to apply this synchronous-reactive model of computation to Cyber Physical systems which foment a close relationship with their operating environment.

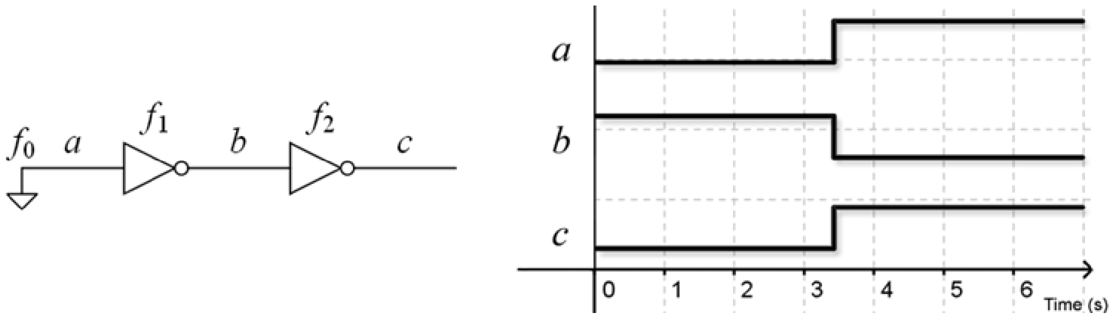


Figure II.1: (a) On the left we have an example SR system. (b) The traces of these NOT blocks in response to a change at time = 3.45s

An example of a synchronous-reactive system would be the Logical NOT component. (Diagram of logical NOT) Operating within the confines of Boolean logic, the Logical NOT simply inverts its input signalwhich will be considered the environment in this isolated example. When implemented a Logical NOT essentially remains passive and holds the output value until an instant of change in the environment alters the input signal. When the instant of change occurs, the system immediately inverts the output so that its function holds true wherein the output is always the logical inverse of the input. This system is reactive in the sense that instants of change are initiated by environmental events, and also synchronous because the notion of time arises within the model as a series of ordered instants

[17]. To clarify the system's operation is not driven by or reliant on time, yet the concepts of "Before", "After", and "At the time same" are all well-defined and controllable. These attributes make synchronous-reactive systems easy to describe as a model: such models compose well into other models of computation by permitting deterministic, hierarchical, concurrent programming [16].

By continuing to study the Logical NOT example, we see that the component is a proper synchronous-reactive system because it operates instantaneously. The logic block's outputs are produced in synchronization with its inputs. Therefore, the Logical NOT is only operational for an infinitesimally short instant when triggered by the environmental input (an abstraction of the electrical current swing) and then remains passive for all other time. Nothing happens between the instants of synchronous-reactive systems and the inverter's continuous-time signal inversion is abstracted as an instantaneous step event. Because no operations take place during between instants, a deployed synchronous-reactive system only needs to be fast enough to simulate synchronous behavior by ably responding to each discrete event [17].

Because of these described properties, synchronous-reactive systems are relatively easy to model. These systems are simple because they have no inherent notion of time. From a programming perspective, this means that components only need to be called or invoked when a change in input occurs. In practice the simulation behavior of a fully synchronous-reactive system much resembles a chain-reaction. Consider the picture chain of Logical NOT blocks as in Figure II.1a: When a change causes 'a' to flip from LOW to HIGH, the $f1$ block reacts by instantaneously changing output b from HIGH to LOW. Then without any time transpiring $f2$ detects a change in its input, and reactively flips the output to the correct value. This behavior is depicted in Figure II.1b. From an observation standpoint, the changes in signals 'a', 'b', and 'c' all occur concurrently at the exact same simulation-time.

II.1.2 Discrete-Event Systems

In discrete-time systems, phenomena change value or state at discrete points in time, rather than continuously or as a reaction to external stimuli [18]. What this means is that a discrete-time system will not reevaluate inputs if an input change occurs between 2 discrete events: changes can only transpire after a specified−and often periodic−interval has elapsed, at a specific point in time. This is different from synchronous-reactive systems which do not have delays or any conceptualization of time, and instead evaluate changes in input immediately [15]. However, a discrete-event model suddenly resembles a synchronous-reactive model if we ignore time and focus only on the sequence of events. In a sense, the synchronous-reactive model is an abstraction of the discrete-event model [14]. So while both models of computation were subject to instants of change, real numbers will now represent time increments between the instants of the discrete-event model.

The Signal Flow metalangauge for modeling control loops obeys discrete-event semantics. Each block contains an intrinsic samplePeriod variable which determines the frequency of invocation. At the end of each period, a Signal Flow block will call its governing C code and produce functional output with respect to the current input. This semantic definition is sufficient for describing the computational process execution that we seek to model with the Signal Flow language.

Let's examine a climate-control system as an example discrete-event system. When the system detects that the room temperature has raised above 70 degrees, the controller starts the air-conditioning unit. Within this system the input temperature changes continuously, yet the temperature sensor performs a sample every 60 seconds. This specific process is only active once every minute, but reliably so. Unlike a synchronous reactive system, it is not an input change which triggers a system event: Rather, the system can only "see" the physical temperature input at specific discrete-events which occur periodically every minute [14]. If the temperature rises above 70 degrees only 2 seconds after a temperature sampling event, then the climate control system will not be able to react for another 58 seconds−when the sensor is next scheduled to operate. Regardless of the fluctuations in temperature, the climate control unit will experience events at 0 seconds, 60 seconds, 120 seconds, and so on. Unlike their synchronous-reactive counterparts, discrete-event systems may retain the same previous inputs and outputs even as an event related invocation occurs.

Discrete-event systems are useful to model because most real-time systems operate according to discrete-event principles. Both systems are defined by their logical correctness as well as their temporal correctness. Furthermore, the scheduling of discrete-event systems very closely resembles the sequencing and scheduling of real-time processors [18]. Take, for example, a typical 1GHz processor which not continuously evolve, nor is its operation a sole reaction to changing input. Instead, the computer processor in operation for infinitely small instants of change which periodically recur every nanosecond. The processor is time-triggered to process inputs at an interval set by the clock crystal. This behavior is very closely modeling by a discrete-event model of computation.

Discrete-event systems simulate similarly to synchronous-reactive systems, but with one small change: a discrete-event component is allowed to schedule itself to run again in the future. Models that obey these discrete-event semantics would no longer react asynchronously to changes in input. This periodic rescheduling is accomplished through persistent creation of super-dense time tags. After a discrete-event component executes it will create a tag for itself to run again in the future, with the specific time determined by its operating frequency. To give an example, a computer's main 1GHz processor might trigger itself to execute 1 nanosecond later in the future. The tag would resemble $tag_{main,\ 1} = (t + 1ns, 0)$. If the graphics processor operates at 4GHz, then after an event instance it would create a tag of the form $t_{graph,\ 1} = (t + 0.25ns, 0)$. These multiple tag requests create an ordered event-queue. It is the job of the simulation tool to walk forward in time and invoke discrete-event elements according their tag-time. This is how the notion of time is instantiated within a discrete-event model. Presumably, the inputs to the model will change along with the passing of simulation time. If the output of a discrete-event block flows into a synchronous-reactive block, then the synchronous-reactive element would be triggered and would operate instantaneously at the instant of the discrete-event. This execution model interacts smoothly with synchronous-reactive elements because both models of computation are semantically similar. With the addition of periodic rescheduling abilities, there is now a measureable notion of time between instants of change within the model, so that any discrete-event system can be properly described and simulated [14].

II.1.3 Continuous-Time Systems

Whereas the discrete-event systems that we have rehashed center around the concept of "when", continuous-time systems focus instead on the concepts of "eventually" and "always" [19]. Signals in these systems do not only undergo change at discrete intervals, but rather are persistently evolving. This is akin to the behavior of physical processes which act as continuously evolving systems. System designers and programmers have come to terms with the fact that models for the computation and the physical world diverge and while not suited for simulating aspects of computational systems, continuous-time models are perfectly suited for modeling physical dynamics [12]. Continuous time models do not undergo discrete instants of change, and instead are constantly evolving along a time continuum with their behavior defined by ordinary differential equations. Concepts such as rotational motion, temperature change, current flow, and air resistance are naturally described by differential equations and are a perfect fit for the continuous-time model.

The Simulink and Modelica languages employ an underlying continuous-time framework [25], [21]. Therefore, modeling physical plants—for example through state-space models—is intuitive, and well supported. Because the discrete-event MoC is an abstraction of continuous-time semantics, these languages are capable of implementing discrete-event blocks which have configurable periodic properties [21]. Modelica is suitable for simulating the execution of mixed MoC's and will be used for analyzing the interaction between discrete computations, and continuous-time elements for our DC/DC converter system [25].

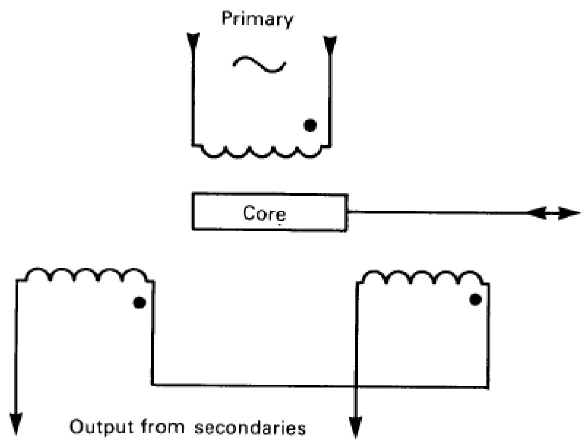


Figure II.2: Circuit layout for a Linear Variable Differential Transformer Displacement Transducer

Let's consider a Linear Variable Differential Transformer Displacement Transducer, or LVDT for short. An LVDT is essentially a high-precision position sensor. Its construction consists of a cylindrical array of solenoid coils aligned adjacently, and placed around a ferromagnetic core as visualized in Figure II.2. The ferromagnetic core is placed around the object whose position is to be measured. An alternating current drives excitation in the center primary coil, and the resulting magnetic field induces current in the secondary coils (as in an electrical transformer). The displacement is measured as a function of the voltage differential between the two secondary coils. When the solenoids are at the center of the ferromagnetic core as in the leftmost diagram of Figure II.3, the differential is precisely zero since the secondary solenoids are out of phase by 180 degrees. However, as the solenoids slide positively away from the core, there becomes an increase in voltage at one of the secondary coils and a decrease in another as seen in Figure II.3. The amplitude of the resulting voltage differential signifies the amount of displacement along the core. The phase of the output allows the system operator to know in what direction the displacement takes place: positive displacement means the differential signal is in phase with the primary, and negative displacement means the signal is out of phase with the primary.

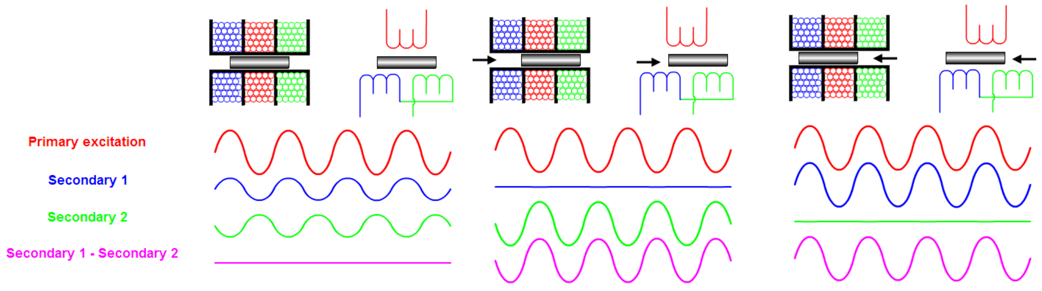


Figure II.3: Behavior of an LVDT across 3 different operating conditions

This device fits into the realm of continuous-time models because its operation is governed by the analog behavior of its circuit. This is not a digital system which undergoes change at clock ticks. Instead the system's currents, fluxes, and voltages are all constantly evolving according to position and as governed by differential equations. Even if the system's input bears discontinuities, the system response will maintain its continuous characteristics. Because the system has analog voltage signal as output, it can provide boundless precision. Unlike discrete-event systems, the signals of continuous-time systems do not pause between the various instants of change.

II.2 Dynamics Simulations

II.2.1 Modelica

Modelica is a modeling language well suited for modeling large, complex, and heterogeneous systems [20]. Unlike other languages, Modelica has been specifically designed to facilitate the exchange of models, model libraries, and simulation specifications [21]. These models, together with connections and blocks, make up the fundamental building blocks of the Modelica environment. Models can be composed hierarchically, can support multi-domain capability, and can involve mixed models of computation [24]. Modelica's multi-physics engine provides the user with the opportunity to combine electrical, mechanical, hydraulic, and thermodynamic components together in one model [20]. Additionally, Modelica is designed for the modeling of continuous-time and discrete-event physical systems and, therefore, is able to solve continuous-time equations as well as discrete equations [21].

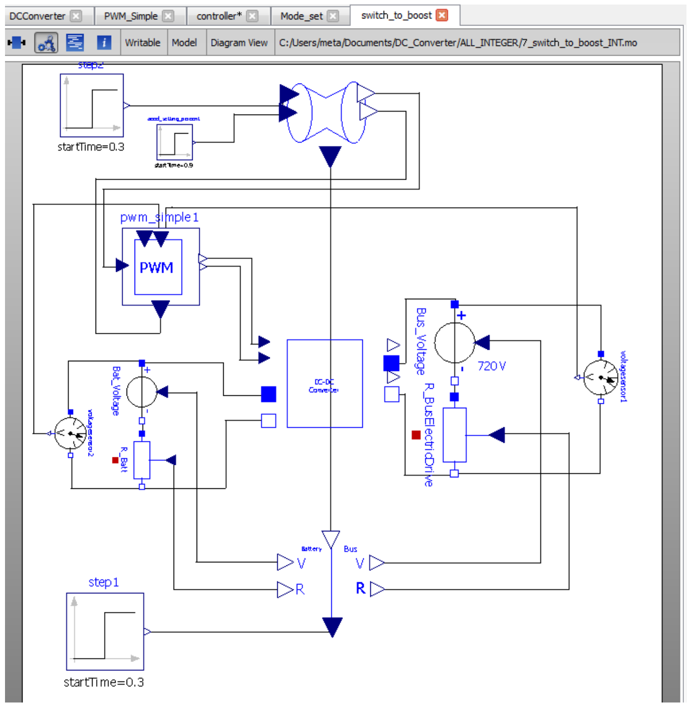


Figure II.4: Example of a mixed domain model in Modelica

Modelica's models use equation-based signals to express the physical constraints imposed by the physical interconnections within a system [24]. These connections are created simply by clicking on both endpoints within a model's schematic, or object diagram. The models themselves, like resistors or gears, can be added to a model in “lego-like” fashion by dragging them onto the object diagram from the predefined object library [21]. The model components can be comprised of additional object diagrams, however, atomic components are defined by an equation based description of the model behavior. At simulation time a Modelica compiler will translate all of the model's subsystems and connections into equations which permit simulation and analysis. The acausal nature of Modelica's connection ports supports simulation of the continuous processes of physical dynamics. Unlike typical casual ports, acausal ports represent a simultaneous, bidirectional energy exchange between components [24]. For example, Modelica's Electrical Pin interface contains both voltage (effort), and current (flow) variables. Modelica compilers enforce the rule that effort on both ends of a connection remains equal, and that flow on both ends of a connection sums to zero. (In terms of electrical connections, this rule enforces Kirchhoff's Current Law.) This acausal support permits the modeling of complex continuous, physical dynamics.

If we examine Figure II.4 we get a glimpse of a mixed-domain model within Modelica. The large blue and white squares which appear on the periphery of components are the bidirectional acausal pins. Each component has at least two pins corresponding to the component's positive and negative terminals. Also visible in Figure II.4 are the causal signal ports, which resemble blue and white triangles. The purely computational components—like the vehicle controller—possess only causal ports, and the ports are strongly typed as input or output signals. This type of component interface is not well suited for physical dynamics, but is adept at modeling computational processes. Hence, the illustrated model is a mixed-domain model because it composes causal computational connections with acausal physical connections.

The Modelica language is configured to allow the generation of efficient simulation code automatically by the compiler. An example of one such compiler is OpenModelica, an open-source editor that also permits the graphical editing of object diagrams. Since its first version in 1997, Modelica has become warmly accepted by the modeling and simulation community [21].

II.2.2 Simulink

Another popular simulation tool is Simulink, which is centered on continuous-time simulation [25]. Unlike Modelica, Simulink is a closed-source language; however, both tool suites provide a similar "drag and drop" type modeling environment. Both tools are adept at modeling and simulating multi-domain dynamic systems, and hybrid models of continuous and discrete components are permitted. Simulink is a causal modeling environment,

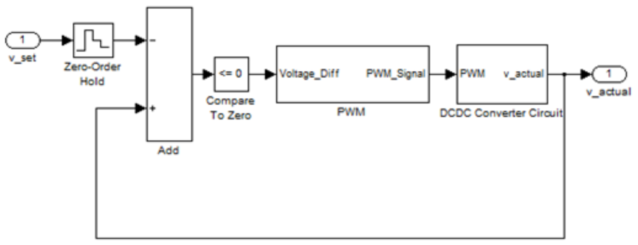


Figure II.5: Example of a Simulink control loop

so all signal flow diagrams have distinct input and output ports [24].

Figure II.5 shows a view of a control loop within Simulink. The model has one input signal, v_set, and one output signal, v_actual. The input signal is connected to the input of a Zero-Order Hold Simulink block which discretizes the input signal at a given sample period. The output is relayed on the right side of the block. Akin to Modelica's causal ports, all Simulink ports are strongly typed as either input or output so the input signal itself is not altered by the block. In the diagram, a chain of blocks manipulate the input signal until it finally reaches the output. This specific control loop determines the amount of error in the output signal compared to the input signal in trying to regulate a target output. The ouput is used as a feedback loop and is summed with the input through the Add block. The output of a block can be replicated and distributed to multiple other Simulink blocks.

Simulink's semantics and syntax make it especially proficient at modeling software and control loops. Once designs are complete, Simulink then allows users to automatically generate C code for deployment on the operational platform [25]. Simulink has gained widespread acceptance within the control and modeling community with many high-profile project implementations.

II.2.3 Software Modeling

Embedded Systems designers are undoubtedly faced with the issue of conceptual incompatibility across mixed semantic domains, since they are highly specialized and since tool suites for these aspects of design diverge with no single unifying model. The described process is lengthy and often requires exorbitant manual rework, and failing to develop a cohesive system model can lead to unintended disruptions in the design cycle. For example, the seemingly innocuous change of a faster data-rate−while compatible in the signal-flow domain−could completely invalidate the system when implemented within the software and hardware architectures. The current solution to avoid ensuing rework is to program conservative performance margins in anticipation of an eventual integration with the hardware platform. Still, peculiarities arising from the choice of hardware or software architecture may not surface until late in the design cycle, or until the system is fully deployed [10]. Long development, deployment, test cycles limit the amount of iterative rework allowed within a timeline, and often schedules do not permit going back and reworking early design phases. However, current state-of-the-art still requires designers to backtrack and rework earlier stages of design in order to patch these issues due to faulty understanding of the interaction between control, software, and hardware [29]. If the domain-specific modeling tools bare different interpretations of a model across tool suites, then the resulting model will become invalid when integrated together. The solution of structuring and patching tool suites to create consistent, exchangeable model definitions is time-consuming in itself, and is not warranted unless the finished products are applicable to multiple different designs.

II.2.4 Modeling within Simulink

Simulink, developed by MathWorks, is widely used for the modeling and simulation of embedded system controllers. Its popularity has led to the formulation of many specialized and third-party function libraries, namely the “Real-Time Workshop Embedded Coder”, and “Aerospace Toolkit.” Specialized functions aside, Simulink provides the user with the fundamental control system formalisms such as Integrators and Transfer Functions.

For example, Figure II.6 shows a simple control loop within the Simulink environment. Starting on the left, we observe that the signal from the input Step Function flows into a

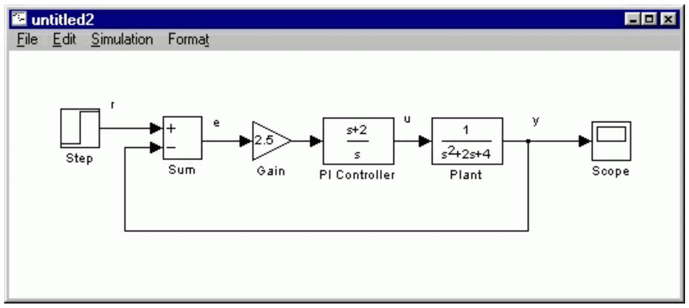


Figure II.6: Example of a Simulink control loop

Sum block where it gets added to negative feedback. The output flows next into a familiar Gain block, followed by a PI Controller block. Finally, the signal flows into plant dynamics, as represented by the Transfer Function block. The functional meaning of these blocks, as well as their graphical syntax is standard in the modeling and control community.

Simulink provides a robust continuous-time environment for coupling plant dynamics with discrete-event signal-flows. Hardware-in-the-loop simulations are also possible. Many design engineers prefer Simulink because its syntax checker can accurately detect faults and algebraic loops within a design. Furthermore, its debugging engine is powerful making Simulink easy to adopt. It has been used for modeling the signal-flows of automotive, avionic, and embedded systems in general. Simulink has received widespread adoption among control engineers, largely due to its flexibility in accommodating different types of diverse design problems [26].

II.2.5 Embedded Systems Modeling Language (ESMoL)

ESMoL reconciles the problem of cross-domain modeling, by incorporating the multiple aspects of dynamics, signal flow, software architecture, and hardware architecture together into a single multi-aspect design environment. ESMoL, created at Vanderbilt University, is a flexible tool suite for creating Embedded Systems that aims to provide a resolution of semantic inconsistency between tool suites [10]. ESMoL's designers took into account the fact that Simulink has become a de facto standard for control and signal-flow composition and, therefore, have allowed ESMoL's users to import existing Simulink and Stateflow models, while still permitting native control loop specification. Within the signal flow paradigm ESMoL possesses many functional blocks that resemble Simulink blocks, and its simulation framework similarly supports synchronous discrete-event semantics, so that no information is lost during the model exchange. What sets ESMoL apart from Simulink is that ESMoL permits the componentization of working signal flows in order to define the actual data flows on the deployed software platform. Details of the target hardware can be linked with the software components, leading to a holistic definition of the modeled system [7]. These are among ESMoL's many processes and techniques which aid in the development of safety-critical embedded systems:

1. The ESMoL tools confine the system design to a single design environment with multiple aspects for representing mixed-domain details of design. This unified environment allows the composition, analysis, and simulation of models as well as the synthesis of deployable program code. ESMoL employs the principles of MIC by regarding modeling as the central activity for system development [3].

2. ESMoL aims to shorten design cycles by providing models with strong analysis and simulation capabilities. ESMoL permits a rapid transition to deployment by supporting code synthesis techniques such as hardware-specific task code generation, as well as data communication code generation.

3. ESMoL allows the definition of deployment related information, such as the mapping of control loops to software constructs, and the mapping of software constructs to hardware platforms. Simulations execute under the constraints of the software and hardware mappings.

4. The two-stage design of ESMoL model interpretation isolates the interpreter code from ESMoL's compositional framework. The motive for this approach is to allow the flexible integration of simulation tools and code generators. The syntax

and semantics of an ESMoL model are first transformed into the intermediate ESMoL_Abstract language. Then simulation and analysis tools create feedback and deployable code from this intermediary representation of the model.

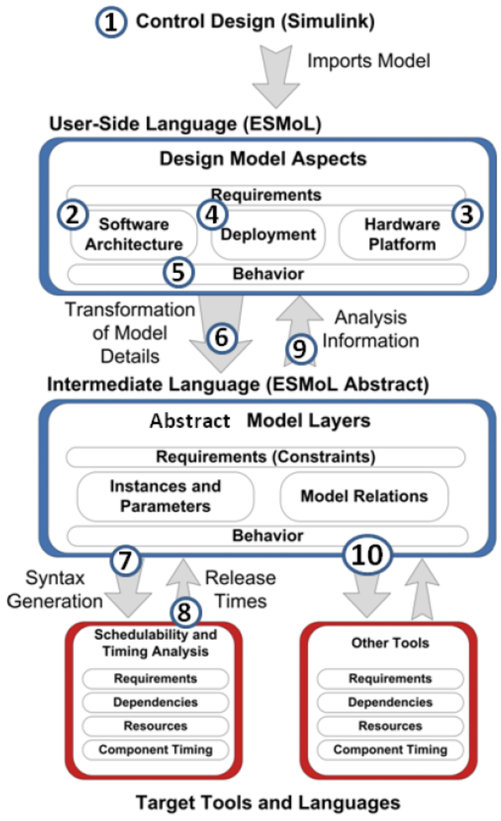


Figure II.7: ESMoL design phases

Figure II.7 shows a process flow that encompasses all phases of ESMoL development. Starting from the top of the diagram at Step 1, the first step in the ESMoL design flow is importing the control design from Simulink. Once the Simulink design is within CyPhyML, Step 2 covers the mapping of the control loop blocks to software processes, and signal flows to data messages. In a similar fashion these hardware components are mapped onto hardware platforms and network interfaces as per Step 3. Development at Steps 4 and 5 covers the specification of timing and scheduling details within the hardware platform. At this point in the process, the model is sufficiently defined to be transformed into the intermediate ESMoL_Abstract language putting all relationships and structural models into a consistent representation (Step 6). A specific interpreter is chosen for Step 7 to generate the simulation code for the model (in our case Modelica), and then this information is fed back into the design to support optimal design modifications (Steps 8 and 9). ESMoL is also capable of synthesizing deployable code for target platforms directly from the models which allows for rapid deployment techniques (Step 10).

The goal of ESMoL and of its unified semantics is to shorten design cycles by automating the software and hardware integration stages of design. As discussed earlier, ESMoL is able to compose Simulink control structures within software and hardware concepts to define the final system implementation. Since componentization information is captured within aspects of the model, ESMoL can automate the deployment process. During simulation, Simulink blocks become realized as C language templates whose code is automatically organized according to the hierarchical and architectural mappings within the model. Simulation models can either be run on the tool suite computer, or on a specific platform to accurately observe the behavior of the model upon target system. If the system does require rework after a simulation, ESMoL facilitates the backtracking process by feeding simulation information back to earlier stages of design. For example, timing information from the simulation is saved and made available in the signal-flow and timing aspects, making it easy to reconcile missed execution deadlines.

The goal of automating these design, simulation and analysis tasks is to foster quicker iterations within the development cycle. With the current state-of-the-art, inconsistencies across tool suites can effectively distort designers' perceptions of a system and mislead them into developing a separate design solution, unrelated to the specified requirements or unable to be implemented. The ESMoL tool suite evades this issue (and additional design iterations) by unifying the multiple design domains within a single multi-aspect environment. The intended goal of ESMoL is for the overall project to move towards a correct implementation that most accurately reflects the designers' understanding of the design problem [10].

II.3 Model-Integrated Computing

Model-Integrated-Computing(MIC) addresses the needs of creating complex multi-domain systems by adopting a fully model-integrated approach to design at all levels of design. In this sense, models not only achieve the objective of designing and representing a system, but also provide the main avenues for analyzing, testing, generating, simulating and even operating embedded systems [1].

Models both become the design language and the analysis tools that underlie every level of system design from beginning to end. As opposed to an animator's model−for example−of a story board which mostly serves as a rough palette for initial brainstorming that is ultimately scrapped, MIC embraces models as the very foundation and continuously evolving architecture of the desired system. Rather than looking to the system model for an oblique outline during design, within the MIC approach the model is actually the platform in which the designer creates and programs the system. Modeling serves as the metalanugage for which system information is drafted, designed, analyzed, simulated, and even deployed [3].

By embracing MIC, we can create metalanguages that represent all necessary system information in the form of models. These languages represent not only the system's operation, but also physical dynamics assumptions about the system's environment [1]. With this specialized approach, we guarantee a tightly coupled design process where we can model the integration constraints and analyze not only the standalone software operation, but also the operation of the software system in its environment.

In our model-based approach we employ metamodeling to compose, create and develop modeling languages. We use the GME editor to define metamodels, and also to compose system designs within the defined metalanguage. CyPhyML integration language is the metalanguage for composing system models from distinct components, which is done in GME. Components can be defined by C-code snipets, or alternatively by references to Modelica blocks since CyPhyML employs the behavorial semantics of the Modelica language.

The Signal Flow modeling language, which is compatible with CyPhyML is our approach to discrete and synchronous software modeling. The DC/DC converter utilizes the Signal Flow language for composing the vehicle controller, and our work with the converter included an expansion of the Signal Flow language. Our model-integrated approach through CyPhyML allows both computational and physical components to be composed together within the same design.

II.3.1 Metamodeling

Metamodeling is essentially language specification. Metamodeling is a form of modeling, when the system being modeled is the process of creating models [3]. This type of language specification is highly mutable, and can evolve new features and functionality as needed. Metamodels become instantiated into a graphical language with functionality determined by the class stereotypes and attributes of the metamodel [10].

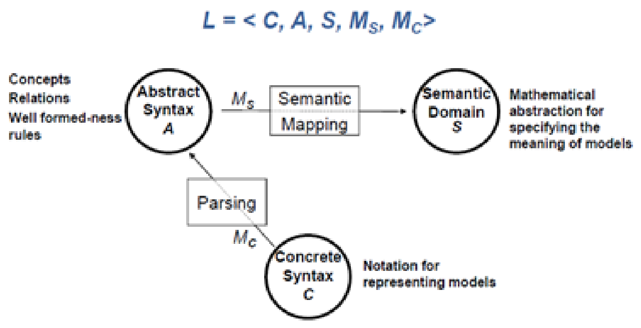


Figure II.8: Relationship between semantics, synatx, and their respective mappings

Design languages are specified via metamodels through syntax and semantics. Within the metamodel the descriptions of all entities and interconnections, as well as the relationships between entities describe the syntax of the instantiated language. More specifically this description refers to the abstract syntax of a language: the complete set of concepts provided by the language as well as its well-formedness rules [11]. The concrete syntax is the notation for representing the models, for example the graphical, table notation of UML. The concrete syntax maps the graphical notation to the underlying abstract syntax, so parsing the concrete syntax transforms the model into abstract syntax as seen in Figure II.8. If we consider the SignalFlow language, we see that its syntax specifies Primitives as atomic entities based in C code. The language supports Subsystems as blocks which contain primitives, ports and interconnections. The language also permits data flows between ports. Contained in this syntax are no details about actual operation or function of the model.

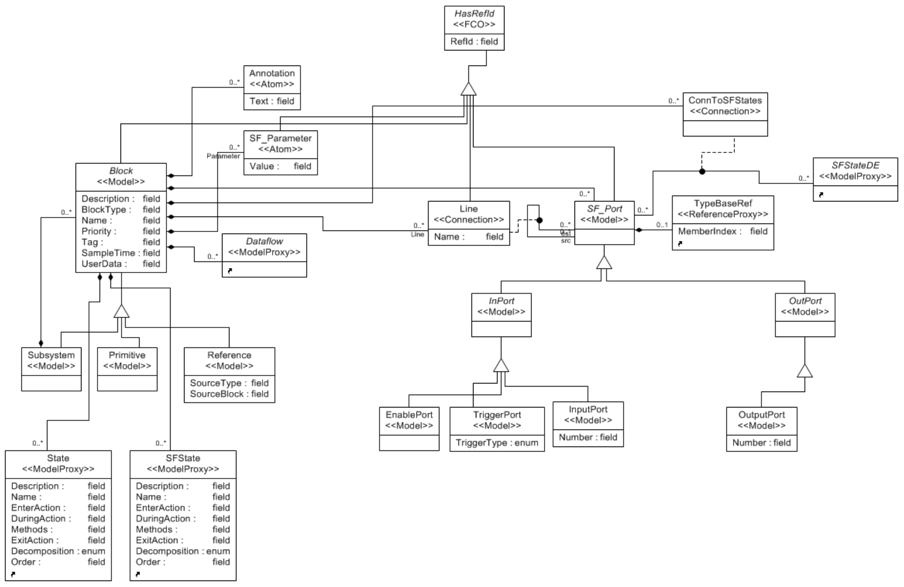

HasRefId <<FCO>> RefId : field
Annotation <<Atom>> Text : field
ConnToSFStates <<Connection>>
SF_Parameter <<Atom>> Value : field
Parameter
Block <<Model>> Description : field BlockType : field Name : field Priority : field Tag : field SampleTime : field UserData : field
SFStateDE <<ModelProxy>>
Line <<Connection>> Name : field
Line
SF_Port <<Model>>
dst
src
TypeBaseRef <<ReferenceProxy>> MemberIndex : field
Dataflow <<ModelProxy>>
InPort <<Model>>
OutPort <<Model>>
Subsystem <<Model>>
Primitive <<Model>>
Reference <<Model>> SourceType : field SourceBlock : field
EnablePort <<Model>>
TriggerPort <<Model>> TriggerType : enum
InputPort <<Model>> Number : field
OutputPort <<Model>> Number : field
State <<ModelProxy>> Description : field Name : field EnterAction : field DuringAction : field Methods : field ExitAction : field Decomposition : enum Order : field
SFState <<ModelProxy>> Description : field Name : field EnterAction : field DuringAction : field Methods : field ExitAction : field Decomposition : enum Order : field
0..*
0..1

Figure II.9: Relationship between semantics, syntax, and their respective mappings

A metalanguage's semantics describe the implementation of the design. As seen in Figure II.8, the syntactical model descriptions become translated into semantic domain through an automatic process called semantic mapping. The semantic domain describes the eventual meaning of a model's composed formalisms. Firstly, the structural semantics describe the meaning of various model instances in terms of their structure. Structural semantics map these model instances to a two-valued domain, distinguishing well-formed models from ill-formed models [38]. Behavioral semantics is quite different in that it maps the metamodel into a mathematical domain capable of capturing precise aspects of behavior, such as dynamics. [39] The behavioral semantics enforce the model's structure and syntax to create a resulting subset of model behaviors. A model's behavioral semantics are determined by first considering the entirety of actions permissible by the individual semantic units in the model. This set of actions is pruned according to restrictions created in the structural composition of the model. The final subset of permissible actions is realized as the behavioral semantics. Aptly named, the behavioral semantics will govern how the particular model will behave.

Figure II.9 illusrates a metamodel of SignalFlow data flow. The metamodel is depicted as a UML-style class diagram. If we examine the Block object, we see that it has many incoming connections from other objects in the digram. If the connection terminates with a dot on the Block endpoint, then it means that the opposing object can be aggregated within a Block within the model [3]. Since the SF_Parameter and Line both are similarly connected to the Block object, both of these objects can be contained side-by-side within a Block object. Subsystems, Primitives, and References are modular types of Block as specified by the inheritance operator (triangle) connecting these objects: they can aggregate, and be connected to all of the same objects that the Block object can [1]. From the containment connection from Block to Subsystem, we gather that subsystems can contain Blocks, such as other Subsystems. In this same way, InPorts and OutPorts are both modular types of SF_Ports. SF_Port has a line connected back to SF_Port that represnts a connection. Unlike a compositional relationship, a connection means that the two endpoints can be connected within a Signal Flow model. In this case, the connection means that SF_Ports can be linked together through a connection called a Line. The Line object, like the SF_Port object, can be aggregated within a block. From this diagram, we see how blocks can be composed in a hierarchy within other Blocks. The ports of block objects allow the designer to specify connections between independent blocks. The implementation of this metamodel as the Signal Flow metalanguage is covered in the *Signal Flow DSML* chapter.

As an advantage to the metaprogrammer, this diagram oriented representation provides an avenue for extensions and additions to metamodeling as may be needed for future designs. Furthermore, this table ordered representation has wide acceptance in the metamodeling community [3]. Because of this the UML model can be easily translated across metamodeling environments. While not immediately apparent the interoperability of a metamodel is crucial, since existing controller, software, and hardware design domains are highly specialized and often conceptually incompatible. It is a distinct advantage to be able to exchange the metamodel across tool suites. The interoperability of a metamodel/model is only feasible if the language is precisely structured. This means that syntax and semantics of the metalanguage must be clearly defined, even if the presentation and implementation are not.

II.3.2 GME

In order to employ a custom metamodel like the one specified in the Signal Flow class diagram, we require metaprogrammable tools such as modeling environments and translators which are capable of representing and enforcing the model's syntax. The metaprogrammable modeling environment embraced by ESMoL is the Generic Modeling Environment (GME) [24]. GME is a direct-manipulation editor permitting visual manipulation of data structures which underly the modeled system. GME is a metaprogrammable in the sense that metamodels can precisely determine 1) the storage interface, so that alternate database schemas can be employed, 2) the editor core, which permits only legal editing operations during composition, and 3) the graphical user interface which provides a presentation of the underlying syntax [1]. For the purpose of reusability GME's components are generic; however, during operation they reference the governing metamodel in order to verify legal composition in an interpretive sense.

The implicit toolset of the GME editor is the MultiGraph Architecture (MGA), a highly reusable framework that is catered towards customized design environments that can embody multiple different design approaches. For example, the MGA was used to develop the software for RDS, and DTool: these systems were used by Boeing in the development of the International Space Station Alpha for the purpose of evaluating the detectability and predictability of faults [8]. In particular, the RDS software code was generated automatically from the MGA model. Furthermore, the MGA is the implicit framework for the Computer Aided Dynamic Data Monitoring System (CADDMAS) created by the USAF Arnold Engineering and Development Center. This system provides real-time vibration analysis for an enormous range of frequency channels by utilizing a network of nearly 100 processors. The MGA was utilized to support the hierarchical modeling of signal flows, hardware resources, and hardware limitations of this parallel processing system [8].

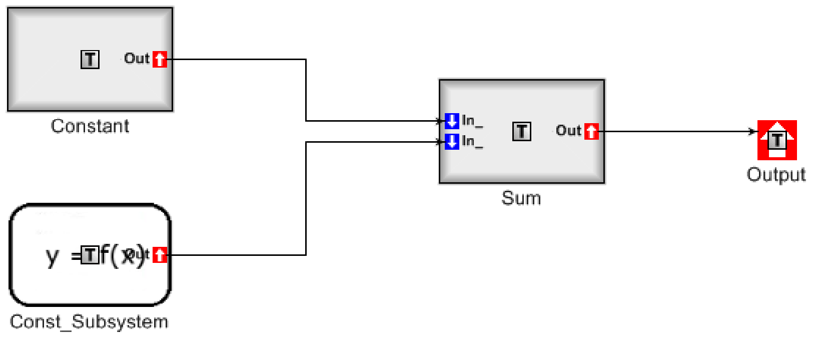


Figure II.10: A simple Signal Flow subsystem containing the Sum and Constant primitives along with an additional subsystem.

GME and MGA are so widely adaptable because of the genericity of its base-level concepts: Models, Connections, Atoms, Sets, References, and Attributes are the high-level modeling stereotypes which become mapped to specific metamodel elements through tagging [1]. These concepts allow the designer to create rich descriptions of physical and computational formalisms which can be catered towards numerous different design techniques.

Figure II.10 shows a GME implementation of the Signal Flow metalanguage from within a subsystem. In Signal Flow, primitives are represented as grey blocks with exposed ports. The Constant block possesses one output port which persistently relays its parameter value as a signal to the Sum block. This directional line implies a one-way connection to the input of the Sum block. The behavior of these atmoic primitives is resolved by referencing the C-code specified as an attribute of each Primitive. As the name implies, the Sum block computes the mathematical sum of the signals on its input ports. The ouput of Sum leads to an Output port, which other high-level components can use to extract the signal from this Signal Flow subsystem. We see that the second input to the Sum block comes not from a primitive, but from another subsystem. From our examination of the Signal Flow metamodel, we know that subsystems are capable of being composed within other subsystems. Within the Const_Subsystem, additional Signal Flow primitives are aggregated to produce a desired output value. The inclusion of subsystems aids in the process of partitioning system function, and imposes no change on the system behavior.

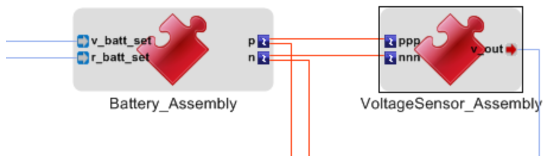


Figure II.11: The DC/DC converter battery as a component assembly within GME.

We now turn to an implementation of CyPhyML within GME (Figure II.11) which is comprised of two components which happen to be contain underlying references to Modelica blocks. At the CyPhyML integration level of hierarchy, the underlying models—whether they be Signal Flow or Modelica—have minimal effect on model composition, as modular components are treated in the same way by CyPhyML. The underlying models do, however, provide the output ports with which we can use to connect these components. The component on the left of Figure II.11 represents the battery and contains input signal ports allowing the real-time configuration of the battery's voltage and internal resistance. The battery circuitry is an electrical component, therefore, it contains two electrical power ports providing connections to the positive and negative terminals of the battery. The support of mixed type interfaces is one of the advantages of employing CyPhyML. Bidirectional connections to the power ports flow down to connect to the DC/DC converter, but two connections also flow to the right into the power ports of the voltage sensor. The voltage sensor functions by producing the voltage difference between its positive and negative terminals as a signal on its output. It effectively converts voltage across two electrical pins into a unitless signal which will ultimately be fed to the PWM for feedback purposes.

From these examples, we see that GME blocks or containers can be composed of completely different metalangauges. The analyzed metalanguages similarly contain Block type objects: Signal Flow possesses primitives and subsystems, while CyPhyML utilizes components and component assemblies. Both languages emlpoy connections although in distinct fashion: Signal Flow connections are causal and one-directional owing the fact that Signal Flow contains only functional blocks, while CyPhyML can contain these causal connections but does so in conjunction with bidirectional power ports to support physical system modeling. The modularity of GME, which allows it manipulate and unify these disparate metalanguages, will be crucial in modeling our DC/DC converter design. What GME provides as a graphical modeling tool is an environment which visually represents custom metalanguages as well as the relationships between their components for creation and editing [8].

II.3.3 The Signal Flow DSML

Signal Flow is a domain specific modeling language that was created at Vanderbilt University from the ESMoL framework. The langauge was further developed during our work with the DC/DC converter, and was vital in being able to model and simulate the vehicle controller. The Signal Flow language is specifically designed for representing software components (like DSP algorithms or user interface components) in a synchronous discrete-time fashion [1]. Signal Flow blocks are fired at discrete, periodic intervals. The abstract semantics which govern Signal Flow are as follows:

1. Signal Flow models consist of causal functional blocks that use ports to express a chain of computational processes.

2. Each functional block has strongly typed input and output blocks which are used for acquiring data and pushing output data downstream.

3. Functional blocks can either be references, subsystems, or primitives. While references are pointers to existing subsystems and primitives, subsystems are blocks that contain either primitives or other subsystems and bare their own input and output ports. Primitive blocks are the atomic actors of Signal Flow and reference C-code algorithms for their behavioral semantics.

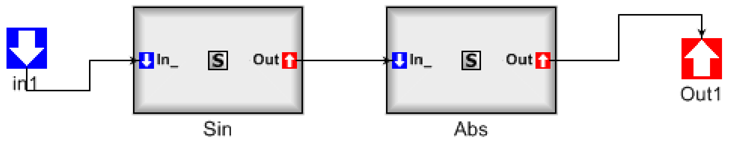


Figure II.12: A simple Signal Flow were the input flows into a Sin block, and then into an Absolute value block.

Figure II.12 shows a diagram of a simple Signal Flow subsystem containing two primitive blocks. When the Sin block and Abs block are used together in tandem, the resulting function on input resembles,

$$y = |sin(x)| \tag{II.1}$$

This describes how the output y will behave given an input signal x. To observe the simulated effects of our Signal Flow function, we compose the model within a Component assembly, and a Testbench before exporting the simulation. A ramp input is used to vary x throghout the simulation window. A resulting Modelica simulation and its accompanying C-code are then generated once the MGA2Modelica process is invoked. When the simulation is run in Modelica, the traces of the input and output are produced as seen in Figure II.13.

From Figure II.13, we see that the simulation correctly produces output that corresponds to Eq. (II.1) given the ramped input. This successful transformation is simulated by executing the C algorithms corresponding to the Signal Flow blocks as they are periodically called during run time.

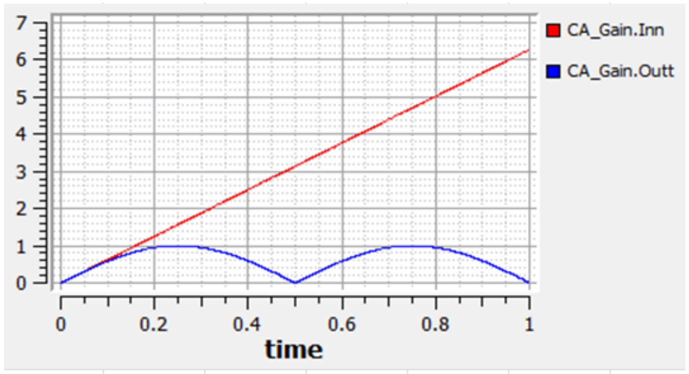
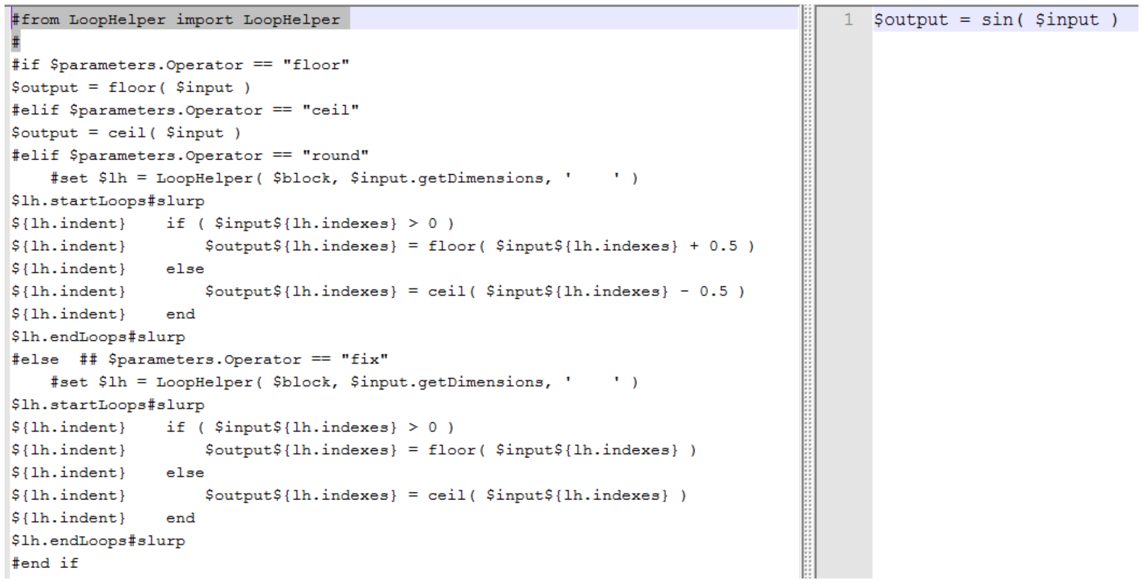


Figure II.13: A simulation of the Sin-Abs Signal Flow system. The red input *x* is pictured as a line, while the blue output *y* is appears as the positive magnitude of a sinewave.

When exporting a simulation, the interpretter will generate a C-code snipet for each of the Signal Flow blocks by referencing the Cheetah template specified in the primitives' Block Type attribute. The Cheetah template for the Sine, and Abs blocks is shown in Figure II.14.



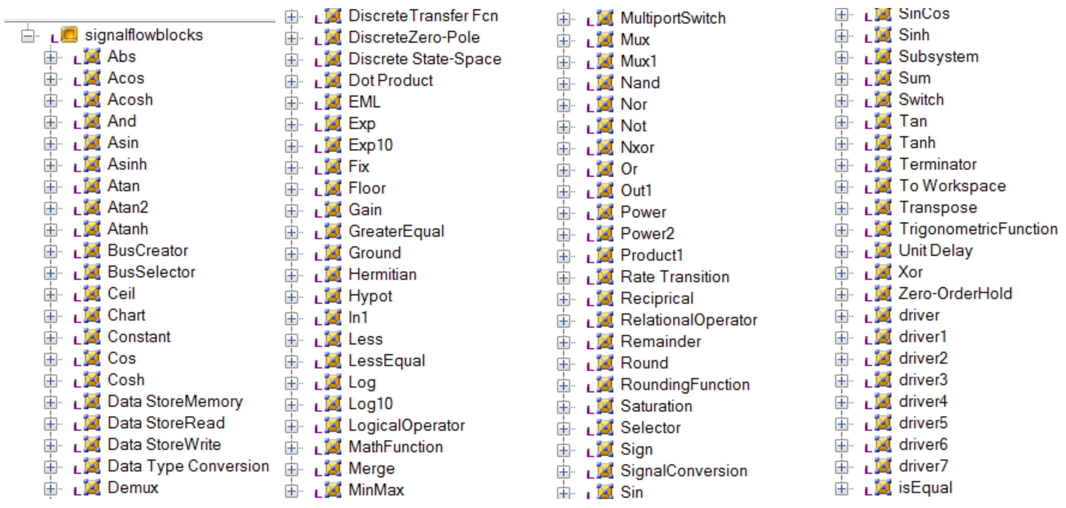
```
#from LoopHelper import LoopHelper
#
#if $parameters.Operator == "floor"
$output = floor( $input )
#elif $parameters.Operator == "ceil"
$output = ceil( $input )
#elif $parameters.Operator == "round"
    #set $lh = LoopHelper( $block, $input.getDimensions, '    ' )
$lh.startLoops#slurp
${lh.indent}    if ( $input${lh.indexes} > 0 )
${lh.indent}        $output${lh.indexes} = floor( $input${lh.indexes} + 0.5 )
${lh.indent}    else
${lh.indent}        $output${lh.indexes} = ceil( $input${lh.indexes} - 0.5 )
${lh.indent}    end
$lh.endLoops#slurp
#else  ## $parameters.Operator == "fix"
    #set $lh = LoopHelper( $block, $input.getDimensions, '    ' )
$lh.startLoops#slurp
${lh.indent}    if ( $input${lh.indexes} > 0 )
${lh.indent}        $output${lh.indexes} = floor( $input${lh.indexes} )
${lh.indent}    else
${lh.indent}        $output${lh.indexes} = ceil( $input${lh.indexes} )
${lh.indent}    end
$lh.endLoops#slurp
#end if
```

```
$output = sin( $input )
```

Figure II.14: Cheetah templates corresponding to the Rounding Signal Flow block on the left, and to the Sin Signal Flow block on the right.

The Cheetah template on the right for the Sin block is very simple. It clearly dictates that the output of the block shall be equal to the sine of the input of the block. The sine function is included in standard math libraries and will be utilized for employing the algorithm. This Cheetah snipet is not C-code, but is a template for generating a lengthier and more formal C-code algorithm.

If we turn attention to the left side of Figure II.14 we can see some additional syntax that empowers Cheetah templates compared to a simple C template. The Rounding block contains numerous lines preceeded by the pound sign which represent python function calls. These interspersed python calls allow the final C-code algorithm to be configured on the fly, by incorporating calculations and branching statements within the Cheetah template. The dynamic calls to python are necessary since the Rounding block's function is dependent on its Operator parameter: If the Operator is equal to "floor", then floor rounding will be performed. If the Operator is equal to "ceil", the the input will be rounded to the ceiling integer. The Python code allows decision making processes to occur during the transcription of the Cheetah template onto the C-code algorithm. So if the Operator is changed between simulations, the Rounding block will still be able to perform the correct action and generate unique C-code for the new simulation. Additionally, this functionality comes in handy when there are multiplexed inputs to blocks, or blocks whose function is affected by the sample rate.



```
signalflowblocks
  Abs                   DiscreteTransfer Fcn   MultiportSwitch       SinCos
  Acos                  DiscreteZero-Pole      Mux                   Sinh
  Acosh                 Discrete State-Space   Mux1                  Subsystem
  And                   Dot Product            Nand                  Sum
  Asin                  EML                    Nor                   Switch
  Asinh                 Exp                    Not                   Tan
  Atan                  Exp10                  Nxor                  Tanh
  Atan2                 Fix                    Or                    Terminator
  Atanh                 Floor                  Out1                  To Workspace
  BusCreator            Gain                   Power                 Transpose
  BusSelector           GreaterEqual           Power2                TrigonometricFunction
  Ceil                  Ground                 Product1              Unit Delay
  Chart                 Hermitian              Rate Transition       Xor
  Constant              Hypot                  Reciprical            Zero-OrderHold
  Cos                   In1                    RelationalOperator    driver
  Cosh                  Less                   Remainder             driver1
  Data StoreMemory      LessEqual              Round                 driver2
  Data StoreRead        Log                    RoundingFunction      driver3
  Data StoreWrite       Log10                  Saturation            driver4
  Data Type Conversion  LogicalOperator        Selector              driver5
  Demux                 MathFunction           Sign                  driver6
                        Merge                  SignalConversion      driver7
                        MinMax                 Sin                   isEqual
```

Figure II.15: The library of currently supported Signal Flow blocks.

What we have just described is technique that Signal Flow employs for synthesizing program code from its atomic functional blocks. Each block literally translates to a C-code software function. The plurality of the Signal Flow block library determines the scope of designs that can be created with Signal Flow, and we have worked to expand the Signal Flow block library during this work. A picture representing all currently supported Signal Flow functions is pictured in Figure II.15. This list of Signal Flow blocks appears in GME once the user instantiates a Signal Flow component. The user can then copy and paste these existing functions into the Signal Flow container.

In the same way that these fundamental Signal Flow blocks were created, any user can create their own specialized Signal Flow blocks by 1) Creating the block with desired parameters and interfaces within the SignalFlowBlocks library file, and 2) writing a corresponding Cheetah template that wil govern the operational behavior of the block. Many of the current Signal Flow blocks were created to mimic the behavior of common Simulink functions, as Signal Flow semantics are the target language of imported Simulink designs within ESMoL. Signal Flow behaves very similarly to the discrete-time operation of Simulink control loops. In accordance with ESMoL's approach of supporting free and open software, we have developed Signal Flow's functional blocks to correspond to pre-existing Modelica functions, as opposed to Simulink functions. Although this makes the import transformation more complicated, it is feasible for the same computational process to behave similarly despite differing software functions.

The Signal Flow metalanguage is crucial in modeling our DC/DC converter because it allows for the modeling of computational functions within CyPhyML. Since CyPhyML is already capable of describing the physical dynamics of Modelica models, this addition of Signal Flow to the langauge allows both physical and computational components to be composed together. In a way not previously achievable through state-of-the-art modeling suites, the Signal Flow DSML allows multi-domain models to composed, manipulated, and simulated together within a single tool suite within a single system. In the following sections, we will evaluate the cooperation of the Signal Flow-based vehicle controller, and the Modelica-based circuit model in our DC/DC converter system.

CHAPTER III

Bidirectional DC/DC Converter

III.1 Converter Overview

III.1.1 Converter Purpose

To motivate our description of the facets of ESMoL, we focus on an actual control design model for the bidirectional DC/DC converter. Figure III.1 shows the vehicle architecture for a series hybrid-electric vehicle which employs a bidirectional DC/DC converter.

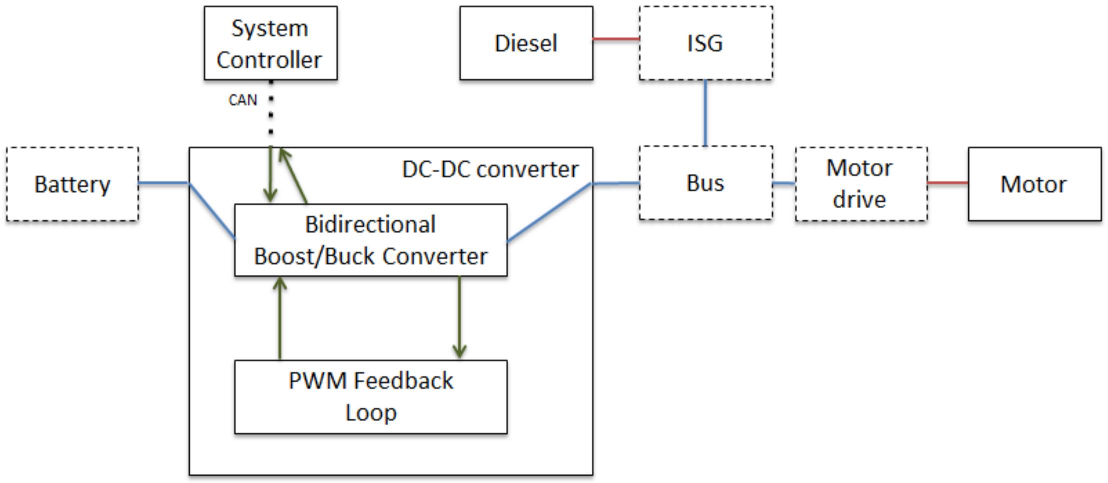


Figure III.1: Series Hybrid-Electric Vehicle Architecture

This hybrid vehicle configuration is equipped with an electric motor for driving the wheels instead of an Internal Combustion Engine, as with a typical automobile. The series hybrid still contains an onboard Internal Combustion Engine which burns diesel fuel, yet the engine's crankshaft does not directly provide torque to the wheels [42]. Instead the engine's mechanical energy is immediately converted into electrical energy, which in turn powers the electric motor among other functions. Encompassing the function of the alternator, an Integrated Starter/Generator (ISG) is fixed to the Internal Combustion Engine's crankshaft and by means of a rotating magnetic field transforms rotational momentum into induced current [40]. The ISG also combines the utility of a starter motor, and allows precise starting and stopping of the engine since the Internal Combustion Motor does not always have to be running while driving. With this drivetrain configuration power to the wheels can either be provided by the Internal Combustion Engine, the battery, or by the combination of the two in a process called "hybrid operation".

The bus is directly connected to the motor, so providing voltage on the bus effectively delivers power to the wheels. From the diagram we see that the battery is also connected to the electric bus, but through the DC/DC converter. The bidirectional DC/DC converter performs the crucial task of regulating voltage between the battery and electric bus. Voltage regulation is required because the 300V battery configuration needs to be boosted up to about 600V in order to drive the electric motor. The reason that the motor's operating voltage is so high is because it needs to deliver enormous amounts of torque to propel the car forward. The motor's power output is proportional to current times voltage, expressed as

$$P = I * V \tag{III.1}$$

Alternatively, by substituting out the voltage variable power can be represented as the current squared divided by the load resistance:

$$P = I^2 * R \tag{III.2}$$

In order to achieve greater motor power, one needs to either increase current to the motor, or voltage across the motor's terminals. Hybrid vehicle designers have chosen motors that operate with high voltages and low currents in order to minimize copper loss. The reduction of efficiency from energy lost in the wiring is a relative to equation III.2, and as a result these losses are called "I squared R Losses". With this knowledge, we are led to the conclusion that it is more efficient to boost voltage from the battery to drive the motor, than to deliver the same amount of power at a lower voltage with higher current.

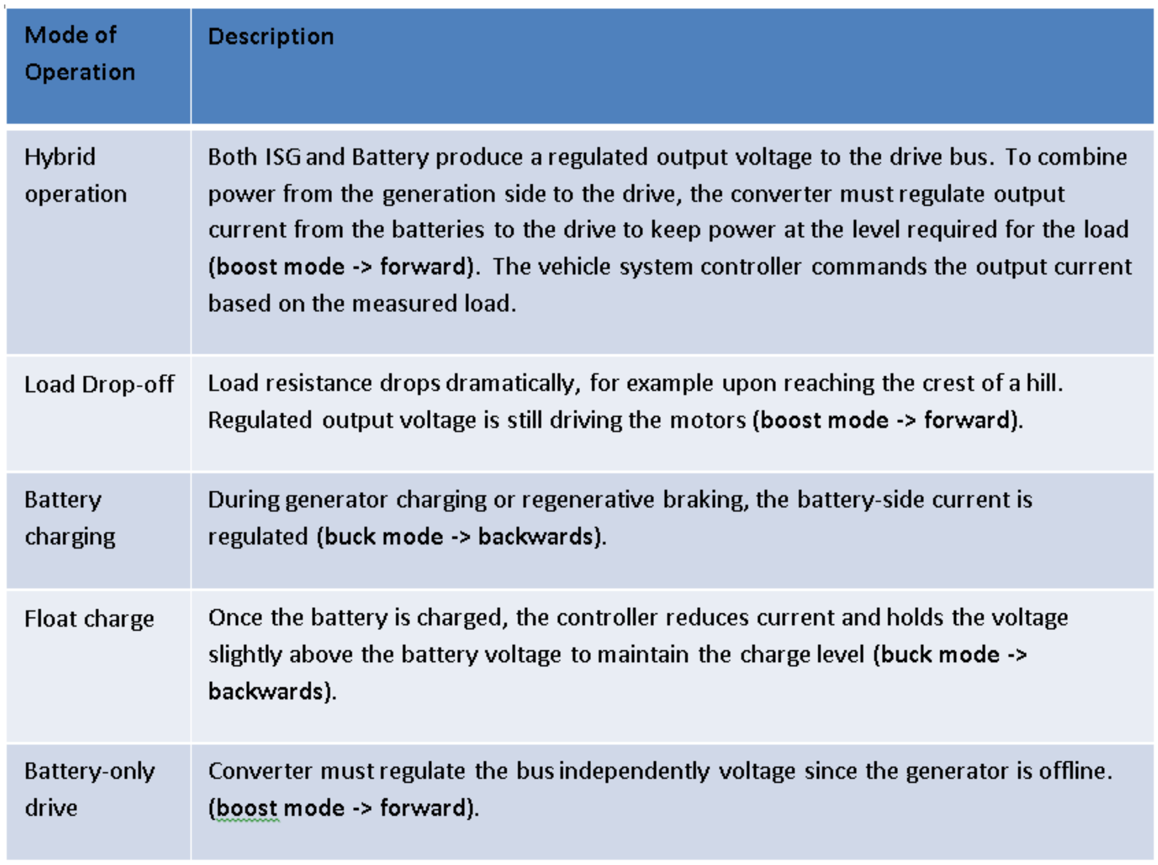
In addition to boosting battery voltage to drive the motor, another one of the bidirectional DC/DC converter's main functions is to charge the battery during regenerative braking by regulating current in the opposite direction. Again, to reduce I squared R losses the motor generates a high voltage on the electric bus during braking events. The DC/DC converter is able to buck this large voltage down to an appropriate level for the batteries, so as to not overcharge them. The phenomenon of regenerative braking is possible because electrical motors are functionally reversible: motors become generators when forced to rotate in the absence of applied electrical power [41]. Much like a dedicated generator, a hybrid vehicle's motor converts rotational momentum into induced current flow when in the presence of a magnetic field. The act of converting rotational momentum from the wheels slows the rotation of the wheels since energy is preserved. Therefore, the car generates usable electrical energy while simultaneously slowing the vehicle during regenerative braking.

When the vehicle transitions from electric drive to regenerative braking, the circuitry alone is not able to re-rout current automatically. Because of this the bidirectional DC/DC converter requires a software controller than can flip the circuit between buck mode and boost mode as appropriate. This controller is also necessary for protecting the electronics within the vehicle drive-train, especially the battery. To delineate, the Li-Ion batteries can only tolerate certain operating conditions: the charge current cannot be too high, a permanent fuse can be blown if the charging voltage is too high, the battery can fail if overcharged beyond capacity, and the battery can fail if discharged too much [37]. The DC/DC converter is the only barrier between the battery and dangerous electrical energy flowing along the bus. The converter's controller is single-handedly in charge of transitioning between operating modes. Therefore, we require high confidence in our controller software design before deployment, otherwise we risk damaging the electrical components of the drive-train.

In order to capture the DC/DC converter's behavior properly during simulation, we require a tool suite that allows us to simultaneously compose computational elements with physical elements, or discrete-event MoC's with continuous-time MoC's. The CyPhyML language is able to perform this by integrating dynamic components with the Signal Flow DSML. Dataflow programming for the controller can take place within Simulink before being exported to CyPhyML. Then as the controller switches between operating modes, we can observe the evolving converter behavior and determine stability, response time, or other formal properties for the entire CPS system.

III.1.2 How the Converter Works

In terms of the operation of the DC/DC Converter device, there are five distinct modes of operation that will arise as the result of typical driving. These five different modes arise because of of varying battery charge, motorist adjustments, and environmental factors imposed by the terrain. These cases are outlined in the table of Figure III.2.



Mode of Operation	Description
Hybrid operation	Both ISG and Battery produce a regulated output voltage to the drive bus. To combine power from the generation side to the drive, the converter must regulate output current from the batteries to the drive to keep power at the level required for the load (**boost mode -> forward**). The vehicle system controller commands the output current based on the measured load.
Load Drop-off	Load resistance drops dramatically, for example upon reaching the crest of a hill. Regulated output voltage is still driving the motors (**boost mode -> forward**).
Battery charging	During generator charging or regenerative braking, the battery-side current is regulated (**buck mode -> backwards**).
Float charge	Once the battery is charged, the controller reduces current and holds the voltage slightly above the battery voltage to maintain the charge level (**buck mode -> backwards**).
Battery-only drive	Converter must regulate the bus independently voltage since the generator is offline. (**boost mode -> forward**).

Figure III.2: The 5 Use Cases of Bidirectional DC/DC Converter Operation

The first case, called hybrid operation, combines current from both the battery and ISG so that maximum power can be delivered to the motor's bus. The direction of current flow for this configuration is illustrated in Figure III.3 . We observe that the battery is providing power to the bus, and its voltage must be boosted by means of the DC/DC Converter. Simultaneously the ISG generates electrical energy by combusting gasoline, and then delivers the resulting current to the motor's bus. The actual voltage level across the motor's terminals is set by the accelerator, which is realized in the converter circuit by adjusting the Pulse Width Modulator. There is not one set voltage, but instead a range of permissible voltages corresponding to a range of motor torques: The lowest acceptable voltage is about 580V corresponding to minimal torque, and the highest voltage at about 720V corresponds to maximum torque delivered to the wheels. ISG and DC/DC Converter must provide similar voltage so that current travels forward through the motor coils without damaging the battery.

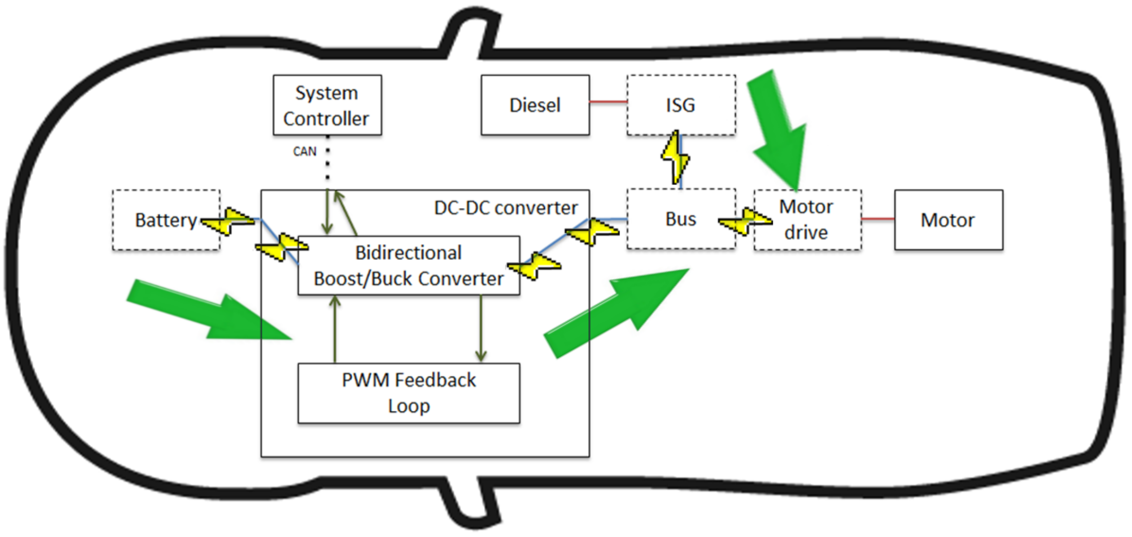


Figure III.3: Power Flow during Hybrid operation and Load drop-off

Again, hybrid operation is the most powerful operating mode so the voltage will approach the upper limit during this use case. The vehicle operator does not manually switch on the ISG and Battery to achieve this operating mode, and instead the vehicle system controller automatically engages the Internal Combustion Engine. Before a transition to hybrid operation, the vehicle system controller will sense that a large torque is requested by the accelerator and that the Battery alone is not able to produce enough electrical power. The ISG will then automatically engage the Internal Combustion Engine, and begin powering the electric motor in tandem with the DC/DC Converter. Hybrid operation would likely engage during initial acceleration from rest, or as the vehicle climbs a steep incline.

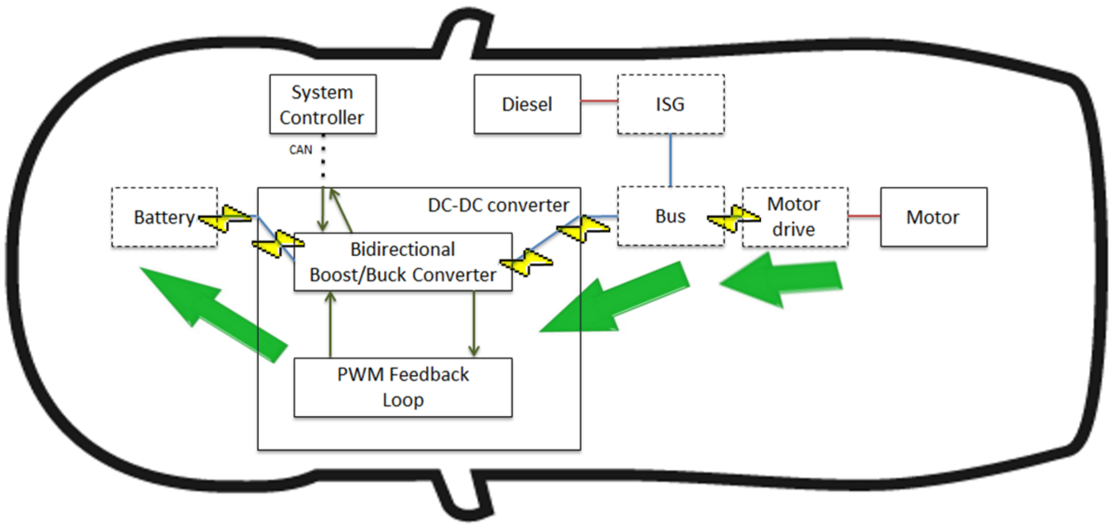


Figure III.4: Regenerative Braking Power Flow during Battery Charging and Charge Floating

A second mode of operation called Load Drop-off is similar to hybrid operation in that current also travels from the Battery and ISG to operate the electric motor. We refer again to Figure III.3 for an accurate depiction of power flow within the vehicle's circuitry. In order for the vehicle to transition to Load Drop-off a change must occur in the environment such that the resistance of the motor drops drastically, such as reaching the crest of a hill. With the same power applied to the motor, the vehicle will travel much faster downhill than uphill so the DC/DC Converter needs to maintain a constant voltage despite changing load conditions. In this mode, the PWM in conjunction with the vehicle controller must maintain control for the driver and protect the engine by preventing abrupt acceleration of the motor. The ISG may become disengaged during this mode because the extra energy from combustion may no longer be needed to propel the vehicle.

This next mode to be introduced employs the Bidirectional DC/DC Converter operating in the reverse direction. When in Battery Charging mode power on the electric bus becomes bucked through the DC/DC Converter, and stored in the battery. There are two possible scenarios that bring about Battery Charging mode, and in each case the battery must not yet be fully charged. In the first scenario the driver has applied the brake pedal in order to slow the vehicle. If reduced or trivial power is applied to the electric motor the motor will not produce mechanical torque; however, the motor's shaft will remain in rotation due to the forward translational momentum of the vehicle. Instead of coasting like an ideal wheel and axle configuration, the electric motor will slow the vehicle by converting the rotational momentum of its rotors into induced current flow [41]. The motor will generate electric power proportional to the force required to slow the vehicle, recovering the energy normally lost as heat during frictional dynamic braking. This abundance of electrical energy is applied towards charging the battery as seen in Figure III.4. The voltage output from the motor during regenerative braking is typically very high, so the DC/DC Converter needs to buck this voltage down to a level more appropriate for the battery: about 300V for charging.

The second scenario of Battery Charging occurs when the vehicle is not accelerating or braking but instead cruising, and energy produced from diesel combustion is able to fully power the driving load. In such a case, excess electrical energy generated from the ISG is routed through the Bidirectional Converter so as to recharge the battery as seen in Figure III.5. This charging phenomenon is possible because the Internal Combustion Engine does not directly turn the wheels, but instead generates electrical energy via the ISG: In addition to powering the electric motor, the ISG is fully capable of using its power output to recharge the battery, thereby converting mechanical energy into electrical energy into chemical energy within the battery. In each of these scenarios, the Bidirectional DC/DC

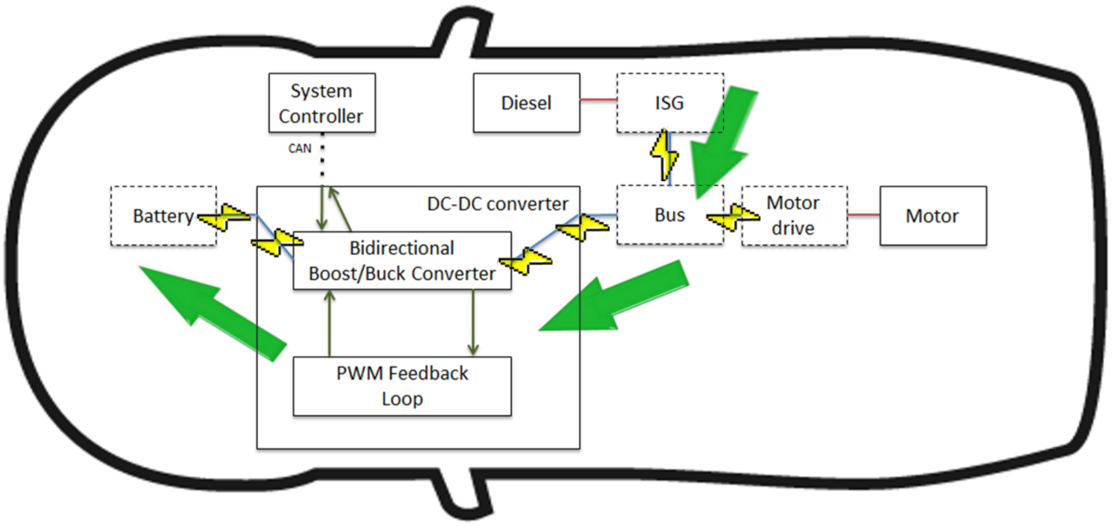


Figure III.5: Diesel Generation Power Flow during Battery Charging

Converter operates similarly: whether the power is supplied by the electric motor or Internal Combustion Engine, the converter must buck the bus voltage down to a safe level for charging the battery.

A fourth mode of operation called Float Charge—also known as Trickle Charge—is similar to Battery Charging mode. The main difference between the two is that during Float Charge the battery is already charged to capacity, therefore, the DC/DC Converter needs to regulate a lower-than-normal voltage across the battery's terminals. The motivation for this is to allow only a minute amount of charge current to trickle into the battery cell. At this Float Charge current—which is only a fraction of the normal charging current and is equal to the self-discharging rate—the battery is able to remain in the fully charged state. The battery can continuously maintain a Float Charge for long periods of time without permanently affecting the integrity of the battery [43]. The battery cannot operate at the normal charging current for indefinite periods of time, as Lithium Ion batteries are not tolerant of overcharging and would eventually fail [37]. The DC/DC Converter's software controller needs to be able to detect when the battery has been charged to capacity in order to govern whether to regulate high voltage and high current, or low voltage and the Float Charge current at the battery's terminals.

Like with Battery Charging mode, Float Charging occurs during braking events as visualized in Figure III.4. While engaged with the wheels during braking, the electric motor generates an abundance of electrical power on the system bus. Even if the battery remains fully charged it is advantageous to employ this extra energy towards maintaining the battery at full capacity, as battery cells will eventually self-discharge over time. In the absence of extra power created from regenerative braking, it would not appropriate to engage the Internal Combustion Engine in order to maintain a Float Charge on the Battery as illustrated in Figure III.5. Since the Float Charge current is very low, a majority of the energy generated by the ISG would be wasted. The scenario in Figure 13 is only pertinent to the Battery Charging mode. However, even with regenerative braking, managing the abundance of generated power is problematic and for this reason ordinary dynamic brakes must work in conjunction with the regenerative electric generator.

Routinely, the braking load is distributed between both brake systems as needed, wherein the electric motor recuperates power and the dynamic brakes dissipate power as frictional heat [44]. During Float Charging, the vehicle may rely more on the dynamic brakes for power dissipation so as to not generate exorbitant electrical energy on the bus [45]. Additionally, if the critical temperature for safe charging has been reached then battery charging would be prohibited, and the vehicle would then automatically switch to the dynamic brakes regardless of the battery's charge. A final important note is that the effectiveness of regenerative braking diminishes at slow speeds, so it is mandatory for the dynamic brakes to engage in order to bring the vehicle to a complete stop [44].

The final mode of operation is Battery-Only Drive, and in this mode the car is propelled

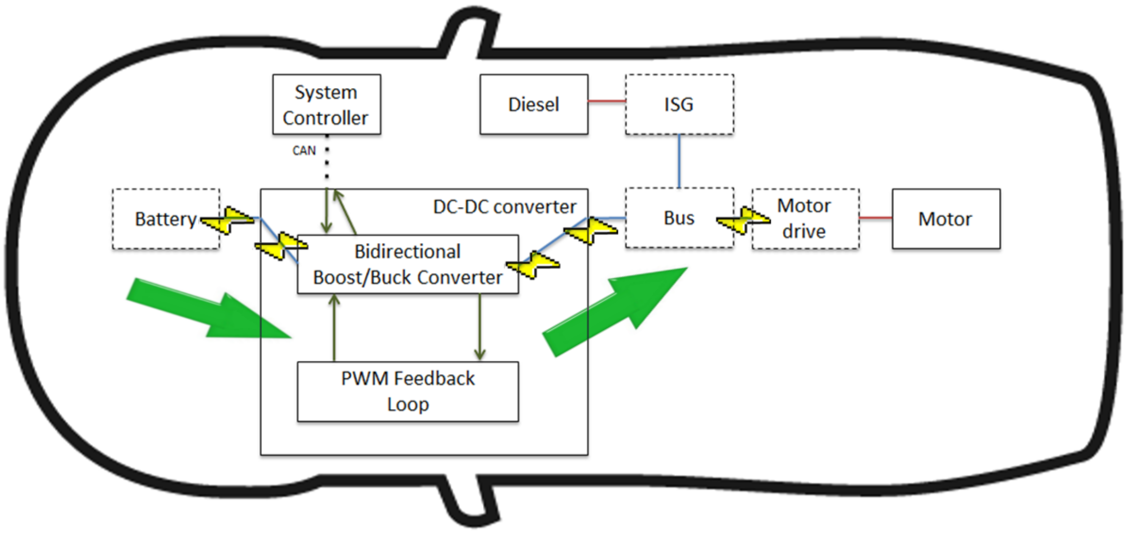


Figure III.6: Power flow for Battery-Only Drive

forward by relying only on the power stored in the battery. As seen in Figure 14, the ISG and Internal Combustion Engine are not used at all in this operating mode. Because of the lack of extra power generation in this mode, the voltage and motor torque remain near the lower end of the acceptable operating range. In order to power the motor, the 300V produced by the battery must be boosted up to at least 580V (citation needed). The Bidirectional DC/DC Converter must safely regulate the battery's power output so as to not permit dangerous levels of discharge current which could destroy the battery.

Battery-Only mode would be ideal for coasting on a highway because a relatively small amount of applied torque would be required to keep the vehicle at its current speed. If the operator were to apply a large acceleration or if the car were to take on an upwards slope, then the Internal Combustion Motor would become engaged to assist with the higher load demand. In many cases the battery alone is suited for powering the electric motor; a pure electric vehicle would always be in Battery-Only Drive when accelerating. Battery-Only Drive is emissionless and improves fuel efficiency by extending the number of miles a hybrid vehicle can travel per gallon.

III.1.3 Ideal Bidirectional DC/DC Converter Circuit

We will now explain how it is that the DC/DC converter is able to provide its boosting and bucking capabilities, as well as how it is able to do so in a bidirectional manner. We begin by referring to a realization of the DC/DC converter circuit that employs two ideal switches, switch *A* and switch *B*, pictured in Figure III.7. We see that the battery−consisting of a voltage source and an internal resistance−is situated on the left side of the circuit model and is coupled with a capacitor. The bus terminals, which connect to the electric motor, are similarly positioned on the right side containing resistive, capacitive, and sourcing elements. The remainder of the circuit consists of an inductor, *L*, and the two switches.

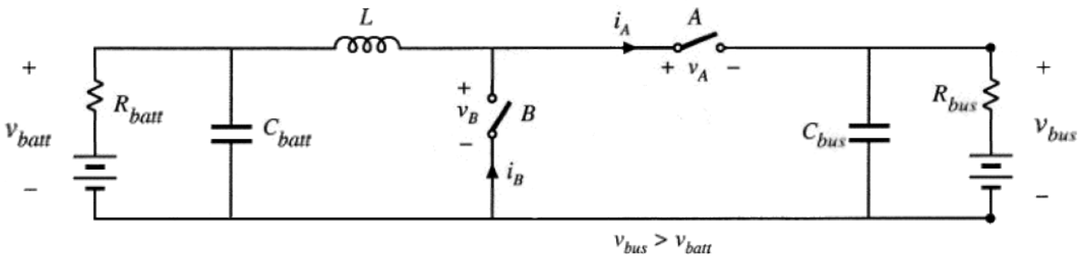
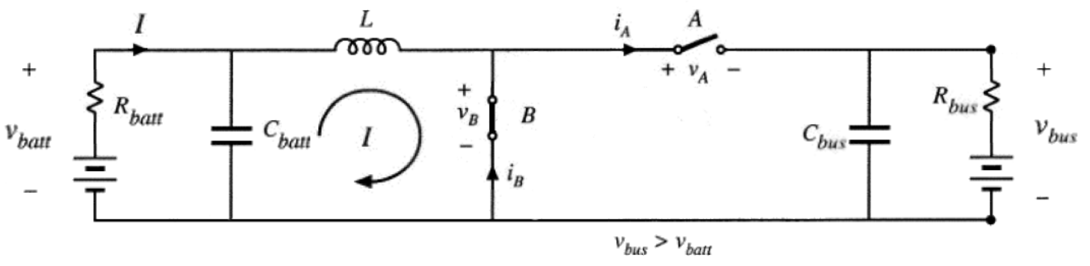


Figure III.7: Realization of the DC/DC converter with ideal switches

Boost Operation

Based on the operating conditions for the vehicle, the resistive and source elements will assume widely different values as during electric-drive, or regenerative braking. To start, let's assume that the vehicle is operating in electric-drive mode, and that the battery voltage needs to be boosted to nearly double its value in order to power the bus terminals. In the first boost mode, interval switch *B* is closed, while switch *A* is open as pictured in Figure III.8.1. Within this mode, the bus output becomes isolated from the battery network and any charge stored in C_{bus} will flow out through the bus terminals. More importantly, the isolated battery network will quickly build up current in the inductor during this interval due to the establishment of a low resistance path to ground. The large positive voltage across the inductor boosts the inductor current *I* for a short time. The boosted inductor current will be used to produce a controllably large voltage across the output load after the switching event, which would be otherwise unattainable in a static circuit configuration.

(1)



(2)

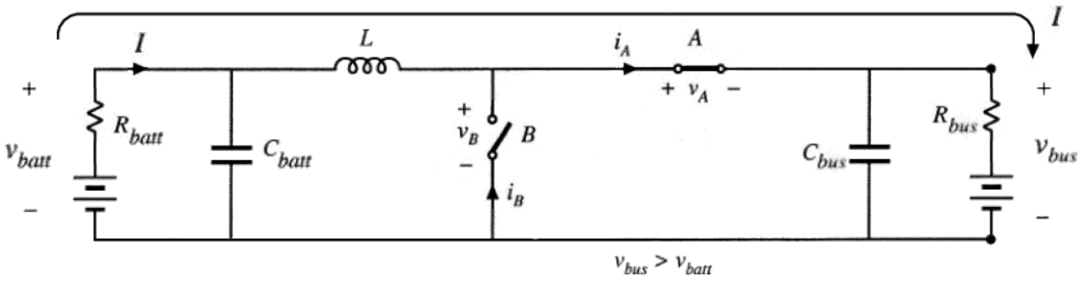


Figure III.8: Converter operation during boost mode: (1) Interval 1 where *A* is open, and current builds up in the inductor. (2) In Interval 2 *A* is closed and the inductor current drives the bus.

After a period of time specified by the product of the duty ratio and the sample period (DT_s), both switches will invert their position. *A* is closed in this second interval, while *B* is open, preventing the previous path to ground as shown in Figure III.8.2. It is important to realize that v_{bus} will always be greater than v_{batt}; therefore, the inductor now has a negative voltage across its terminals. Due to the inherent properties of an inductive coil, the inductor current cannot instantaneously switch directions to reflect the newly applied voltage. Instead the inductor current lessens gradually while maintaining positive current flow. So by means of these switching events, we are able to−for short a time−pass a positive current "uphill" against a negatively applied voltage.

Without the inductor, it would be impossible for v_{batt} to produce a voltage across v_{bus} greater than its own. The high voltage across the output of a boost converter is achieved by (1) effectively "short-circuiting" the battery through the inverter to amass inductor current in mode 1, and then (2) rapdily closing switches to direct the boosted inductor current through the output load.

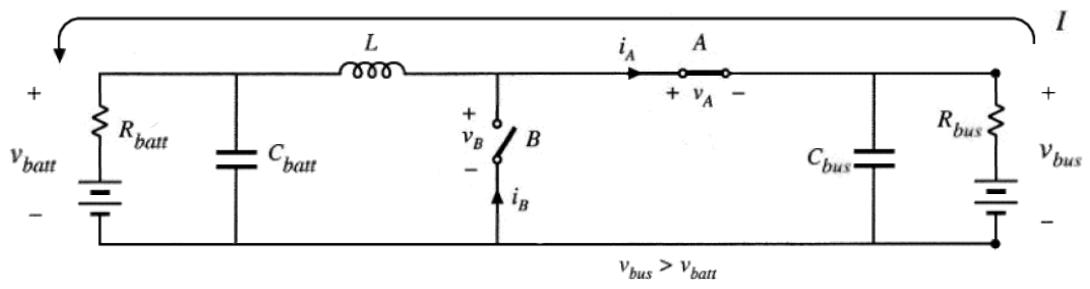
If the second interval switches before the inductor current reaches zero, then the inductor will always supply positive current and positive voltage across v_{bus}, despite the fact that the applied voltage across the inductor is negative. Even if the inductor current crosses zero amps, the output capacitor C_{bus} will smooth the voltage transistion at the bus output.

The amount by which the output voltage is boosted is determined by the relative proportion of time that the circuit is in each interval. If the circuit is mostly in interval 1, then it will produce a large voltage at the output. The effect is lessened as the proportion of time of the second interval is increased. It is common practice to refer to the fraction of the full cycle dedicated to the 1st interval as the duty ratio, D, and to the time duration of the full switching period as T_s. A pulse-width-modulated circuit will enforce the duty ratio, which can vary from cycle to cycle, by appropriately opening and closing the converter switches.

Buck Operation

A standalone buck converter is constructed using the very same elements used in a boost converter: capacitors, inductors, resistors, and switching elements. The bidirectional DC/DC converter is specifically constructed to allow the same elements used for boost conversion to be employed towards buck conversion in the reverse direction. The principle of bucking is similar to that of boosting, in that both conversion techniques are centered around manipulating current within the inductor. In buck mode, the inductor feeds directly into the battery output and is sometimes isolated from the source voltage, as opposed to boost mode where the inductor is always connected to the source voltage and only periodically drives the output. In effect, the inductor is able to maintain a current level that is less than what would be possible if the bus voltage source were to directly drive the output.

(1)



(2)

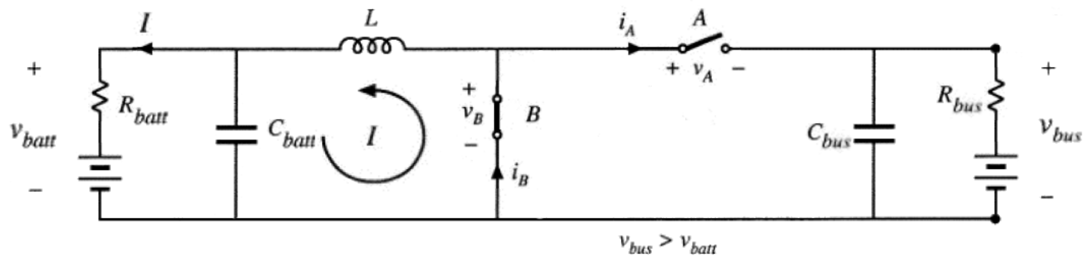


Figure III.9: Buck mode converter operation: (1) Within Interval 1, *A* is closed allowing current to build up in the inductor, and (2) in Interval 2, *A* is open and the inductor drives the battery while isolated from the source voltage.

Let's examine the first interval of buck operation as pictured in Figure III.9.1. We need to be cognizant that buck mode operates in the reverse direction, so the source voltage, v_{bus}, is now on the right side of the circuit model and the output voltage is on the left. We observe that current follows the outer loop in this switching interval. Switch *A* is closed, allowing v_{bus} to directly drive the battery load and build up current in the inductor. v_{batt} will never rise above v_{bus}, and the inductor current will never get large enough to cause v_{batt} to do so.

When the converter transitions into the second interval, as pictured in Figure III.9.2, the battery and inductor network become isolated from the high voltage v_{bus} source. Furthermore, switch *B* becomes closed in this mode and opens up a path to ground at the positive terminal of the inductor, thereby applying a negative voltage across the inductor. This prompts a gradual negative change in inductor current. The inductor still passes current to the output in this mode, though it is less than the peak current value experienced at the end of the first interval.

Buck mode operation achieves an output voltage lower than the source voltage by bucking—or blocking—current flow from the v_{bus} source for a portion of the switching cycle. The inductor is essentially in limbo between peak current and zero current as the converter progresses in time. By maintaining an inductor current that is less than the peak current, the converter is able to regulate a controllably low voltage at the battery terminals with maximum efficiency. Our circuit design could have employed a simple voltage divider in order to regulate v_{bus} down to half voltage, yet this technique would be highly inefficient due to large power losses in the divider's resistive network.

Since the maximum possible voltage at the output is v_{bus}, we need to determine how much below v_{bus} the output voltage will settle. Like with boost mode operation, the output voltage is determined by the duty ratio, D. With buck operation, a lower duty ratio corresponds to a lower votlage across the battery terminals. The PWM controls the duty ratio dynamically while maintaining a constant T_s. Even if the resistance across the output were to somehow change, the PWM would be able to use a feedback loop to maintain the regulated voltage by adjusting the duty ratio on the fly.

III.2 Bidirectional DC/DC Converter Equations

We will now evaluate the DC/DC converter circuit and generate circuit equations that govern the behavior of both boost and buck mode operation. Evaluating the circuit topology will allow us to (1) solve for the conversion ratio, ($M(D)$), (2) solve for the efficiency function, (η), and (3) determine how component values will govern the behavior of the converter. We will start by introducing the volt-second and amp-second balance principles, which are fundamental to power converter operation.

III.2.1 Volt-Second Balance and Amp-Second balance

It is important to rehash the principles of the inductor volt-second balance and capacitor charge balance in preparing to evaluate the DC/DC converter equations. These principles are inherent in switching type converters, and they also provide additional equations comprised of the unknowns for which we will be solving. A coverter circuit is somewhat different from a typical RLC circuit because of its high frequency switching elements. It difficult to construct a switching converter which converts a pure DC signal without any high frequency switching harmonics [46], therefore, we need a way of representing these small AC fluctuations within our circuit model. Quantities like inductor currents and capacitor voltages can be separated into their DC and AC components as such:

$$\begin{aligned} i_L(t) &= I + i_{ripple}(t) \\ v_c(t) &= V + v_{ripple}(t) \end{aligned} \tag{III.3}$$

These variables can be broken up into a static DC component and a time-varying AC component. At equilibrium the range of $i_{ripple}(t)$ is bounded, therefore, $i_L(t)$ is limited to $I_L \pm \triangle i_L$. We turn to Figure III.10 for a plot of inductor current, $i_L(t)$, over one switching period, T_s.

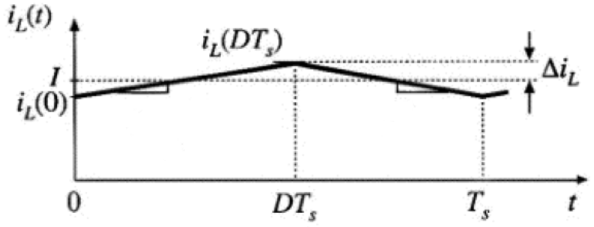


Figure III.10: Inductor current waveform ($i_L(t)$) in a boost mode converter circuit. The current varies about the DC value, I, by $\triangle i_L$.

We note that i has a periodic ripple in steady-state which fluctuates by $\triangle i_L$ about the DC value I. Within steady-state the current at time = 0 is equal to the current at time = T_s, $2T_s$, $3T_s$, and so on. Because of this periodic behavior, we can make an important assumption about the volt-second balance of the inductor which will ultimately aid in solving for the converter's conversion ratio. Starting with the governing equation of an inductor,

$$L\frac{di_L}{dt} = v_L(t) \tag{III.4}$$

Integrating both sides of the expression over one switching period, from 0 to T_s, and dividing by inductance L yields

$$i_L(T_s) - i_L(0) = \frac{1}{L}\int_0^{T_s} v_L(t)dt = 0 \tag{III.5}$$

This inequality becomes 0 because of the periodic nature of the current ripple; the inductor current returns to the same value at the begining of each switching cycle. During each period the current assumes a positive slope within the first interval and a negative slope within the second interval, as in Figure III.10. The slope changes at time = nDT_s, and time = nT_s where n is any integer. The Duty ratio, D, determines the specific fraction of time that is divided between the two modes. Even with varying values of D, the periodic quality of the inductor current ripple is preserved.

Since Eq. (III.5) equates to 0, the integral must also be equal to zero

$$\int_0^{T_s} v_L(t)dt = 0 \tag{III.6}$$

The principle of inductor volt-second balance is based on the fact that Eqs. (III.5) and (III.6) hold true in equilibrium no matter the value of the Duty ratio, as D is absent from the equations. This means that even if D is as small as 0.05−with a positive slope for 5% of the cycle, and a negative slope for 95% of the switching cycle−the current will return to its same initial magnitude at the end of each cycle. We can say that the volt-second value of the first interval is balanced by the volt-second value of the second interval, equating to 0. It is important to note that the SI unit of Eq. (III.6) is volt-seconds. Dividing by T_s changes the SI unit to voltage; more specifically, (III.7) represents average DC inductor voltage.

$$\frac{1}{T_s}\int_0^{T_s} v_L(t)dt = \langle v_L \rangle = 0 \tag{III.7}$$

This expression states that, in equilibrium, the applied inductor voltage must have a 0-valued DC component. If $\langle v_L \rangle$ were somehow non-zero, then the inductor current would grow persistently without bounds, according to Eq. (III.4). The equation in (III.7) can be applied in conjunction with the converter's KVL and KCL equations, and is crucial in solving for the conversion ratio and power losses in the DC/DC converter.

In regards to the inductor's volt-second balance, a similar concept can be applied to the load capacitor within a converter: The amp-second balance describes the behavior of current flowing in and out of the capacitor during switching. Recall that the current of a capacitor is proportional to its voltage differential:

$$C\frac{dv_c}{dt} = i_c(t) \tag{III.8}$$

Proceed to intgrate both sides from 0 to T_s

$$v_c(T_s) - v_c(0) = \frac{1}{C}\int_0^{T_s} i_c(t)dt = 0 \tag{III.9}$$

In equilibrium, the capacitor voltage, $v_c(t)$, posesses small AC components, v_{ripple}, as well as a DC voltage component, V. The peroidic nature of the capacitor voltage means that the left side of Eq. (III.9) equates to 0 volts. From this, we can again make the assumption that the integral on the right hand side is also equal to 0. Eq. (III.9) can be expressed in terms of amp-seconds as,

$$\int_0^{T_s} i_c(t)dt = 0 \tag{III.10}$$

Eq. (III.10) tells us that the charge on the capacitor is balanced between switching intervals, and inherent property of steady-state operation. We can divide by the switching period to express the average capacitor voltage as follows:

$$\frac{1}{T_s}\int_0^{T_s} i_c(t)dt = \langle i_c(t)\rangle = 0 \tag{III.11}$$

As with the volt-second principle, we can apply this similar assumption of Eq. (III.11) towards solving the converter equations. The charge balance in the capacitor has to be zero in steady-state, otherwise the capacitor current, $i_c(t)$, would grow exponentially and render the converter unstable. It should be noted here that the amp-second principle does not apply to a capacitor in parallel with a constant voltage source, because then the capacitor voltage differential, $dv_c(t)/dt$, would be trivial.

III.2.2 Boost Mode Equations

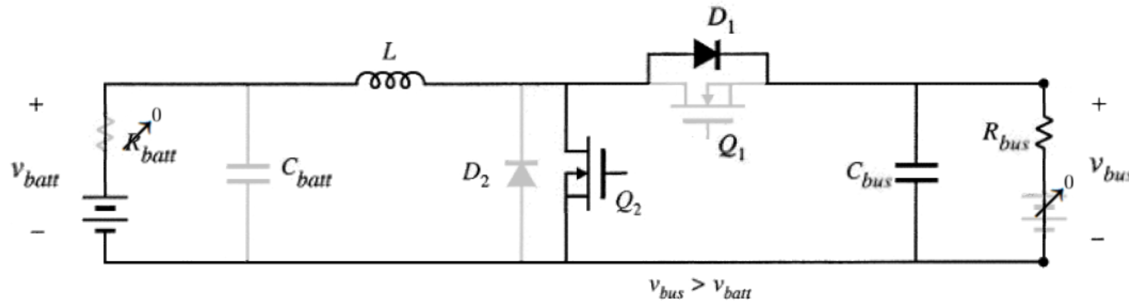


Figure III.11: DC/DC converter boost mode equivalent circuit. Some superflous components are greyed out according to assumptions regarding the circuit topology.

The first course of action in evaluating the DC/DC converter is to make some initial assumptions that will simplify the circuit with regards to the boost mode topology. A diagram of the practical realization of the converter circuit is pictured in Figure III.11. First of all, the buck switch, Q_1, remains off for the entirety of boost mode operation, so we can open up the circuit in place of this component. Diode D_2 remains off, because it will always have a negative voltage across its terminals in this configuration. Likewise, it can be simplified to an open circuit. The constant voltage of the battery drives conversion in boost mode and C_{batt} spans the battery terminals. As previously mentioned, C_{batt} will not obey the charge balance principle since it maintains a constant voltage ($dv(t)/dt = 0$) throughout the entire switching period. With minimal exposure to AC ripple, C_{batt} can be simplified to its DC component disposition: an open circuit. While discharging the internal resistance of the battery is trvially small—on the mΩ scale—so it can be simplified to a closed circuit in the converter model [47]. Finally, we should ignore the bus-side voltage source since the electric motor does not generate voltage on the bus while in forward-drive operation. With these changes implemented, the circuit more closely resembles the canonical boost converter model.

After reducing the circuit, we should proceed by realizing the circuit in both of its switching modes separately. As seen in Figure III.12(a), when the NMOS is ON, a path to ground is established which isolates the battery and inductor from the bus. The battery loop endures nontrivial resistive losses inherent in the inductor and semiconductor, modeled as resistors in series. In the first switching interval the voltage across the inductor is given by

$$\begin{aligned} v_{L,1}(t) &= V_{batt} - iR_L - iR_{on} \\ &\approx V_{batt} - IR_L - IR_{on} \end{aligned} \tag{III.12}$$

The current flowing into the bus capacitor is

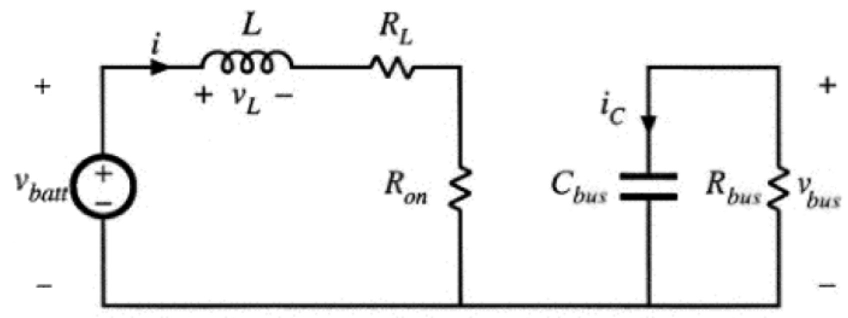
$$i_{C,1}(t) = -\frac{v_{bus}}{R_{bus}} \approx -\frac{V_{bus}}{R_{bus}} \tag{III.13}$$

As we know, voltages and currents within the converter are comprised of a DC signal component coupled with small signal AC ripple. When manipulating equations involving the actual inductor current $i(t)$ as in (III.12), (III.15) and (III.16) above, we can approximate $i(t)$ as the ideal DC signal component, I, if the small ripple approximation holds:

$$if \quad I \gg i_{ripple}(t) \quad then \quad i(t) = I \tag{III.14}$$

Eq. (III.14) outlines the small ripple approximation. This assumption will initially help us

1.)



2.)

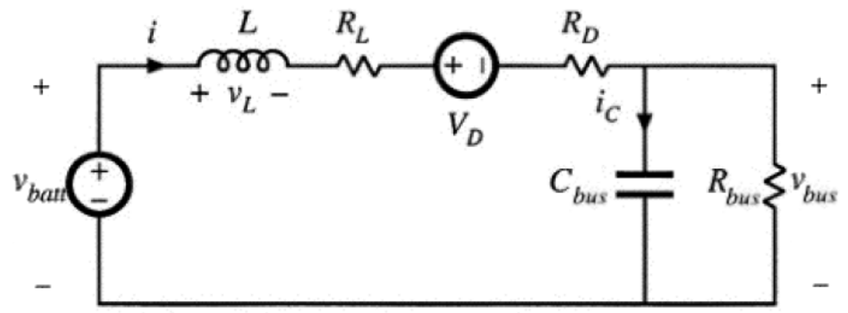


Figure III.12: Boost mode equivalent circuit. (1) ON position during the first subinterval 2.) OFF position during the second subinterval

in determining the average current and voltage values since these DC values do not change over time.

If we turn our attention now to the circuit configuration for the second interval, we see that a new loop arises in which the inductor current directly drives the load on the bus. Consequently, the inductor current ripple will assume a negative slope during this switching interval. With the diode in the ON position, we need to model both the voltage drop and internal resistance of the diode. The equations for inductor voltage and capacitor current in this second stage become

$$\begin{aligned} v_{L,2}(t) &= V_{batt} - iR_L - V_D - iR_D - v_{bus} \\ &\approx V_{batt} - IR_L - V_D - IR_D - V_{bus} \end{aligned} \tag{III.15}$$

$$i_{C,2}(t) = i - \frac{v_{bus}}{R_{bus}} \approx I - \frac{V_{bus}}{R_{bus}} \tag{III.16}$$

Now that we have gathered our inductor voltages for both switching modes, let's examine Eq. (III.5) and evaluate the integral on the right-hand side. Since there is a discontinuity at time = DT_s, it behooves us to split this integral into the sum of two separate integration periods as such:

$$i_L(T_s) - i_L(0) = \frac{1}{L}\left[\ \int_0^{DT_s} v_{L,1}(t)dt\ + \int_{DT_S}^{Ts} v_{L,2}(t)dt\ \right] = 0 \tag{III.17}$$

Since our gathered values of inductor voltage are absent of any time variables, the integration becomes a simple multiplication by t. The value of $v_L(t)$ in each integral was previously determined above: $v_L(t)$ is described by Eq. (III.12) during the first switching interval, and Eq. (III.15) during the second interval. By substituting these values and integrating, we obtain

$$i_L(T_s) - i_L(0) = \frac{1}{L}[\ (DT_s * v_{L_1}(t))\ + (T_s - DT_s) * v_{L_2}(t))\ ] = 0 \tag{III.18}$$

$$= \frac{1}{L}[(DT_s(V_{batt} - IR_L - IR_{on}) + (T_s - DT_s)(V_{batt} - IR_L - V_D - IR_D - V_{bus})] \tag{III.19}$$

By distributing factors, we can reduce Eq. (III.19) as such:

$$i_L(T_s) - i_L(0) = \frac{T_s}{L}[V_{batt} - IR_L - IDR_{on} - (1-D)(V_D + IR_D + V_{bus})] \tag{III.20}$$

Now if we divide both sides by T_s, the SI units of the equation becomes amperes/second, and the left-hand side of the equation resembles Newton's difference quotient for the derivative of i_L.

$$\frac{i_L(T_s+0) - i_L(0)}{T_s} = \frac{1}{L}[V_{batt} - IR_L - IDR_{on} - (1-D)(V_D + IR_D + V_{bus})] \tag{III.21}$$

From our knowledge of calculus, the left side of Eq. (III.21) is equivalent to the following,

$$\frac{i_L(T_s+0)-i_L(0)}{T_s} = \frac{\triangle i_L(t)}{\triangle t} = \frac{di_L(t)}{dt} \tag{III.22}$$

And so finally, by transferring the inductance term across the inequality we can relate our voltage equation to the governing inductor equation.

$$L\frac{di_L(t)}{dt} = [V_{batt} - IR_L - IDR_{on} - (1-D)(V_D + IR_D + V_{bus})] \tag{III.23}$$

Now if we choose to focus only on the DC time-invariant component of the inductor current, $(I \approx i_L(t))$, we can set the right hand side of Eq. (III.23) to 0 according to the volt-second balance principle outlined in Eq. (III.7).

$$L\frac{dI}{dt} = 0 = [V_{batt} - IR_L - IDR_{on} - (1-D)(V_D + IR_D + V_{bus})] \tag{III.24}$$

This equation captures the DC components of voltages around the converter's boost loop with loop current equal to the DC inductor current I. An equivalent circuit corrseponding to this equation is illustrated in Figure III.13. A routine shorthand notation is to express $(1-D)$ as D'.

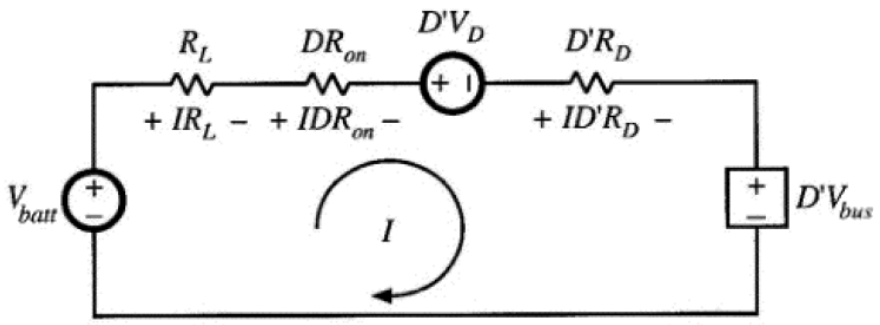


Figure III.13: Equivalent circuit to Eq. (III.24)

We will now follow through with a similar procedure by relating the capacitor charge balance to our capacitor current equations. Starting with Eq. (III.9), and then substituting in the capacitor equations of (III.16) and (III.13) we proceed as follows:

$$v_c(T_s) - v_c(0) = \frac{1}{C}\int_0^{T_s} i_c(t)dt = 0 \tag{III.25}$$

$$v_c(T_s) - v_c(0) = \frac{1}{C}[i_{c,1}(t) * DT_s + i_{c,2} * (T_s - DT_s)] \tag{III.26}$$

$$\frac{v_c(T_s) - v_c(0)}{T_s} = \frac{1}{C}[\frac{-DV_{bus}}{R_{bus}} + (1-D)(I - \frac{V_{bus}}{R_{bus}})] \tag{III.27}$$

Since the left hand side represents the derivative of the capacitor current, we can multiply by C to relate the governing capacitor equation to our circuit equation as shown below

$$C\frac{dv_c(t)}{dt} = \frac{-DV_{bus}}{R_{bus}} + (1-D)(I - \frac{V_{bus}}{R_{bus}}) \tag{III.28}$$

If we wish to focus on the DC value of capacitor voltage, we can apply the capacitor charge balance principle and equate our expression to zero

$$C\frac{dV_{bus}}{dt} = 0 = \frac{-DV_{bus}}{R_{bus}} + (1-D)(I - \frac{V_{bus}}{R_{bus}}) \tag{III.29}$$

The relation can be further reduced by factoring the right-hand side,

$$C\frac{dV_{bus}}{dt} = 0 = \frac{-V_{bus}}{R_{bus}} + (1-D)I \tag{III.30}$$

The KCL equation of Eq. (III.30) can be represented as an equivalent circuit model. Such a model is illustrated in Figure III.14

Now we can perform a novel circuit reduction by first aligning the two circuits in Figures III.13 and III.14 adjacently. This is pictured in Figure III.15(a), where we observe that the inductor circuit on the left has a dependent voltage source which relies on V_{bus} from the capacitor circuit. Furthermore, within the capacitor circuit on the right, there is a current source element that is dependent on the inductor voltage in the opposing circuit

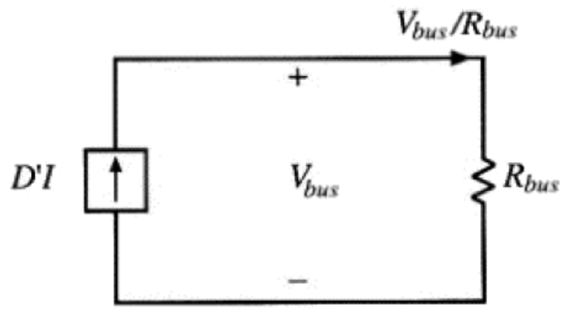


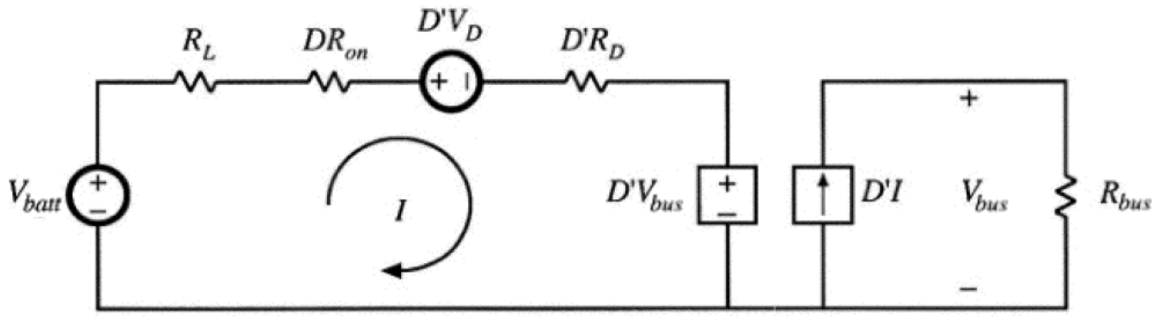
Figure III.14: Equivalent circuit to Eq. (III.30)

model. Based on the definition of a transformer, where current in the primary coil drives voltage in the secondary [46], these two dependent sources can be combined into an ideal DC transformer within the circuit, thereby unifying the converter circuit model. Based on the identical coefficient governing the dependent sources, the transformer will assume a turns ratio of $(1-D):1$, or $D':1$.

It is particularly advantageous to use this circuit reduction technique because it combines both switching modes into a single circuit with all switching behavior accounted for. In order to solve for the bus voltage now, we must employ the fact that passive circuit components can be realized on the opposite side of a transformer by manipulating the impedance of the component. This is necessary since the battery is situated on the primary side of the transformer, yet the output load is on the secondary side. While the turns ratio is given as $D':1$, the impedance turns ratio is the square of this ratio, or $D'^2:1$. Therefore, in order to refer to R_{bus} from the primary side of the transformer, we need to divide its value by 1, and multiply by D'^2. This allows us to use the voltage divider technique to calculate the DC voltage across the output terminals of the converter, spanned by R_{bus}. In our calculation, we must also be aware that V_{batt}, when referred to on the secondary, is equal to the quotient of the primary coil current and the turns ratio.

$$V_{bus} = \frac{(V_{batt} - D'V_D)}{D'} * \left(\frac{(D'^2 R_{bus})}{R_L + DR_{on} + D'R_D + D'^2 R_{bus}}\right) \tag{III.31}$$

(a)



(b)

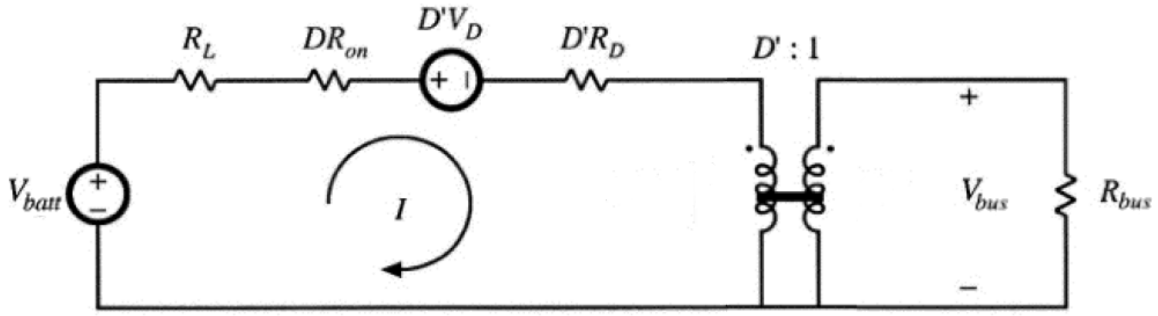


Figure III.15: (a) The equivalent circuit models of Eqs. (III.30) and (III.24) aligned adjacently. (b) The dependent sources then are substituted with an ideal DC transformer.

The conversion ratio, $M(D)$, is the ratio of the input voltage over the output voltage. To obtain $M(D)$, we divide Eq. (III.31) by the battery voltage

$$M(D) = \frac{V_{bus}}{V_{batt}} \tag{III.32}$$

$$M(D) = \frac{1}{D'} * (1 - \frac{D'V_D}{V_{batt}}) \frac{D'^2 R_{bus}}{R_L + DR_{on} + D'R_D + D'^2 R_{bus}} \tag{III.33}$$

The conversion ratio is a dimensionless quantity that tells us the output voltage of a converter when multiplied by a given input voltage. Routinely, the majority of variables within the conversion ratio are determined by fixed elements within the converter except fo D the Duty ratio, and in this case V_{batt} the battery voltage.

We can see from the resistance variables in the denominator of Eq. (III.33) that the internal resistances of the circuit components lead to a deviation from the ideal conversion ratio of $1/D'$. This effect can be minimized if the parasitic resistances are much less than the resistance across the electric bus terminals $(R_L, R_{on}, R_D, R_{bus} \ll R_{bus})$.

From the conversion ratio, the efficiency of the transformer can be easily found. Since the efficiency, η, is equal to the ratio of the output power over input power, we only need to multiply $M(D)$ by the current-conversion ratio, I_{bus}/I_{batt}.

$$\eta = \frac{V_{bus}}{V_{batt}} * \frac{D'I}{I} = (1 - \frac{D'V_D}{V_{batt}}) \frac{D'^2 R_{bus}}{R_L + DR_{on} + DR_D + D'^2 R_{bus}} \tag{III.34}$$

With a maximum efficiency of $\eta = 1$ corresponding to 100%, we see that the greatest efficiency loss occurs as D approaches 1, or equivalently, as D' approaches 0. These power losses prevent this realistic implementation of the converter from approaching infinite voltage conversion as D approaches 1, as it would in the ideal model. However, maximum attainable efficiency can be achieved if the parasitic resistances are much smaller than the effective bus resistance, and if the diode knee voltage is much smaller than the battery voltage:

$$\begin{gathered} (R_L, R_{on}, R_D \ll R_{bus}) \\ (V_D \ll V_{batt}) \end{gathered} \tag{III.35}$$

III.2.3 DCM in Boost Operation

All the analysis that we have done up to this point hinges on the assumption that the AC current ripple is much smaller than the pure DC current within the converter. This is in accordance with small ripple approximation as outlined in Eq. (III.14). However, the specific components that we choose to implement the DC/DC converter may actually invalidate Eq. (III.14), wherein the AC current ripple may be greater than the DC component of the current. To test this we must manipulate our circuit equations from (III.12) and (III.15) to solve numerically for the magnitude of the AC current ripple $\triangle i_L$, as well as the pure DC component I.

Since the parasitic resistances of our implemented design are minimal and conform to the conditions outlined in Eq. (III.35), we are compelled to simplify our inverter voltage equations to facilitate further algebraic manipulation. The equations that we had derived for inductor voltage in Eqs. (III.12) and (III.15) will be manipulated as follows,

$$\begin{aligned} v_{L,1}(t) &\approx V_{batt} - IR_L - IR_{on} \\ &\approx V_{batt} \end{aligned} \tag{III.36}$$

$$\begin{aligned} v_{L,2}(t) &\approx V_{batt} - IR_L - V_D - IR_D - V_{bus} \\ &\approx V_{batt} - V_{bus} \end{aligned} \tag{III.37}$$

We proceed by substituting the simplified equation of $v_{L,1}(t)$ into the governing inductor equation (III.4), and then divide both sides by the inductance variable L

$$\frac{di_{L,1}(t)}{dt} = \frac{v_{L,1}(t)}{L} = \frac{V_{batt}}{L} \tag{III.38}$$

This equation tells us that the slope of $i_L(t)$ during the first subinterval is equal to V_{batt}/L. This is visualized in Figure III.16.

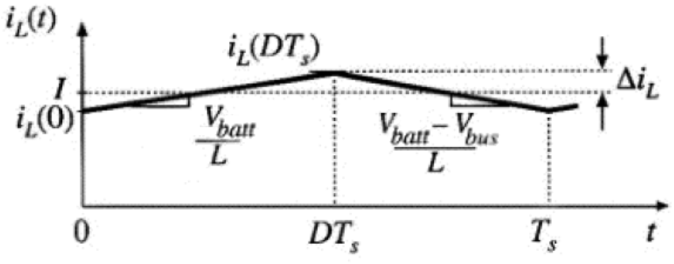


Figure III.16: $i_L(t)$ in CCM where $\triangle i < I$

It similarly follows that the slope of $i_L(t)$ during the second subinterval is equal to $(V_{batt} - V_{bus})/L$.

$$\frac{di_{L,2}(t)}{dt} = \frac{v_{L,2}(t)}{L} = \frac{V_{batt} - V_{bus}}{L} \tag{III.39}$$

While the ripple never deviates more than $\triangle i_L$ from I, the current actually increases or decreases by $2\triangle i_L$ during each subinterval. This is known as the peak-to-peak ripple. Now since we wish to calculate the change in $i_L(t)$ over the first interval, we need to multiply the slope by the duration of the first interval, or DT_s.

$$2\triangle i_L = \frac{V}{L}(DT_s) \tag{III.40}$$

So the ripple magnitude becomes equal to,

$$\triangle i_L = \frac{V_{batt}}{2L}(DT_s) \tag{III.41}$$

By extending this technique to slope of $i_L(t)$ during the second subinterval, we obtain:

$$\triangle i_L = \frac{V_{batt} - V_{bus}}{2L}(D'T_s) \tag{III.42}$$

By recognizing that the lossless conversion ratio specifies $V_{bus}/V_{batt} = 1/D'$, we are able to substitute V_{bus} out of Eq. (III.42)

$$\triangle i_L = \frac{V_{batt} - (V_{batt}/D')}{2L}(D'T_s) \tag{III.43}$$

Hence, we have solved for $\triangle i_L$.

To solve for the DC value of current I, we turn to our derived capacitor charge balance equation (III.30) and solve for I as follows:

$$C\frac{dV_{bus}}{dt} = 0 = \frac{-V_{bus}}{R_{bus}} + D'I \tag{III.44}$$

$$\frac{V_{bus}}{D'R_{bus}} = I \tag{III.45}$$

Again, V_{bus} can be substituted out using the ideal conversion ratio,

$$I = \frac{V_{batt}/D'}{D'R_{bus}} = \frac{V_{batt}}{D'^2 R_{bus}} \tag{III.46}$$

So now that we have derived $\triangle i_L$ and I, let's find the criteria for continuous conduction mode (CCM), and discontinuous conduction mode (DCM). If the AC ripple is greater in magnitude than the DC current, then the converter will be operating in DCM. We substitute the current variables from Eqs. (III.41) and (III.46) as follows,

$$\triangle i_L > I \tag{III.47}$$

$$\frac{V_{batt}}{2L}(DT_s) > \frac{V_{batt}}{D'^2 R_{bus}} \tag{III.48}$$

Now we push the component related variables to the right-hand side, and the duty ratio variables to the left-side,

$$DCM\ if \quad D * D'^2 > \frac{2L}{RT_s} \tag{III.49}$$

$$DCM\ if \quad K_{crit}(D) > K \tag{III.50}$$

From this we gather that the converter circuit enters DCM once the duty ratio-dependent value $K_{crit}(D)$ becomes greater than the component-depdendent K value.

We can now numerically evaluate this expression. Our modeled DC/DC converter has the following component values: $L = 1.4$mH, $T_s = 50$us, $R = 1960\Omega$. Solving for K yields

$$K = \frac{2L}{RT_s} = \frac{2(1.4mh)}{(1960\Omega)(50us)} = 0.0286 \tag{III.51}$$

Now if we solve $K_{crit}(D) > 0.0286$ for D, we can find the range of possible duty ratios in which the circuit is in DCM for our boost converter topology. The result yields,

$$D * D'^2 > 0.0286 \tag{III.52}$$

$$DCM\ if \quad 0.0304 < D < 0.812 \tag{III.53}$$

This means that the converter will enter DCM for a majority of the range of possible duty ratios. A quick calculation employing typical voltage values in conjunction with the ideal conversion ratio tells us what D is required for boost mode operation in the hybrid vehicle.

$$M(D) = \frac{1}{D'} = \frac{V_{bus}}{V_{batt}} = \frac{600V}{300V} \tag{III.54}$$

$$D = 0.5 \tag{III.55}$$

This result lies in the middle of the range spanned by Eq. (III.53), therefore, our converter will operate firmly in DCM.

Now that we have determined that our converter operates in DCM we need to alter our realization of the circuit model. One might expect the inductor current to briefly become negative if $\triangle i_L > I$. However, if we refer back to Figure III.11, we observe that diode $D1$ only permits the flow of positive $i_L(t)$. This is of particular importance when the $Q2$ MOS device is OFF. What this means is that the circuit assumes a new third switching mode when $i_L(t)$ becomes negative, which is different from the switching modes in Figure III.12. Figure III.17 depicts this third switching mode which occurs when the diode current reaches zero in the second subinterval.

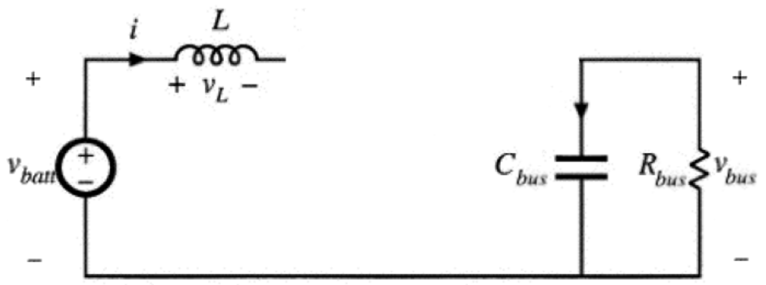


Figure III.17: The DCM subinterval where $D1$ turns off while $Q1$ is OFF

We observe that the inductor cannot pass current to the rest of the circuit within this subinterval and remains isolated from the bus terminals. If we turn to Figure III.18, we see the effect of this switching mode on the inductor current $i_L(t)$.

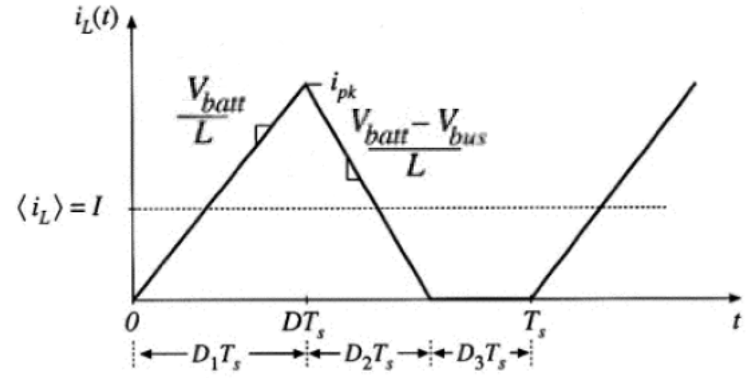


Figure III.18: A third subinterval D_3 arises while in DCM where $i_L(t) = 0$

We can see that the current ripple is now separated into 3 distinct subintervals, with D_3 representing the discontinuous interval. In order to solve for the DCM conversion ratio, we need to combine the equations of these three subintervals. We have already determined the inductor voltage and capacitor current for the first two intervals. For the first interval we have

$$\begin{aligned} v_L(t) &= V_{batt} \\ i_c(t) &= -V_{Bus}/R \end{aligned} \tag{III.56}$$

Similarly, for the second interval

$$\begin{aligned} v_L(t) &= V_{batt} - V_{bus} \\ i_c(t) &= -V_{Bus}/R + i_L(t) \end{aligned} \tag{III.57}$$

Lastly, within the third subinterval we have

$$\begin{aligned} v_L(t) &= 0 \\ i_c(t) &= -V_{Bus}/R \end{aligned} \tag{III.58}$$

The small ripple approximation (III.14) has been applied to the voltages, but not to the inductor current. Parasitic resistances are trivial given the chosen component values and will be left out during this analysis.

We can relate the voltage across the three subintervals via the volt-second balance principle (III.5). We need to introduce variables D_2 and D_3, corresponding to the fractional length of the second and third subintervals.

$$\langle v_L \rangle = 0 = D_1 V_{batt} + D_2(V_{batt} - V_{bus}) + D_3(0) \tag{III.59}$$

If we isolate D_2 on the left-hand side we are left with

$$D_2 = \frac{V_{batt} D_1}{V_{bus} - V_{batt}} \tag{III.60}$$

We require an additional equation in order to solve for all the unknowns in our system, so we direct attention towards the diode current $i_D(t)$. If we look back at Figure III.11, we see that the diode feeds directly into the output node on the bus. The resulting KCL equation is:

$$i_c(t) = i_D(t) - \frac{V_{bus}}{R} \tag{III.61}$$

If we focus only on the average current values, we can apply the charge balance principle (III.11) which gives us,

$$\langle i_D(t) \rangle = \frac{V_{bus}}{R} \tag{III.62}$$

The average diode current is nontrivial, and can be found by integrating the diode current over the entire switching interval $0 \to T_s$. We know that the diode only conducts during the second subinterval, and that its current is equal to $i_L(t)$ during this time. If we focus only on the second subinterval, Figure III.18 provides an accurate representation of $i_D(t)$. The diode current starts at the i_{peak} value, and then drops gradually to 0A where it remains until the second subinterval again. The integral of $i_D(t)$ would be equal to this area under the triangle within the second interval. Since $i_L(t)$ and $i_D(t)$ both experience the same peak current value we can form an expression for the i_{peak} value based on the slope of the inductor current.

$$i_{peak} = \frac{v_{L,1}}{L} * D_1 T_s = \frac{V_{batt}}{L} * D_1 T_s \tag{III.63}$$

Now that we have an expression for i_{peak} we can easily evaluate the integratal for the average value of $i_D(t)$ by recognizing that the area under $i_D(t)$ is of triangular geometry.

$$\langle i_D(t) \rangle = \frac{1}{T_s} \int_0^{T_s} i_D(t) dt = \frac{1}{2} i_{peak} * D_2 = \frac{V_{batt} D_1 D_2 T_s}{2L} \tag{III.64}$$

We can now equate the right-hand side to the definition of $\langle i_D(t) \rangle$ from Eq. (III.62) as follows

$$\langle i_D(t) \rangle = \frac{V_{bus}}{R} = \frac{V_{batt} D_1 D_2 T_s}{2L} \tag{III.65}$$

By substituting Eq. (III.60) for D_2, we achieve

$$\frac{V_{bus}}{R} = \frac{V_{batt} D_1 (D_1 \frac{V_{batt}}{V_{bus} - V_{batt}}) T_s}{2L} \tag{III.66}$$

We recognize that the definition of K, as derived in Eq. (III.49), is present in our equation:

$$V_{bus} = \frac{1}{K} \frac{V_{batt}^2 D_1^2}{V_{bus} - V_{batt}} \tag{III.67}$$

$$0 = V_{bus}^2 - V_{batt} V_{bus} - \frac{V_{batt}^2 D_1^2}{K} \tag{III.68}$$

The quadratic formula is now employed leaving us with

$$V_{bus} = \frac{V_{batt} \pm V_{batt} \sqrt{1 + \frac{4D_1^2}{K}}}{2} \tag{III.69}$$

Finally, by dividing both sides by V_{batt} we are left with the formula for the DCM conversion ratio which is a function of the duty ratio, and also the K value.

$$M(D,K) = \frac{V_{bus}}{V_{batt}} = \frac{1 + \sqrt{1 + 4D^2/K}}{2} \tag{III.70}$$

Since $1 \ll 4D^2/K$ under the radical, the conversion ratio is approximately equal to:

$$M(D,K) \approx \frac{1}{2} + \frac{D}{\sqrt{K}} \tag{III.71}$$

This function intersects with the ideal conversion ratio, $1/D'$, at limits previously specified by the $K_{crit}(D)$ equation (III.53). With this new conversion ratio for DCM, we observe that the required duty ratio for typical boosting within the hybrid vehicle is different from the CCM value calculated in Eq. (III.55).

$$M(D) \approx \frac{1}{2} + \frac{D}{\sqrt{K}} = \frac{V_{bus}}{V_{batt}} \tag{III.72}$$

$$\frac{1}{2} + \frac{D}{\sqrt{0.0286}} = \frac{600V}{300V} \tag{III.73}$$

$$D = 0.254 \tag{III.74}$$

This value of D is much different from the CCM D value, so the function of the circuit is significantly transformed when the conditions are right for DCM. We could have chosen a higher value of L for the converter which would have increased the K value above the $K_{crit}(D)$ value and ensured CCM; however, because of the properties of an inductor (III.4), the time to reach steady-state, or response time, would be reduced with such a decision.

III.2.4 Buck Mode Equations

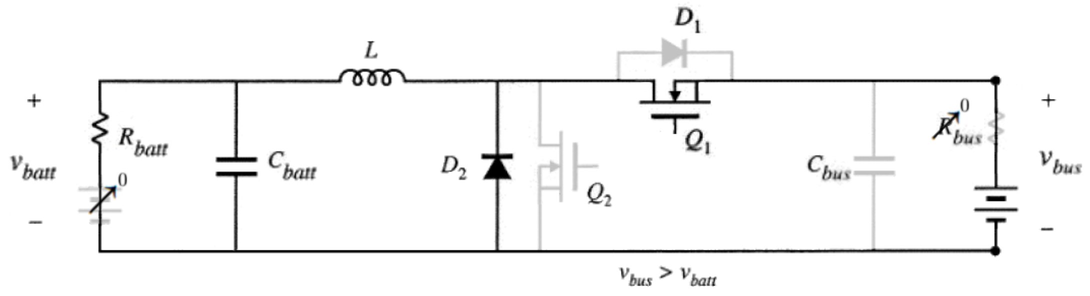


Figure III.19: DC/DC converter circuit with practical switch realization. The buck mode equivalent circuit is achieved after performing circuit reudctions.

As with the boost mode circuit, we can start our analysis by intelligently eliminating some circuit components. Q_2 and D_2 will always remain OFF in buck mode, and can be removed from the circuit model. Since the capacitor C_{bus} maintains a constant voltage across its terminals in steady-state, we can also eliminate it from the circuit. The bus resistance R_{bus} becomes negligible in buck mode and can be removed. Finally, while the battery is in a charging state it should be treated as a resistive load, and not as a constant voltage source, therefore, we neglect the battery cell voltage during buck analysis [47].

We start our analysis in interval 1 as pictured in Figure III.20.1. After employing the small ripple approximation (III.14), voltage across the inductor is given by

$$\begin{aligned} v_{L,1}(t) &= (V_{bus} - iR_L - iR_{on}) - v_{batt} \\ &\approx (V_{bus} - IR_L - IR_{on}) - V_{batt} \end{aligned} \tag{III.75}$$

Likewise, current flowing into the battery capacitor is equal to

$$\begin{aligned} i_{c,1}(t) &= i - v_{batt}/R \\ &\approx I - V_{batt}/R \end{aligned} \tag{III.76}$$

Next we form the inductor voltage and capacitor current equations pertinent to the second subinterval, pictured in Figure III.20.2. Focusing on the inductor voltage we gather:

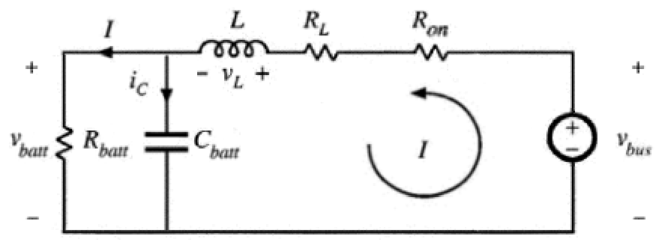
$$\begin{aligned} v_{L,2}(t) &= (-iR_L - iR_D - V_D) - v_{batt} \\ &\approx (-IR_L - IR_D - V_D) - V_{batt} \end{aligned} \tag{III.77}$$

The current flowing into C_{batt} is identical to its value during the first subinterval

$$\begin{aligned} i_{c,2}(t) &= i - v_{batt}/R \\ &\approx I - V_{batt}/R \end{aligned} \tag{III.78}$$

Since the capacitor current is the same in each operating mode, we can derive the following

(1)



(2)

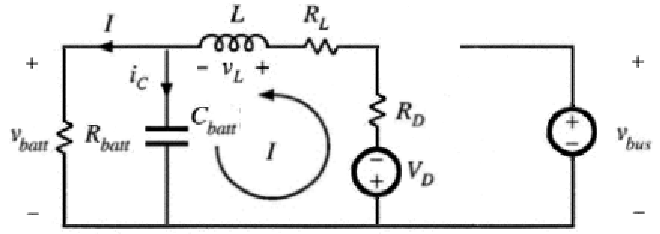


Figure III.20: Buck mode equivalent circuit (1) during the first subinterval, and (2) during the second subinterval

equation relating I and V_{batt}

$$\begin{aligned} \langle i_c(t) \rangle = \quad 0 \quad &= I - V_{batt}/R \\ V_{batt}/R &= I \end{aligned} \tag{III.79}$$

By employing the volt-second balance principle (III.7), a KCL equation can be derived by the inductor voltage expressions:

$$\langle v_L(t) \rangle = 0 = D v_{L,1}(t) + D' v_{L,2}(t) \tag{III.80}$$

$$0 = DV_{bus} - IR_L - DIR_{on} - D'IR_D - D'V_D - V_{batt} \tag{III.81}$$

By substituting in Eq. (III.79), we are left with

$$0 = DV_{bus} - IR_L - DIR_{on} - D'IR_D - D'V_D - IR_{batt} \tag{III.82}$$

An equivalent circuit model of this equation is represented in Figure III.21.

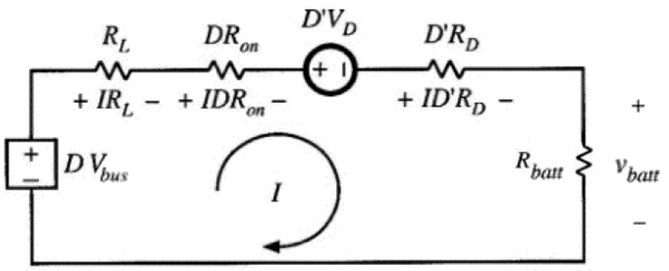


Figure III.21: Equivalent circuit to Eq. (III.79) governing buck mode operation.

As with the boost converter, a buck converter can be equivalently modeled by an ideal DC transformer with a turns ratio equal to the conversion ratio [47], yet there is no I_{bus} equation corresponding to the bus-voltage dependent source. We can quickly derive an equation for I_{bus}, given our knowledge that the bus current is zero for the second subinterval, and that $I_{bus} = i_L(t)$ during the first subinterval. Therefore,

$$I_{bus} = DI \tag{III.83}$$

The equivalent circuit for this equation is pictured in Figure III.22. These two equivalent circuits can be joiend together by replacing their dependent sources with an ideal DC transformer. The result is pictured in Figure III.23.

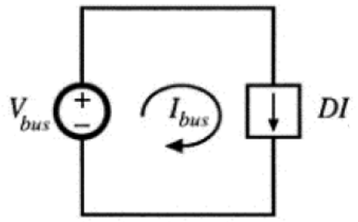


Figure III.22: Equivalent circuit to Eq. (III.83)

From this equivalent circuit, we can derive the equation for the buck mode conversion ratio. By employing the voltage division rule across R_{batt} we can write an expression for

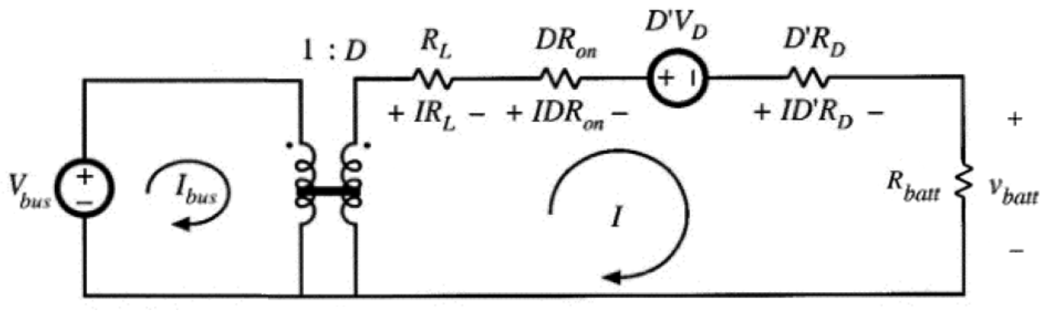


Figure III.23: Equations (III.79) and (III.83) are combined by realizing an ideal DC converter from the dependent sources.

V_{batt} as follows.

$$V_{batt} = D(V_{bus} - \frac{D'V_D}{D}) * (\frac{R_{batt}}{R_L + DR_{on} + D'R_D + R_{batt}}) \tag{III.84}$$

We can derive the conversion ratio by dividing V_{batt} by the input voltage, V_{bus}.

$$M(D) = \frac{V_{batt}}{V_{bus}} \tag{III.85}$$

$$M(D) = D(1 - \frac{D'V_D}{DV_{bus}}) * (\frac{R_{batt}}{R_L + DR_{on} + D'R_D + R_{batt}}) \tag{III.86}$$

When the diode voltage and resistive losses are small as expressed in Eq. (III.35), the conversion ratio in Eq. (III.86) conforms to the ideal conversion ratio for a buck converter ($M(D) = D$). A low duty ratio corresponds to greater voltage conversion, compared to boost mode operation where a high duty ratio corresponds to greater voltage conversion.

To solve for the efficiency, we must manipulate our equation in (III.86) so that we have an expression for P_{out}/P_{in}. This is accomplished by multiplying the conversion ratio by I_{batt}/I_{bus} as follows,

$$\eta = \frac{V_{batt}}{V_{bus}} * \frac{I}{DI} = (1 - \frac{D'V_D}{DV_{bus}}) * (\frac{R_{batt}}{R_L + DR_{on} + D'R_D + R_{batt}}) \tag{III.87}$$

As with boost mode operation, maximum efficiency is maintained when the parasitic resistances are as small as possible compared to the R_{batt}. From the similarity of Eqs. (III.86) and (III.87), we can form a simple expression relating the ideal conversion ratio to the practical value:

$$M(D) = M_{ideal}(D) * \eta \tag{III.88}$$

Hence, the practical conversion ratio is the product of the ideal ratio and the efficiency function. In our DC/DC converter implementation the parasitic resistances are kept small to avoid loss of efficiency. Even though there is some power lost as a result of buck mode conversion, this loss is relatively small compared to the losses experienced in a step-down voltage divder.

III.2.5 DCM in Buck Mode

Now that we have derived the conversion ratio and efficiency for continuous circuit mode, we should check that the small-ripple assumption (III.14) holds true for the inductor current. Our hybrid vehicle implementation requires a small inductor so that the converter can quickly respond to mode changes, however, small converter inductors tend to bring about DCM conversion. To check our ripple assumptions, we must calculate an expression for $\triangle i_L$. In doing so we first must alter our volt-second and amp-second expressions to account for the fact that the components chosen for our implementation render the parasitic resistances negligible according to (III.35). Hence, our simplified circuit equations resemble

$$\begin{aligned} \langle v_L(t) \rangle &= 0 = DV_{bus} - V_{batt} \\ \langle i_c(t) \rangle &= 0 = i(t) - V_{batt}/R_{batt} \end{aligned} \tag{III.89}$$

In the second subinterval, $v_L(t)$ becomes equal to $(-V_{batt})$ as gathered from these equations. Moreover, the relationship between inductor voltage and $di_L(t)/dt$ allows us to solve for the magnitude of the inductor current ripple via the governing equation (III.4). Since we know that the inductor current changes by $2\triangle i_L$ during a subinterval and that the duration of the second subinterval is $D'T_s$, we can solve for $\triangle i_L$ as such,

$$\frac{di_L(t)}{dt} = \frac{v_{L,2}(t)}{L} = -\frac{V_{batt}}{L} \tag{III.90}$$

$$-2\triangle i_L = \frac{di_L(t)}{dt} D'T_s = -\frac{V_{batt}}{L} D'T_s \tag{III.91}$$

$$\triangle i_L = \frac{V_{batt}}{2L} D'T_s \tag{III.92}$$

Now that we have solved for $\triangle i_L$ and I, we can form an expression to check whether the current ripple is greater than the DC current value.

The converter will be operating in DCM if

$$\triangle i_L > I \tag{III.93}$$

$$\frac{V_{batt}}{2L} D'T_s > \frac{V_{batt}}{R_{batt}} \tag{III.94}$$

$$DCM \;\; if \quad D' > \frac{2L}{R_{batt} T_s} \tag{III.95}$$

$$DCM \;\; if \quad K_{crit}(D) > K \tag{III.96}$$

This leaves us with the $K_{crit}(D)$ equation for a buck converter. The K term is not specific to a boost converter, as the same expression for K was derived for the boost converter in Eq. (III.49). Therefore, we can substitute the numeric value of K from our boost converter calculations into Eq. (III.95), and solve for the critical duty ratio:

$$\begin{gathered} DCM \;\; if \quad D > 0.0286 \\ DCM \;\; if \quad D < 0.971 \end{gathered} \tag{III.97}$$

Eq. (III.97) tells us that the converter circuit operates in DCM mode over approximately 97% of the duty ratio range with our implementation. Therefore, we are compelled to calculate the conversion ratio for the buck converter within DCM.

As we discussed earlier, a converter with a large ripple and small DC value runs the risk of turning OFF the diode (D_2 in this case) used to realize an ideal switch (B). The buck mode converter cannot operate in continuous mode if at any time $i_L(t)$ drops to zero amps. Since we have assurance that the inductor current will drop to zero within our implementation, we must evaluate the buck mode circuit with the expectation that the diode D_2 will shut off during the second subinterval, but before the begining of the first interval. This third interval of operation is pictured in Figure III.24.

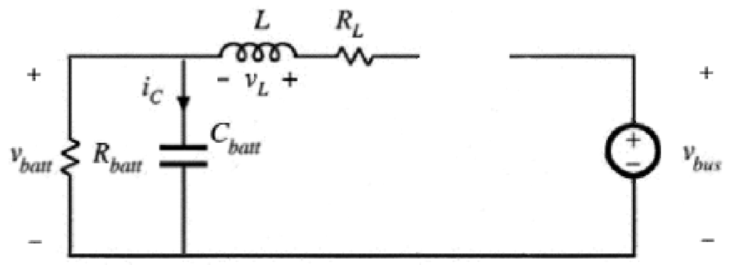


Figure III.24: The third subinterval of buck mode operation within DCM

With both Q_1 and D_2 off, the inductor is no longer able to pass current. If the inductor current were somehow non-zero upon entering this mode, then arcing would occur across the open circuit because inductors cannot realize instantaneous current change. For this third subinterval we create expressions for $v_{L,3}(t)$ and $i_{c,3}(t)$,

$$\begin{aligned} v_{L,3}(t) &= V_{batt} - V_{batt} = 0 \\ i_{c,3}(t) &= i_L - V_{batt}/R_{batt} \end{aligned} \tag{III.98}$$

Now we employ these third interval expressions in conjunction with expressions for the first two intervals by recognizing that $\langle v_L(t) \rangle$ and $\langle i_c(t) \rangle$ are equal to 0 across all three intervals, according to the volt-second and amp-second principles of Eqs. (III.5) and (III.10). Hence we have,

$$\langle v_L(t) \rangle = 0 = D_1(V_{bus} - V_{batt}) + D_2(-V_{batt}) + D_3(0) \tag{III.99}$$

This can be rearranged as,

$$D_2 = D_1 \frac{(V_{bus} - V_{batt})}{V_{batt}} \qquad -or- \qquad (D_1 + D_2) = D_1 \frac{V_{bus}}{V_{batt}} \tag{III.100}$$

This provides us an expression for D_2. Equivalently with $\langle i_c(t) \rangle$ we have,

$$\langle i_c(t) \rangle = 0 = D_1(i_L - \frac{V_{batt}}{R_{batt}}) + D_2(i_L - \frac{V_{batt}}{R_{batt}}) + D_3(i_L - \frac{V_{batt}}{R_{batt}}) \tag{III.101}$$

$$0 = i_L - \frac{V_{batt}}{R_{batt}} \tag{III.102}$$

We need to formulate an addition equation from the circuit in order to account for the addition of new unknowns. Since we known the voltage across the inductor, we can easily integrate to solve for the $\langle i_L \rangle$. We start by determining the peak current value, i_{peak}. It is calculated as follows,

$$i_{peak} = \frac{di_L(t)}{dt}(D_1 T_s) = \frac{v_{L,1}}{L}(D_1 T_s) = \frac{(V_{bus} - V_{batt})}{L}(D_1 T_s) \tag{III.103}$$

Subsequently, the integral for $\langle i_L \rangle$ is evaluated as the area beneath a triangle.

$$\langle i_L \rangle = \frac{1}{T_s} \int_0^{T_s} i_L dt = \frac{1}{2} i_{peak}(D_1 + D_2) = \frac{(V_{bus} - V_{batt})}{2L}(D_1 T_s)(D_1 + D_2) \tag{III.104}$$

The average inductor current is not equal to zero. We substitute the definition of i_L from Eq. (III.102) into Eq. (III.104):

$$\frac{V_{batt}}{R_{batt}} = \frac{(V_{bus} - V_{batt})}{2L}(D_1 T_s)(D_1 + D_2) \tag{III.105}$$

Furthermore, we have a known expression for $(D_1 + D_2)$ so we can substitute Eq. (III.100) into Eq. (III.105)

$$\frac{V_{batt}}{R_{batt}} = \frac{(V_{bus} - V_{batt})}{2L}(D_1 T_s)(D_1 \frac{V_{bus}}{V_{batt}}) \tag{III.106}$$

To simplify our algebra we substitute for the definition of K,

$$V_{batt} = \frac{(V_{bus} - V_{batt})}{K}(D_1^2)(\frac{V_{bus}}{V_{batt}}) \tag{III.107}$$

This equation expands to a quadratic expression of the form:

$$0 = V_{bus}^2 - V_{bus}V_{batt} - \frac{V_{batt}K}{D_1^2} \tag{III.108}$$

Solving for the roots of V_{bus}, we obtain,

$$V_{bus} = \frac{V_{batt} \pm \sqrt{V_{batt}^2 + \frac{4V_{batt}^2 K}{D_1^2}}}{2} \tag{III.109}$$

$$V_{bus} = V_{batt}\frac{1 \pm \sqrt{1 + 4K/D^2}}{2} \tag{III.110}$$

Since the buck converter does not invert the voltage we select the positive root, and then isolate V_{batt}/V_{bus} to obtain the final DCM conversion ratio.

$$M(D,K) = \frac{V_{batt}}{V_{bus}} = \frac{2}{1 + \sqrt{1 + 4K/D^2}} \tag{III.111}$$

This final expression is the conversion ratio for a buck converter operating in DCM; it is a function of the both the duty ratio D, and the technology coefficient K. Given the components chosen for our converter implementation, this conversion ratio will be invoked while recharging the battery instead of the CCM conversion ratio $(M(D) = D)$. Figure III.25 demonstrates how the duty ratio is altered by DCM effects.

As we can see, DCM conversion drastically changes the shape of the conversion ratio function. The DCM conversion ratio makes it particularly difficult to regulate low-end voltages by lessening the precision that the duty ratio has against the conversion ratio. DCM behavior appears once $K_{crit}(D) = D'$ drops below K. The maximum possible value

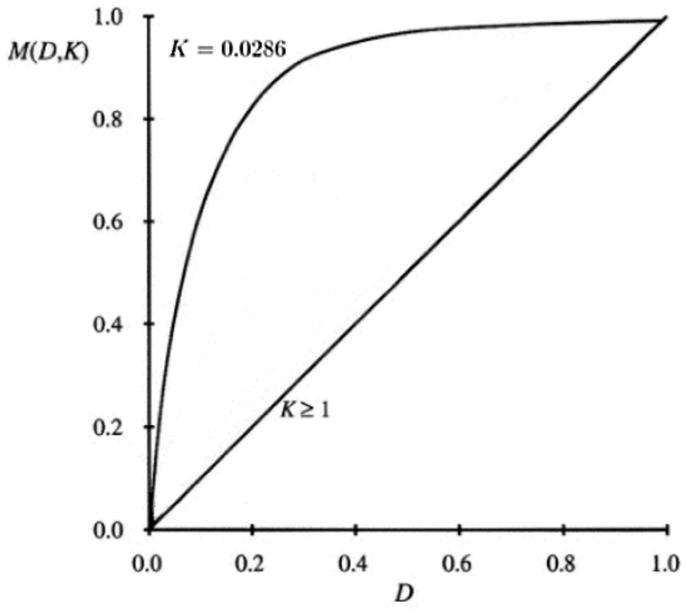


Figure III.25: Plot of the conversion ratio vs. duty, where $K \geq 1$ (CCM), and K = 0.0286 (DCM).

for $K_{crit}(D)$ is 1, so if $K > 1$ then the circuit will operate in full CCM.

The equations that we derived in this section regarding both buck and boost mode govern modeled converter behavior, and therefore, actual converter behavior. We will employ these equations towards designing a PWM and selecting component values. Furthermore, these equations will be used to check the validity of the modeled bidirectional converter.

III.3 Model Synthesis

To test the behavior of the proposed DC/DC converter, we need to instantiate a model of the converter within the ESMoL tool suite. The ESMoL framework allows designers to either import a model created in Simulink, or directly compose a model within the GME modeling environment. We will provide a thorough study of utilizing both these avenues to model the behavior of the cyber-physical converter design.

III.3.1 Dynamic Circuit Composition

We start by logically separating the computational components of the DC/DC converter, from the physical circuit elements. The circuit consists of the electronic components of the DC/DC converter which were analyzed in section 3.2, as well as a PWM component. The ESMoL's CyPhyML integration language utilizes the operational semantics of Modelica for performing simulations, therefore, we will use the Modelica environment for constructing and debugging the converter circuit model. Once we have achieved correctness we can easily realize the model within CyPhyML where it can be integrated with the controller component.

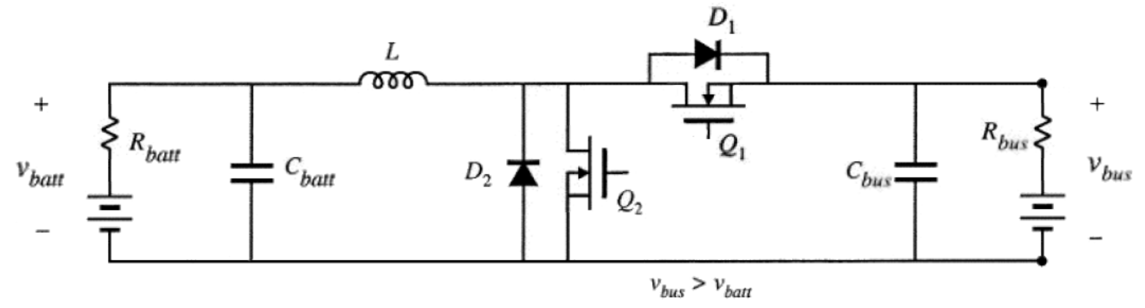


Figure III.26: Full bidirectional DC/DC converter model

The bidirectional DC/DC converter is created within Modelica by instantiating the circuit components, and then connecting their terminals together according to the circuit model in Figure III.26. This is a relatively simple process considering the Modelica block library already contains definitions for all of the required circuit components within the converter. The components used are resistors, capacitors, inductors, diodes, and NMOS transistors. The model will also include an abstract component that allows a Modelica signal to control an electrical voltage source; this is necsesary to allow the PWM to deliver a V_{GS} voltage to the MOS switches, thereby turning them ON or OFF.

The realization of the bidirectional DC/DC converter within Modelica is pictured in Figure III.27. The component relations are easily established by clicking on both endpoints of an intended device connection. An electrical connection represents an equation which

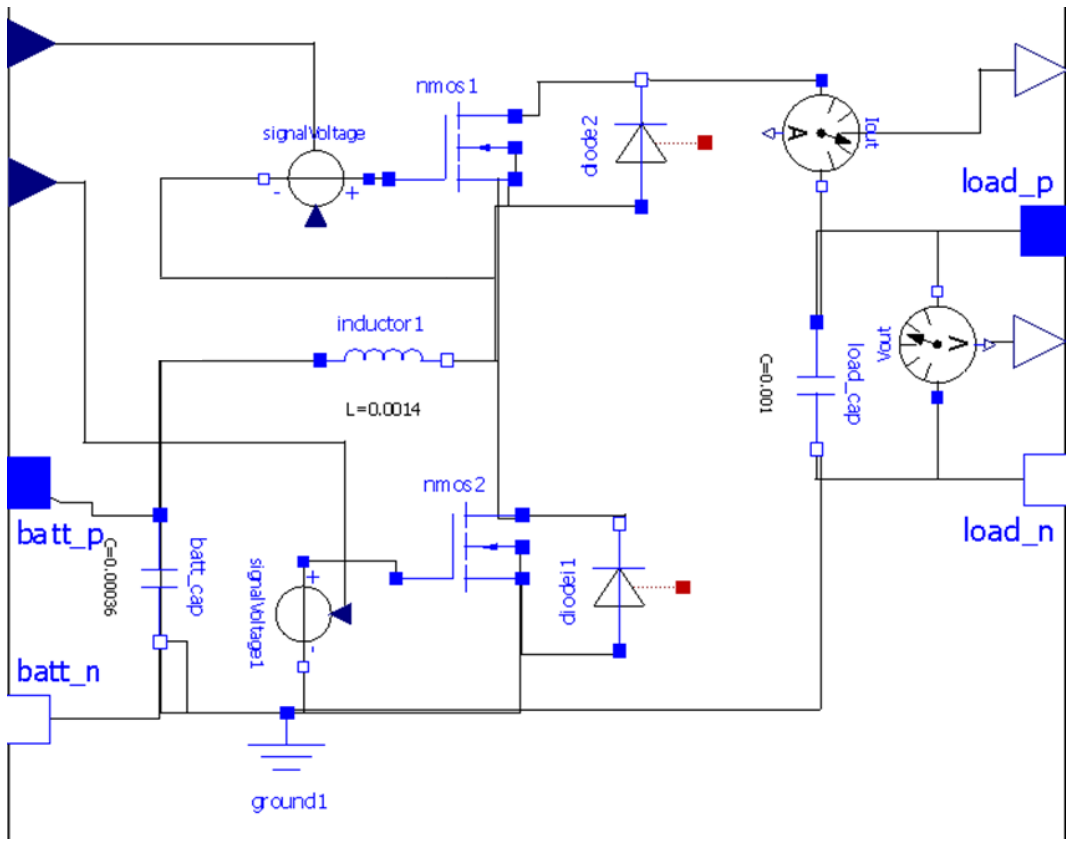
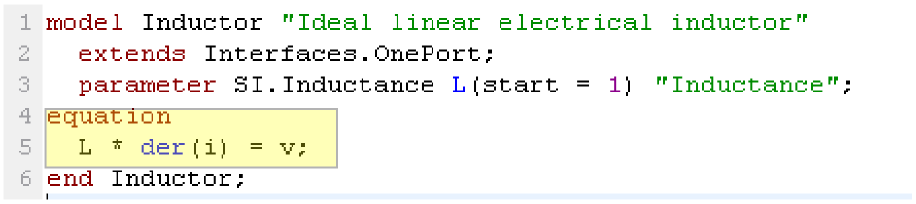


Figure III.27: Full bidirectional DC/DC converter realization within Modelica

bridges current and voltage variables between separate components in a bidirectional manner according to their modeled behavior. Component definitions are descriptive equations based on the electrical properties that govern the real physical component. For example, the governing equation of an inductor, as given in Eq (III.4), is the same semantic definition used by Modelica to represent the inductor model.

$$L\frac{di(t)}{dt} = v_L(t) \quad \equiv \quad L * der(i) = v \tag{III.112}$$

As seen in the highlighted portion of Figure III.28, the behavior of the inductor is not sequential code but instead a simple physical relationship expressed as a differential equation. The Modelica framework handles the time evolution of this equation based on the inductance parameter, *L*, and also on the governing equations of interconnected components.



```
model Inductor "Ideal linear electrical inductor"
  extends Interfaces.OnePort;
  parameter SI.Inductance L(start = 1) "Inductance";
equation
  L * der(i) = v;
end Inductor;
```

Figure III.28: The defining equation of the inductor component within Modelica

As intended, the circuit in Figure III.27 visually resembles the circuit diagram configuration. Only the power transformation components are represented in this model as the bus and battery components are modeled as separate entities. There are clearly labeled terminals on either side of the converter that allow the bus and battery to be seamlessly integrated into the circuit. We can see various values of the electrical components in this model view, for example the inductor value is shown as 1.4mH: This low inductance value was chosen in order to elicit quick voltage transitions from the circuitry, however, the low inductance value is what causes the circuit to operate primarily in DCM. Component values are easily specified in Modelica by modifying a component's properties. Also visible in the model are the component values of the bus and battery capacitors. The capacitance of C_{bus} was purposefully chosen to be greater than the capacitance of C_{batt} $(1000\mu F > 360\mu F)$ because the bus experiences proportionally higher voltage and current compared to the battery.

If we turn attention to the battery component model of Figure III.29, we see that its composition is not merely a static resistor in series with a constant voltage source. Instead we observe a variable resistor and a variable voltage source which are environmentally triggered by the vehicle's current mode of operation. Although not pictured, the bus component is modeled in an identical fashion with a variable source and resistor. In this way we can emulate an environmental interface to the battery and bus components so that their dynamic component values correctly correspond to the current mode of operation of the vehicle. This modular control is necessary, for example, because the battery assumes much different internal resistance while discharging compared to while charging. Additionally the electric bus can produce hundreds of volts of induced charge during regenerative braking, and then appropriately zero volts of sourced voltage during hybrid drive. As a rule of thumb, if the bus or battery are not presently sourcing voltage, then they become primarily a resistive load.

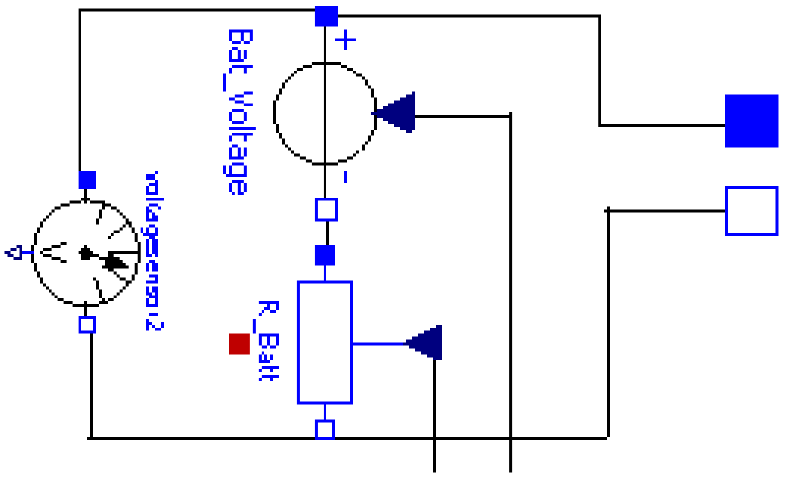


Figure III.29: The battery component as defined within Modelica, consisting of a voltage source, a resistor, and a voltage sensor.

An abstract component called the mode-set specifies the voltage and resistance values of the bus and battery according to the operating mode of the hybrid vehicle. For modeling and simulation purposes, this virtual component acts in direct response to the vehicle software controller's input from the environment (ie. the Brake_ON signal). Since we have not explicitly modeled the electric motor or any translational motion effects, the mode-set provides a way of emulating practical driving behavior of the hybrid vehicle so that we can simulate the effect of mode transitions on the bidirectional DC/DC converter.

Apparent in Figure III.29 is a voltage sensing element coupled across the terminals of the battery. The sensed voltage is delivered to the PWM component so that we can achieve closed-loop feedback control. The PWM is fed a target voltage value from the converter's controller, and this target value is variable depending on the level of desired acceleration for example. If the PWM senses that the boosted output voltage is too low across the bus, then it will conduct switching with a progressively higher duty ratio D. Dynamic feedback is necessary in the hybrid vehicle implementation because the resistance of the electric motor load is not always known.

The PWM has two outputs which each control the NMOS switches within the converter. The device creates a pulse width modulated signal exclusively on one of the MOS devices, while the other MOS device remains in the OFF position. The Q_1 switch is modulated during bucking and remains completely OFF during buck mode operation, and vice versa for the Q_2 switch. The direction of power conversion is governed by the current operating mode of the vehicle, and directly enforced in the PWM by the vehicle software controller. The modulated signal generated by the PWM is initially high until DT_s wherein the MOS gate voltage then becomes zero until the end of the switching period. We have selected a switching period of $T_s = 50\mu s$, corresponding to a switching frequency of 20kHz−a typical value for DC/DC conversion as higher frequencies tend to cause a drop-off in efficiency [46], [47].

The PWM in our design was modeled as a discrete finite-state-machine instead of as a continuous dynamic circuit. The comparator, sawtooth wave generator, and sampler were abstracted away in this manner. Instead, these functions are performed computationally by controlling the duty of the Pulse block included in the Modelica library: based on the sign and magnitude of difference between the sensed output voltage and the target voltage, the PWM will adjust the duty appropriately for the next switching cycle.

III.3.2 Dataflow Composition

The ESMoL tool suite allows model exchange between Simulink and the CyPhyML integration language. Therefore, we can utilize Simulink to build and perform unit testing on the vehicle software controller (VSC) prior to integration with the dynamic circuit model. A Simulink model of the desired control logic is pictured in Figure III.30.

The controller consists of 8 distinct software functions, which are expressed as Simulink blocks. These functional blocks were preexisting in the Simulink library, so instantiating the blocks was a matter of dragging the blocks into the VSC subsystem. Connecting the blocks and thereby specifying the data flow is also performed by dragging the output of one block to the input of the next. Simulink blocks are functional in that the output is a function of the input, which is ideal for modeling the controller's software behavior. On the other hand, Simulink would not be well suited for composing the physical circuit elements since Simulink ports are not bidirectional: They are not as capable at modeling electrical behavior as is the Modelica environment.

The VSC essentially instructs the DC/DC converter whether to operate in boost or buck mode, and also what voltage to regulate at the output terminals. The VSC does this by processing information from the brake pedal, gas pedal, and also the battery voltage sensor.

The software component has three input ports and two output ports. The first input is a two-valued signal which signifies whether or not the vehicle battery is fully charged. This signal will be true if the battery is at full cell voltage and cannot be charged any further at risk of damaging the battery. The controller uses this signal in buck mode operation (ie. Brake_On = TRUE) to determine what voltage to regulate across the battery's terminals. If the battery is fully charged, the controller will instruct the PWM to regulate a voltage which is slightly less than the charging voltage so as to only maintain the cell voltage.

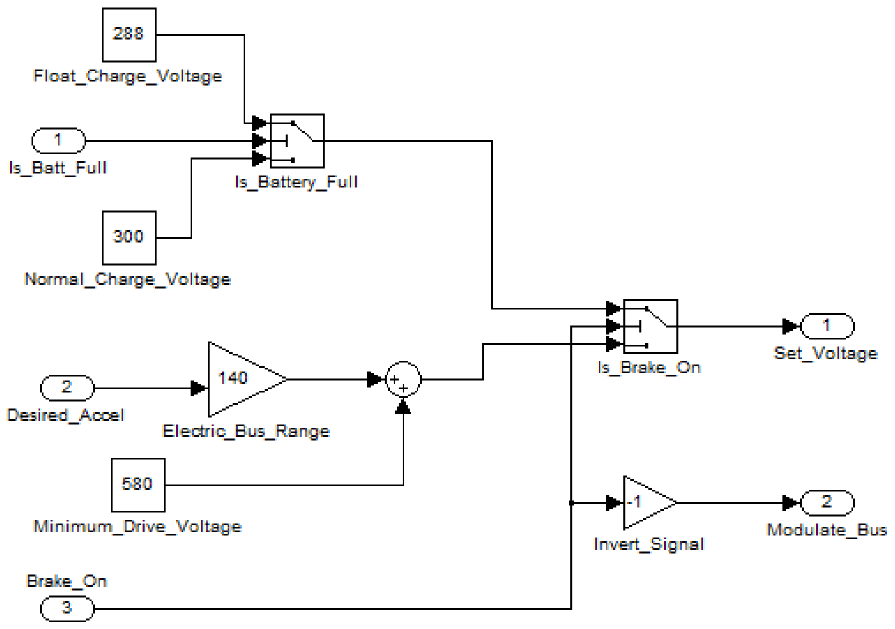


Figure III.30: A possible Simulink configuration of the vehicle software controller.

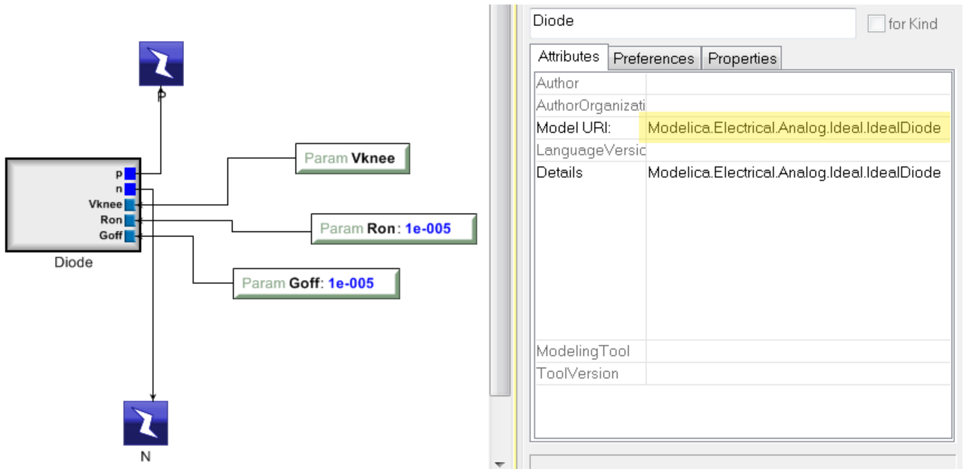
The second input labeled Desired_Accel is governed by the gas pedal and expresses the fractional value of full acceleration that is desired by the driver. Since the safe operating limits of the electric motor are between 580V and 720V, the Desired_Accel signal is multiplied by the operational voltage range, 140V, and then summed with the minimum voltage. The resulting signal becomes 720V when the gas pedal is fully pressed down, and 580V when the minimum acceleration is desired.

The third input, labeled as Brake_On in Figure III.30, is a two-valued signal which indicates whether the vehicle operator has pressed the brake pedal. When the VSC senses that the brake has been pressed, it starts transitioning the DC/DC converter into buck mode so that recuperated braking energy can be used towards regenerating the battery. The output of the VSC provides two signals to the PWM: The VSC (1) instructs the PWM whether to maintain buck mode or boost mode conversion, and (2) tells the PWM at what set point to regulate the converted voltage as determined by the gas pedal and battery sensor. The VSC uses the Modulate_Bus signal to instruct the PWM whether to modulate the bus or the battery (TRUE $\equiv$ Modulate the bus, FALSE $\equiv$ Modulate the battery). The Set_Voltage output determines the target voltage for the current direction of conversion. This value will be 288V or 300V for the battery, and will be between 580V and 720V for the bus.

III.3.3 GME Composition

In order to perform a combined simulation of the vehicle controller and the DC/DC converter, we need to compose these models together within the CyPhyML integration language. Our model-integrated approach utilizes the Graph Modeling Editor (GME) for composing and integrating models from ESMoL languages. We will first focus on integrating the dynamic circuit model designed in Modelica.

Composing the DC/DC converter circuit within GME is relatively straightforward, since GME allows users to instantiate models which reference functional components from the Modelica standard library. To do so, a shell for the Modelica block first needs to be



Diode

Author	
AuthorOrganizati	
Model URI:	Modelica.Electrical.Analog.Ideal.IdealDiode
LanguageVersic	
Details	Modelica.Electrical.Analog.Ideal.IdealDiode
ModelingTool	
ToolVersion	

Figure III.31: The diode Modelica component as embedded into a CyPhyML component. Two power ports and three parameter ports are exposed as interfaces.

created, and then the interfaces and parameters from the Modelica definition need to be instantiated within the shell. Finally, by specifying the Modelica-library location of the block within the “Model URI” attribute field, we establish a link between the GME model and the operational semantics of the Modelica block definition. These steps are shown in Figure III.31. The highlighted portion on the right side of the diagram shows where the Modelica model path is captured. The Modelica model component is shown on the left, and from it we can see that the model already has its ports and parameters created. The two electrical pins are marked *p* and *n* have connections linking them with special CyPhyML electrical power ports. The parameter values for diode resistance and conductance are shown connected to parameter atoms which allow these variables to be exposed at higher levels in the hierarchy. The electrical power ports are also exposed and function as interfaces from higher levels in the hierarchy.

The DC/DC Converter circuit consists entirely of Modelica blocks, so repeating this process for all components would create the Modelica components within the GME environment. The visualized Modelica diode in Figure III.31 resides within a CyPhyML component. CyPhyML components can contain Modelica primitives, or alternatively Signal Flow functions. The vehicle controller will exist entirely within one CyPhyML component and will be a compound of Signal Flow primitives.

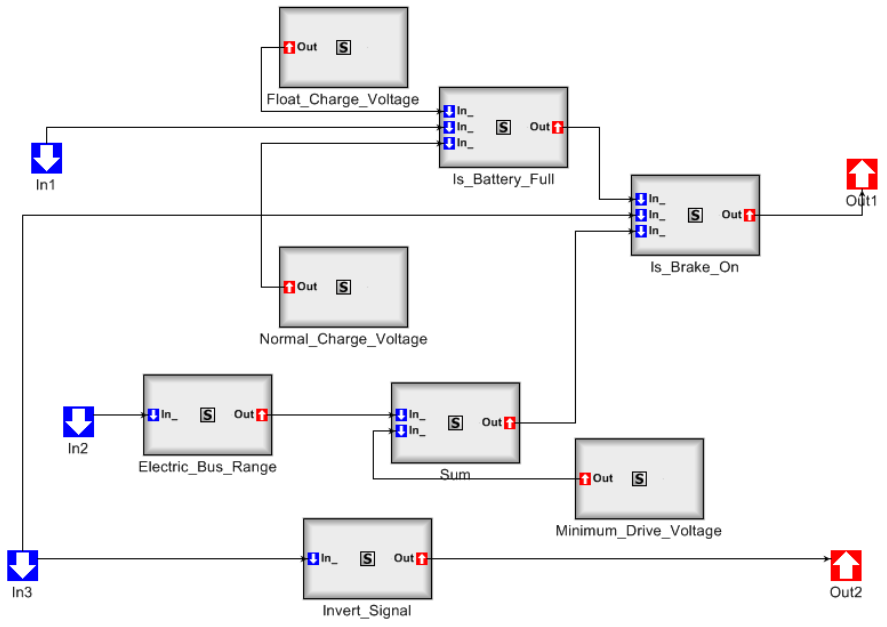


Figure III.32: The vehicle system controller design realized from Signal Flow primitives within GME

Importing the Simulink-created vehicle controller into GME is different from importing any Modelica block. The process is automated with the help of the MDL2MGA function which automatically transforms a Simulink file into a GME's *mga* format [4]. It does so by generating a replica of the Simulink model from Signal Flow blocks, the native discrete-event DSML. The Signal Flow subsystem will be used to specify the function of the vehicle controller within CyPhyML. An example of a possible controller configuration is displayed in Figure III.32.

Within GME, the vehicle controller is comprised solely of Signal Flow blocks. However, the Signal Flow representation serves as a nearly identical model of the original Simulink controller. This is due to the fact that both languages rely on canonical, C-based math functions for their operation during simulation and deployment. The controller in GME is can be arranged to appear similar to the Simulink version since both representations rely on the same functional blocks. Within the GME model $In1$ corresponds to the Is_Battery_Full signal, $In2$ corresponds to the Desired_Accel input, and $In3$ to the Brake_On

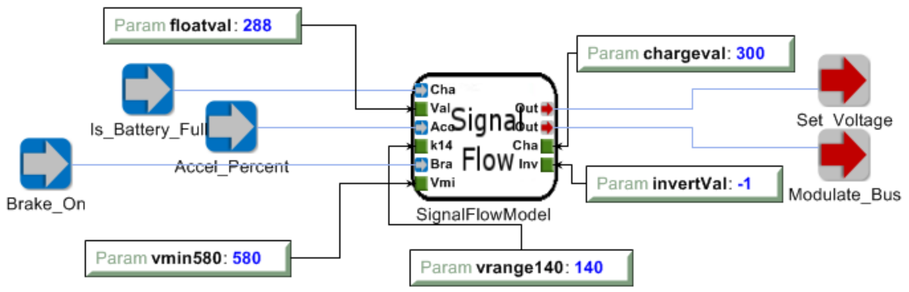


Figure III.33: The Signal Flow controller embedded within a CyPhyML component. Numerous causal signal ports and parameter assignments function as component interfaces.

signal. With the controller design implemented within GME, the output signals of the controller can then be used to command the DC/DC converter. Figure III.33 depicts the same controller from one level up, in the component level hierarchy where only the parameters, input ports and output ports are exposed.

At this point, the DC/DC converter components exist within the CyPhyML framework but their relations are not yet established. Before integrating the vehicle controller, the DC/DC converter circuit needs to be composed within GME from its Modelica components. This composition is simple once all the Modelica circuit components have been created within GME. Up to this point we have only shown illustrations of the constructs contained within a CyPhyML component (ie. Figures III.33 and III.29). We will now manipulate the CyPhyML components from a higher level in the hierarchy: from within component assemblies. Within a component assembly GME depicts components as blocks with parameter, signal, and power ports exposed, and marks all components with a puzzle-piece icon.

The DC/DC component assembly is pictured in Figure III.34 and contains multiple components representing the capacitors, inductor, NMOS switches, diodes, and voltage sources. These components are the same components that we used when drafting the DC/DC converter model in Modelica, and are associated using the same connections between components as in the Modelica model. In GME connections are established by clicking once on each endpoint of the desired connection. Furthermore, assignment connections are established in the same manner, and these connections allow parameter values to be passed down into the components from upper levels (ie. the Testbench) of the model hierarchy.

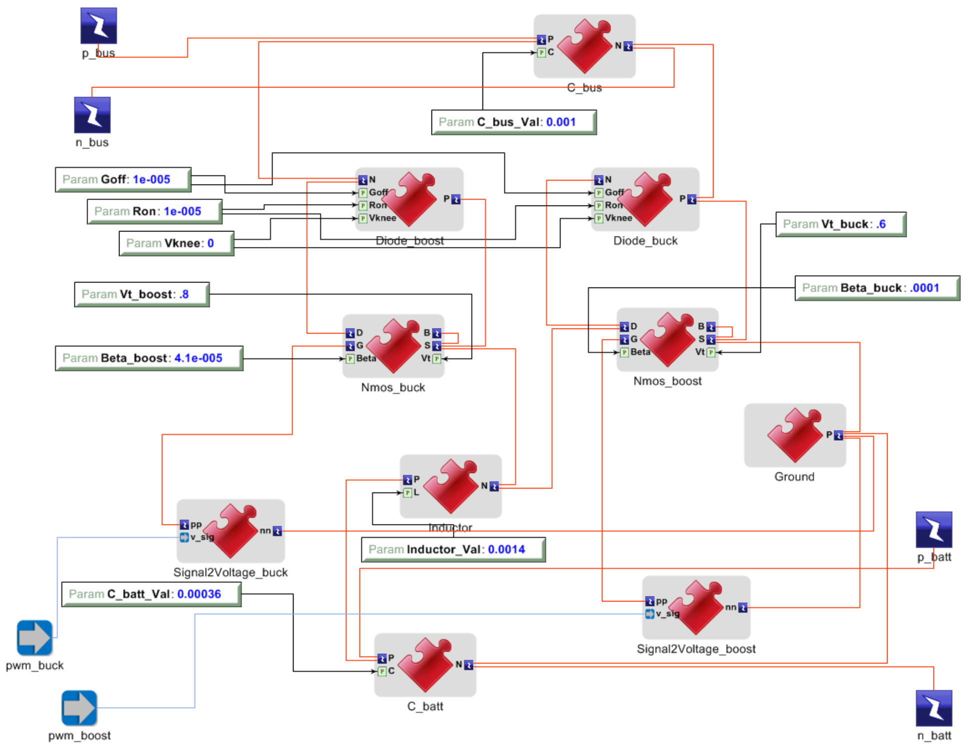


Figure III.34: The converter circuit component assembly within GME. The circuit is composed of numerous CyPhyML components which contain embedded Modelica model references.

In composing this component assembly, we have established a DC/DC converter model within GME that is semantically identical to the Modelica model, yet the syntax is different between modeling suites. We can observe in Figure III.34 the PWM signal ports, as well as the power ports to the bus and battery. The next step in the modeling process is to combine the component assembly of the DC/DC converter with the PWM, the mode-set (MS), the bus, the battery, the voltage sensors, and most importantly the vehicle controller, into one all-encompassing component assembly.

Corresponding to the two input signal ports on the DC/DC converter, two output signals generated by the PWM which control the high frequency switching, need to be connected

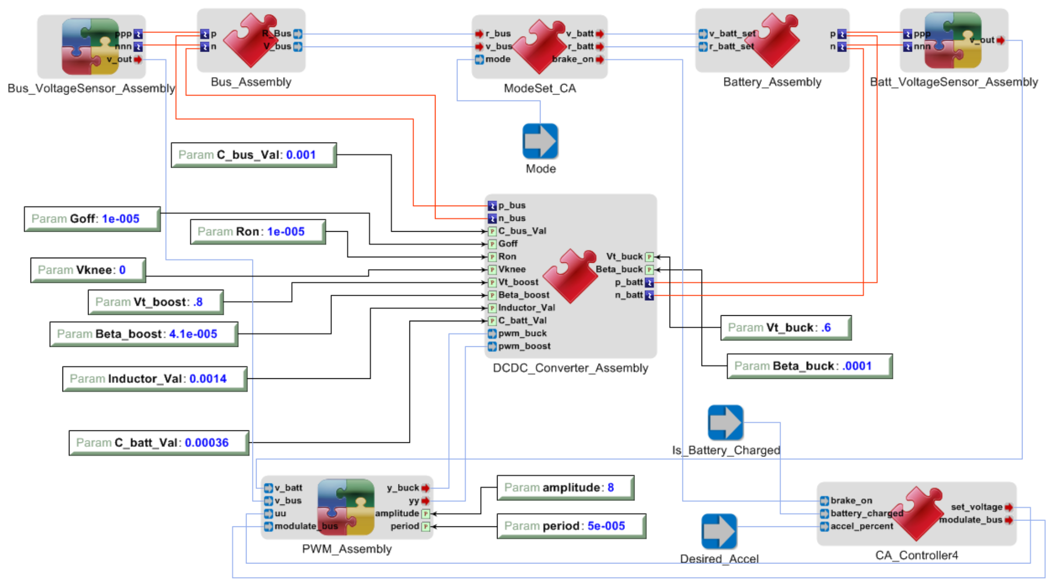


Figure III.35: The top-level DC/DC converter component assembly within GME, incorporating the converter circuit, bus, battery, voltage sensors, mode-set, PWM, and vehicle software controller.

to the DC/DC converter. The required connections can be seen in Figure III.35 which portrays the final component assembly incorporating all model components. Signals from the voltage sensor of the bus and battery both flow back into the PWM allowing it regulate voltage with a feedback loop. The PWM is shown to receive the set voltage in a signal from the VSC, as well as a signal specifying buck or boost mode.

The input signals to the VSC will be custom commands that outline a transitional event within the converter. An example transition could be a change from regenerative braking to forward drive, or from battery charging to float charging. A similar input signal to the mode-set (MS) component helps to precisely specify a transitional event. The MS allows the bus and battery to synchronously behave according to the desired operating conditions. For example one mode specifies the conditions of high engine load, another specifies low load, and a third mode engages the brakes. The bus and battery are modeled as independent components with their characteristics governed by the MS, and their electric behavior determined by the power flow of the DC/DC converter.

In this final assembly we have combined all elements of the DC/DC converter−from computational control elements to dynamic physical elements−into a single model where control signals from the controller can influence the electrical behavior of the converter circuit. In doing so we have unified controller logic and physical dynamics within a single framework, and will be able to test different operating conditions on the DC/DC converter as a whole with the help of GME's testbench composition functionality. We will be able to make reasonable assumptions about the behavior of the modeled DC/DC converter, and be able to perform formal verification techniques. The three exposed signals of *Mode*, *Is_Battery_Charged*, and *Desired_Accel* will allow us to design simulations that evoke particular transitions in the converter's operation.

CHAPTER IV

Evaluation

As part of our evaluation of the DC/DC converter, we will briefly describe how to use GME to compose a testbench for a specific use case.

IV.1 Testbench Composition

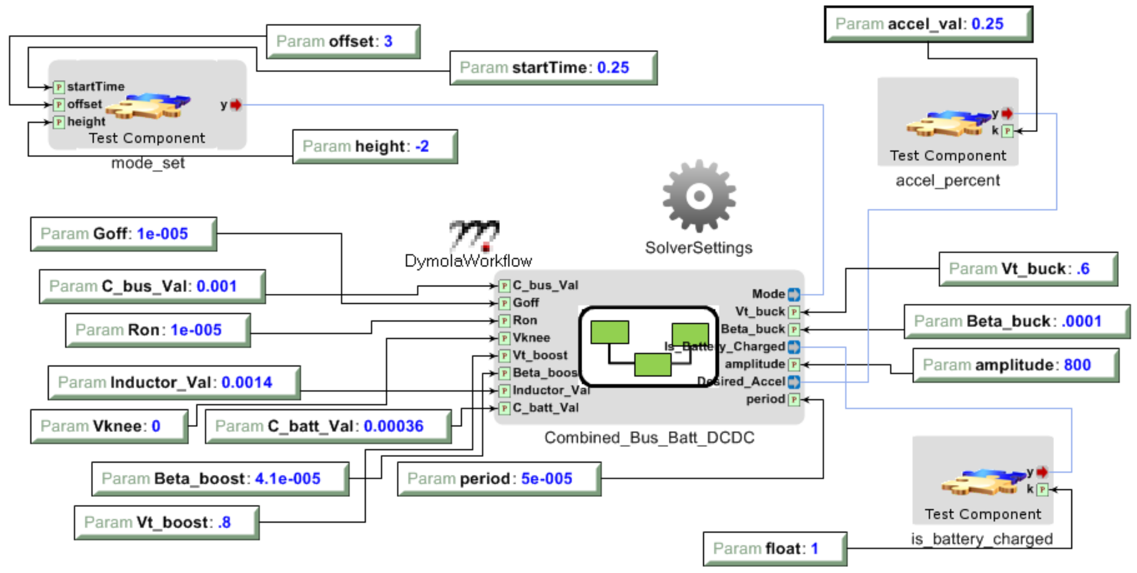


Figure IV.1: The full DC/DC converter model instantiated within a testbench. The three test components provide external stimuli to the converter model throughout the simulation window. This testbench describes a transition from float charge operation to hybrid-drive.

Using GME we can specify desired input stimuli to the DC/DC converter assembly within a testbench, and then observe the physical consequences of such stimuli as experienced by the converter. A testbench is created within GME and incorporates the component assembly as the system under test. Device parameters are ultimately configured from the testbench editing window. GME does not perform the simulation itself, and instead the testbench is exported in the Modelica language format and simulated by the Modelica editor.

A testbench for the DC/DC converter specifying a transition from regenerative braking to forward drive is pictured in Figure IV.1, however, this same general structure is employed for all simulations. Aside from the visible parameter assignments, we can also witness three distinct test components which each command input to the DC/DC converter. Starting in the bottom-right corner of Figure IV.1 we observe that the signal corresponding to Is_Battery_Charged is fixed at a positive value. This input would be positively valued only if the battery were fully charged, in which case the float charge will be applied to the battery if the vehicle enters regenerative braking. Next in the top right of Figure IV.1 we are able to see the Accel_Percent test component, which ultimately determines the level of voltage to be regulated on the bus during forward drive. Its constant value of 0.25 corresponds to a near minimum value of regulation, and translates to a target voltage of about 615 volts. Finally, the test component for setting the mode of the DC/DC converter is located in the top-left of Figure IV.1, and is modeled by a step function. The signal starts in mode 3−as specified by the offset−and then drops to mode 1 at time = 0.25s. The duration for all simulations is 0.5 seconds, so the mode transition should occur halfway through the simulation window. Based on the configuration of the testbench the DC/DC converter will start operation within regenerative braking, and then transition to hybrid drive halfway through the simulation. The following section analyzes the resulting simulation generated from this testbench with a focus on the voltage response of the battery and electric bus.

IV.2 Simulation Evaluation

IV.2.1 Float-charge transition to hybrid-drive

From the voltage response graph of Figure IV.2 we observe the transient response of the bus voltage (top/red) and the battery voltage (bottom/blue) as simulated from the testbench model. We see that the bus is initially generating a relatively steady voltage because braking is engaged. It takes about 0.1 seconds then for the DC/DC converter to regulate a steady voltage across the terminals of the battery: the applied battery voltage is initially zero, then

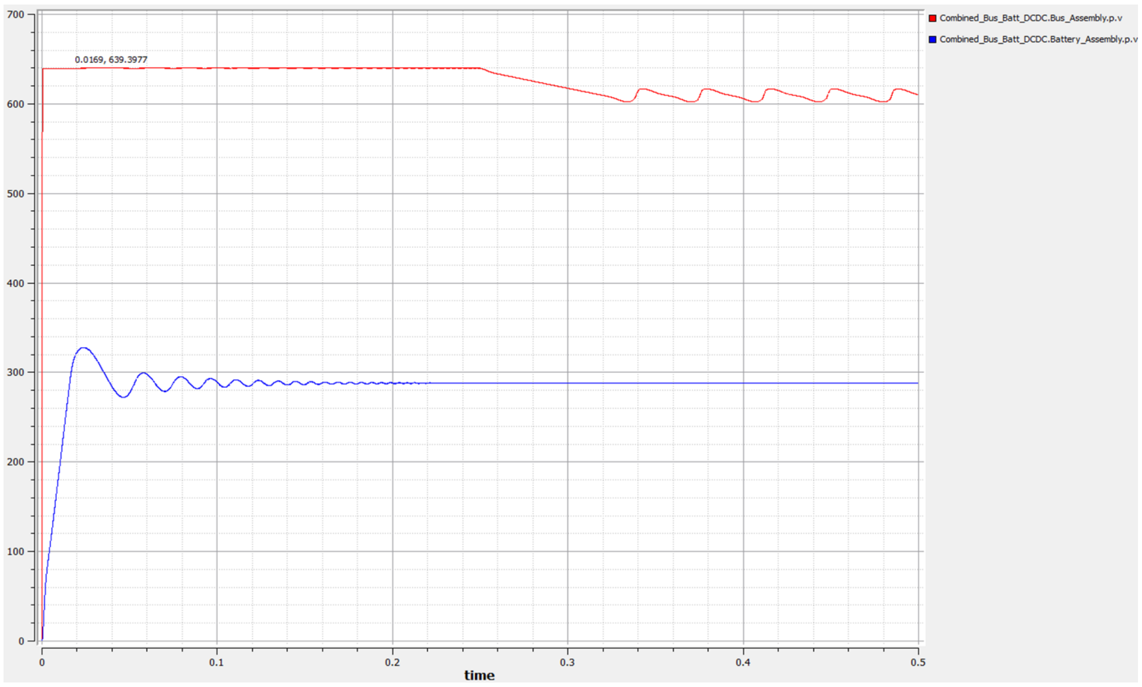


Figure IV.2: Float-charge transition to hybrid-drive: Pictured is the voltage response of the bus (top/red), and battery (bottom/blue) during the dynamic simulation of this use case. This simulation corresponds to the testbench in Figure IV.1.

rises and overshoots the desired level by about 40 volts before settling at around 288V. The ripple is less than 1 volt after about 0.2 seconds. The normal battery charging voltage should be about 300V, and because the plotted voltage is slightly less than the full battery cell voltage we gather from the graphs that the controller is actively applying the float-charge voltage across the battery. This indicates that the controller has correctly processed the Is_Battery_Full signal and is appropriately regulating voltage across the battery via the DC/DC converter.

At time = 0.25s, the converter switches from braking to forward drive prompted by engaging the gas pedal. The controller detects that the brake is no longer pressed via the Brake_On signal, and then commands the DC/DC converter to transition from buck mode, to boost mode. The DC/DC converter firmly closes the buck-NMOS and reroutes the PWM signal to the boost-NMOS thereby changing the direction of the converter. The battery's cell voltage is boosted to about 615V by the converter: this value was specified by the

Desired_Accel input (corresponding to the gas pedal), and is regulated at the output of the converter by the vehicle controller via the PWM. Since we have selected a minimal inductance value for the inductor, the transition delay is mostly due to the large capacitor across the bus terminals which needs to discharge by 30V after exiting braking operation. Still the steady-state voltage across the bus is achieved in less than 0.1 seconds, at 35.1ms. The small inductor−relative to the load resistance−is ultimately responsible for the nontrivial ripple at the output during steady-state of 13.92V. The inductor is able to change current so quickly that the PWM is unable to maintain single-digit volt precision given its sample rate. This ripple would be more of a concern if it were to occur within the response of the battery voltage, since the ripple could damage the battery. However, the ripple falls within the acceptable voltage range of the motor at the bus terminals, and the magnitude of the ripple is inversely proportional to the response time of the converter.

IV.2.2 Hybrid-drive transition to load drop-off

For a second illustrative example of testbench composition we examine a use case where the road load abruptly drops-off, corresponding to a scenario where the hybrid vehicle reaches the crest of a hill. This testbench, pictured in Figure IV.3, is nearly identical to the testbench of the previous scenario but with a few changes.

The Is_Battery_Full signal is not used in this scenario because the vehicle remains in forward boost mode before and after the mode transition. As the mode changes from 1 to 2 at time = 0.25s, the resistance of the bus drops from 1960Ω to 500Ω. As specified by the Accel_Percent signal the bus voltage will be maintained at 615V throughout the simulation, but the conversion ratio will change as the resistance of the load drops-off. This scenario will allow us to witness the changing dynamics of the DC/DC converter as it experiences the load drop-off. The simulated response of the battery and bus voltages are pictured in Figure IV.4.

There is a visible positive change in the magnitude of the steady-state ripple once the

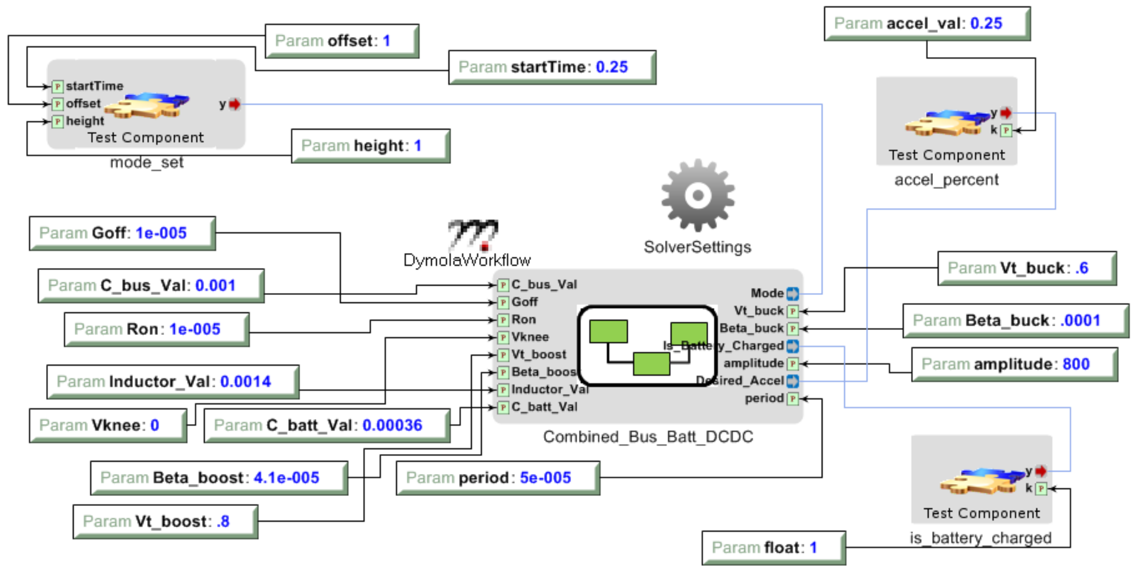


Figure IV.3: Testbench of the full DC/DC converter model characterizing a mode transition between hybrid-drive transition and load drop-off. The transition is invoked by varying the Mode signal input at time = 0.25s, which commands a drop-off in bus load resistance.

load of the road drops-off at time = 0.25s. The widening of the ripple (from approximately 15V to 22.98V) is largely due to the fact that the conversion ratio for the circuit changes as a result of the load change. Moreover, due to the shape of the conversion ratio function, the PWM can maintain higher precision when the duty ratio is lower because a change in duty translates to a lesser change in output voltage while in the lower duty range. The reason that the conversion ratio changes is because the load-dependent K value affects the shape of the DCM conversion ratio function. Previously, we determined that the K value for the converter with the full 1960Ω load is equal to about 0.0286. Now if we substitute the new resistance of 500Ω into the K equation in (III.51), we get a K value of

$$K = \frac{2L}{RT_s} = \frac{2(1.4mH)}{(500\Omega)(50us)} = 0.112 \tag{IV.1}$$

If we substitute this higher K value into the boost mode DCM conversion ratio function and compare it with the conversion ratio associated with the old K value, we see that the same duty ratio is no longer sufficient in boosting the battery to 615V. With a high load, and K =

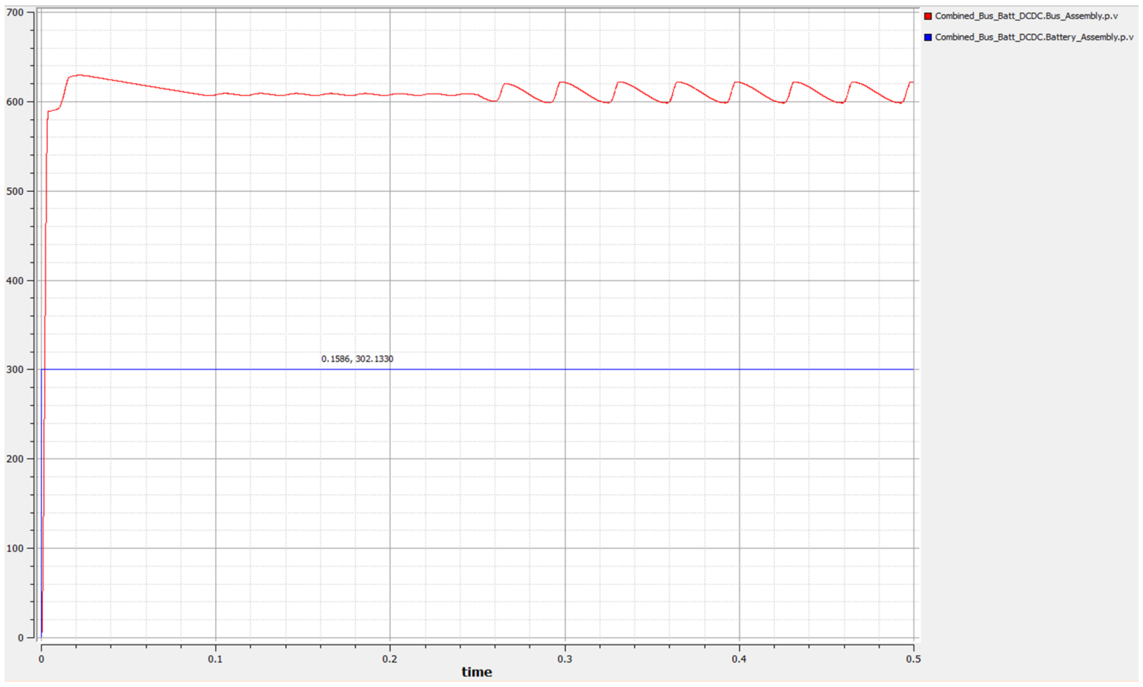


Figure IV.4: Hybrid-drive transition to load drop-off: The bus and battery voltage response. This corresponds to the testbench in Figure IV.3. Varying the load alters the conversion ratio and steady-state ripple as visible in the second half of the simulation window.

0.286 we require a duty ratio of

$$M(D,K) = \frac{1+\sqrt{1+4D^2/K}}{2} = \frac{V_{bus}}{V_{batt}} \tag{IV.2}$$

$$M(D,K) = \frac{1+\sqrt{1+4D^2/(0.0286)}}{2} = \frac{615V}{300V} \tag{IV.3}$$

$$D = 0.248 \tag{IV.4}$$

Now with the load dropped-off and a K value of 0.112, the appropriate duty ratio to maintain 615V is given by,

$$M(D,K) = \frac{1+\sqrt{1+4D^2/K}}{2} = \frac{V_{bus}}{V_{batt}} \tag{IV.5}$$

$$M(D,K) = \frac{1+\sqrt{1+4D^2/(0.112)}}{2} = \frac{615V}{300V} \tag{IV.6}$$

$$D = 0.491 \tag{IV.7}$$

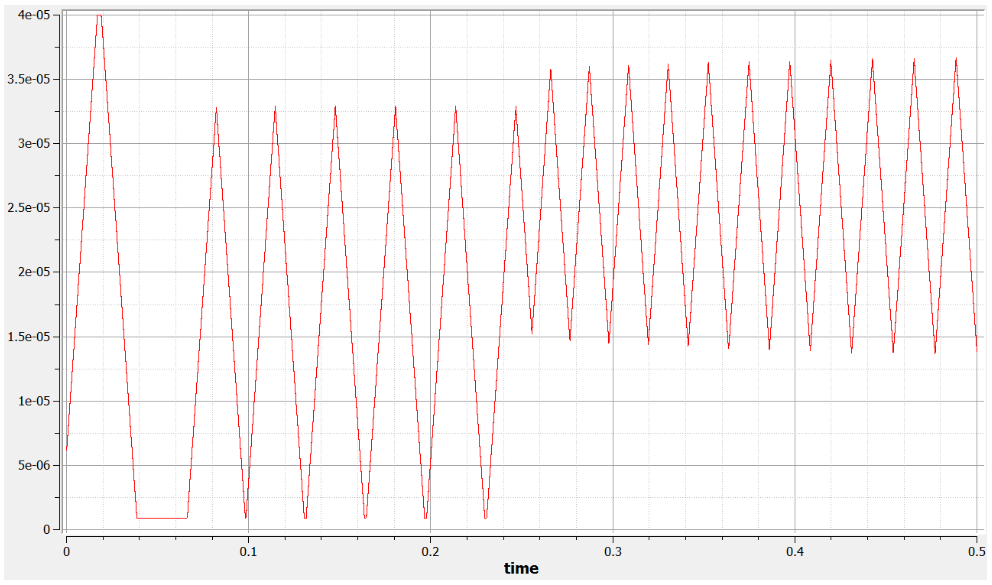


Figure IV.5: Plot of PWM pulse width ($D * T_s$) during a transition from hybrid-drive to load drop-off. A higher duty ratio is witnessed after the transition, even though the target voltage remains the same. The load drop-off causes a change in the converter's conversion ratio.

The result of these equations tells us that during the load drop-off transition, the PWM will maintain a higher duty in order to regulate the same voltage output. At this higher duty ratio, the PWM is more granular and creates a larger ripple in the bus voltage. Figure IV.5 shows the value of the width of the first interval switching over time for this load drop-off simulation. The corresponding duty ratio can be calculated by dividing this width by the sample period ($T_s = 50us$). We must recognize that the duty ratio is comprised of the DC duty ratio, as well as the small signal duty ripple, represented as

$$\widetilde{D} = D + \widehat{d} \tag{IV.8}$$

For simpliity, we will focus on the large-signal value of D.

The large-signal value of D is the DC average of the complete duty signal. In the first half of the simulation, we calculate D to be about 0.29. This is approximately equal to the expected D value of 0.248. In the second half of the simulation D is about equal to 0.50, which corresponds to the expected duty value of 0.491.

This plot of interval width tells us that (1) the load drop-off scenario effectively alters the conversion ratio of the DC/DC converter and, therefore, the voltage response is altered, and also that (2) our DC/DC converter model is highly accurate and its exhibited duty ratio closely matches the duty ratio calculated from the DCM boost mode equations.

IV.2.3 Battery charging transition to float charging

The next use case that we will simulate and evaluate is the transition to trickle or float charging from normal battery charging. This case occurs when either the brakes or the diesel engine has charged the battery to maximum voltage, after which a lower voltage must be maintained across the battery's terminals to avoid overcharging. The testbench for this scenario maintains the DC/DC converter in regenerative braking, and forces a transition in the Is_Battery_Full signal at time = 0.25s. The plot of this simulation is pictured in Figure IV.6.

If we observe the red voltage signal of the bus, we can see that constant voltage is being generated on the bus. The DC/DC converter is using this source voltage to regulate the charging voltage across the battery which is represented as the blue signal in the plot. Like with the first simulation, the response time of the battery voltage is less than 0.1 seconds, at 32.8ms. Initially the 300V charging voltage is maintained until the transition where then the target voltage lowers to 288V to achieve float charging operation. This transition has a markedly quick rise time of 6.2ms, which is possible because the voltage only experiences a change of 12 volts compared to the initial change of 300 volts.

This use case is very common for the hybrid vehicle and will occur every time that the battery becomes charged to full capacity. The quick response time prevents excessive overcharging once the sensor has detected that the battery has reached maximum voltage.

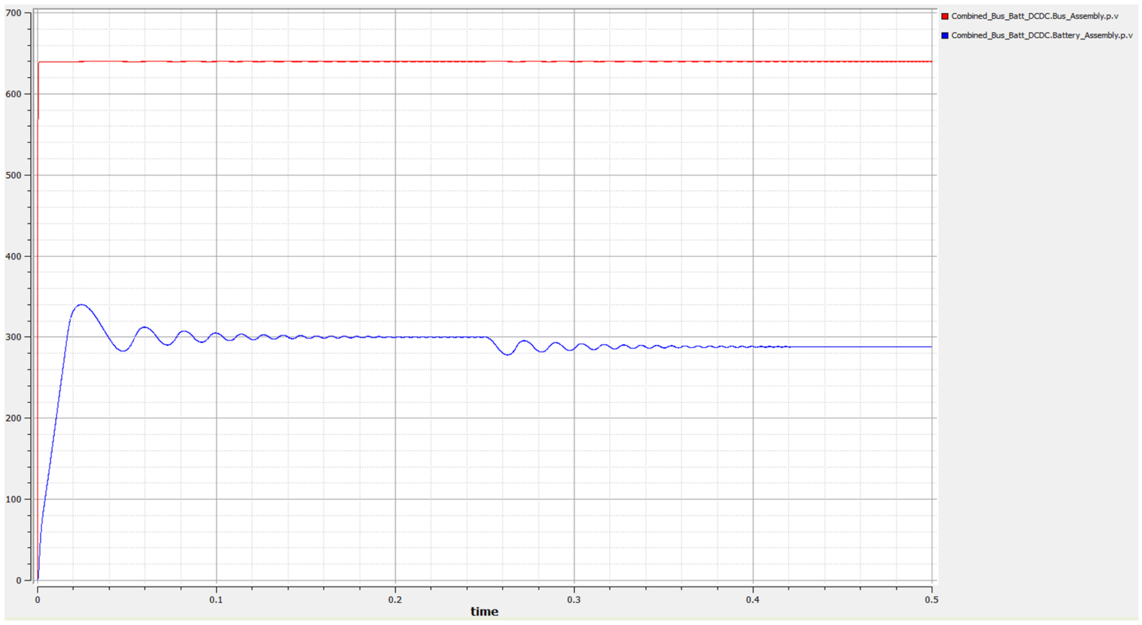


Figure IV.6: Battery charging transition to float charging: Voltage response of the bus and battery. Once the controller detects a fully charged battery, a float charge is regulated across the battery to avoid overcharging.

This quick response is made possible by the small inductor within the DC/DC converter. The transition to float charging does not create any unintended spikes in voltage across the battery which is important for preserving the operating life of the battery cells.

IV.2.4 Hybrid drive transition to battery-only drive

Figure IV.7 depicts the voltage response of a simulation where the vehicle transitions from diesel assisted hybrid-drive into battery-only drive. When the battery is working by itself, it needs to provide more power to the motor compared to hybrid-drive where the battery works in conjunction with the diesel generator. Halfway through the simulation the voltage converted from the battery is boosted to a much higher 720 volts, from its initial 580 volts. The higher voltage mode, therefore, corresponds to the battery-only drive mode of operation.

In the graph of Figure IV.7, the bus voltage takes about 0.1 seconds to transition to the high-voltage battery-only drive. The settling time is equal to 87.9ms. The voltage surpasses

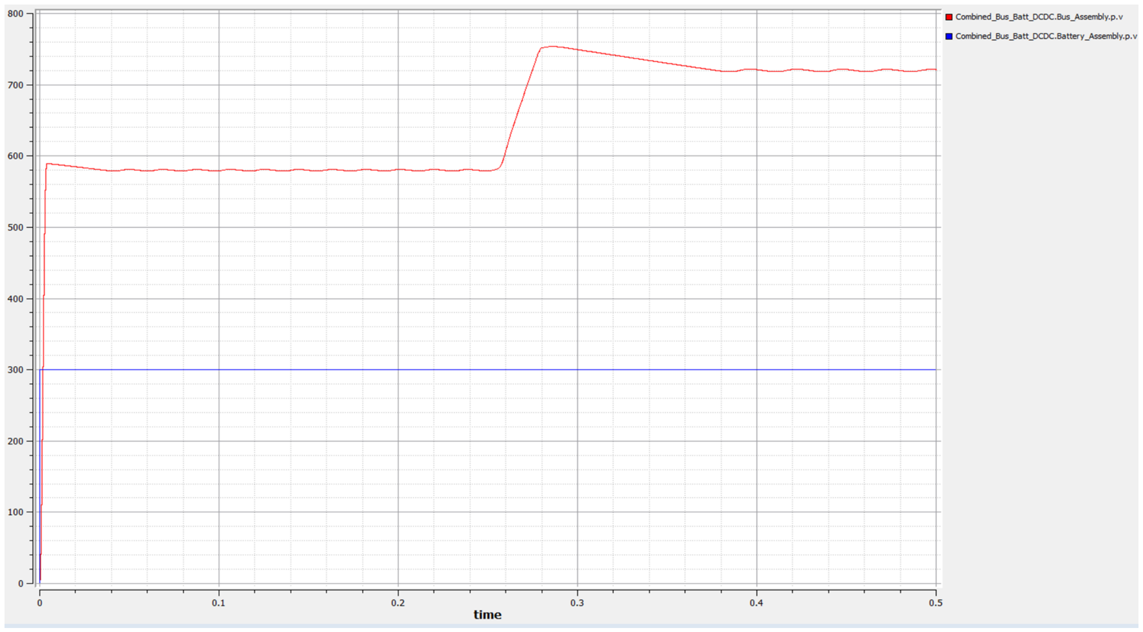


Figure IV.7: Hybrid drive transition to battery-only drive: Plot of bus and battery voltage during this use case.

the target voltage at only 23.9ms, but then takes much longer to reach equilibrium at the target level. The bus voltage is boosted an additional 140 volts during this transition, and the voltage overshoot is responsible for the extended settling time compared to the rise time in this scenario.

In boost mode operation it is apparent that a positive voltage transition occurs much quicker than a negative voltage transition. One cause of this is the fact that the duty ratio is bandlimited by the PWM at $D = 0.02$ ($D = 1\mu s$), and never goes to zero. A lower duty ratio corresponds to a lower bus voltage as specified by the conversion ratio. If the duty were able to temporarily drop to zero then only the bus capacitor would govern the drop in bus voltage, with no additional and unwanted current provided by the converter leading to a quicker transition. With positive voltage adjustments, the duty ratio never reaches the upper limit of $D = 0.98$ ($D = 49\mu s$) because the maximum desired bus voltage still corresponds to the lower range of possible duty ratio values. Very modest changes in duty ratio bring the boost operation into CCM from DCM which leads to large positive voltage adjustments.

The inability of the DC/DC converter to completely shut-off is what causes slow negative adjustments in bus voltage. This behavior is exhibited to a greater degree in the opposite transition from battery-only drive to hybrid-drive. What we learn from this unideal behavior is that it would be better to design a PWM that is linear with respect to output voltage, rather than linear with respect to duty ratio. Additionally, this behavior indicates to us that a smaller value for the C_{bus} would be more ideal in order to incite a quicker change in bus voltage during transitions.

IV.2.5 Battery-only drive transition to hybrid-drive

In this scenario the vehicle transitions from high conversion, battery-only drive to hybrid-drive where the diesel engine begins assisting at time = 0.25s. After the diesel generator starts providing voltage to the bus, less power is required from the battery to propel the car with the same acceleration. The simulation results of this scenario are pictured in Figure IV.8.

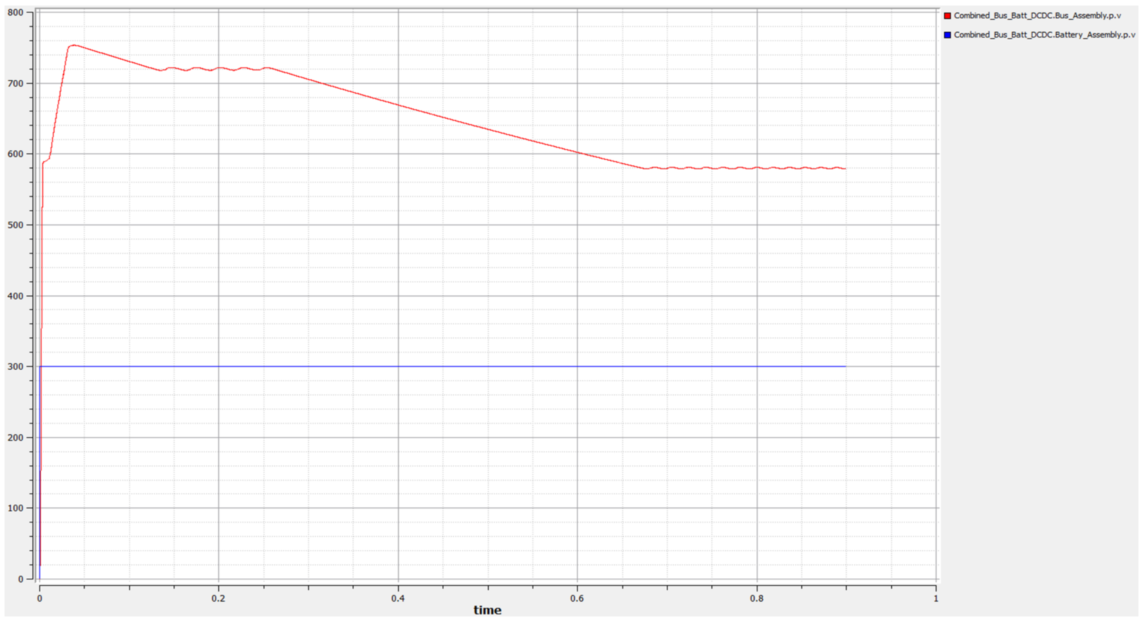


Figure IV.8: Battery-only drive transition to hybrid-drive: plot of bus and battery voltage during this use case. This is the only simulation with a window larger than 0.5 seconds. An area for converter improvement is hastening this negative voltage adjustment during boost mode operation.

Immediately apparent from the results of this simulation is that the simulation window has been extended to 0.9s. This is because the response of the DC/DC converter takes longer than the remaining half of the 0.5 second window, with a fall time equal to 0.4230 seconds−by far the largest. This slow negative voltage adjustment can be attributed to the fact that our PWM does not allow a duty ratio of zero, which would bring about the fastest possible voltage reduction. On the other hand, this transition is not so slow in an absolute sense since this use case covers a transition from the highest possible drive voltage to the lowest possible drive voltage. However, compared to the opposing transition of hybrid-drive to battery-only drive this transition is markedly slower and remains an area for design improvement.

IV.2.6 Hybrid-drive transition to battery charging

The next scenario covers a transition from forward hybrid-drive to regenerative braking. This scenario requires that the DC/DC converter switch from boost operation to buck operation halfway through the simulation window. The voltage response of this simulation is shown in Figure IV.9. Initially, voltage from the battery (blue) is boosted to deliver 650V across the bus terminals. After the transition, voltage becomes generated by the bus relative to the intensity of braking, and this excess voltage is used towards charging the battery. To accomplish the charging, the DC/DC converter bucks the high bus voltage down to an appropriate charging level for the battery at 300V. The switching event creates slight turbulence in the battery response; however, the oscillations do not cause the signal to ripple more than 2% of the target value.

Since this transition encompasses a switch from boost mode to buck mode, the inductor current must switch from positive to negative over the course of the transition. A large inductor would hamper the transition time in this scenario by forcing a gradual transition of voltage. Therefore, a small inductor is beneficial towards improving the response time when the converter needs to switch directions. Also helpful to this effect is the fact that the converter operates in DCM for a majority of both buck and boost mode. If the inductor

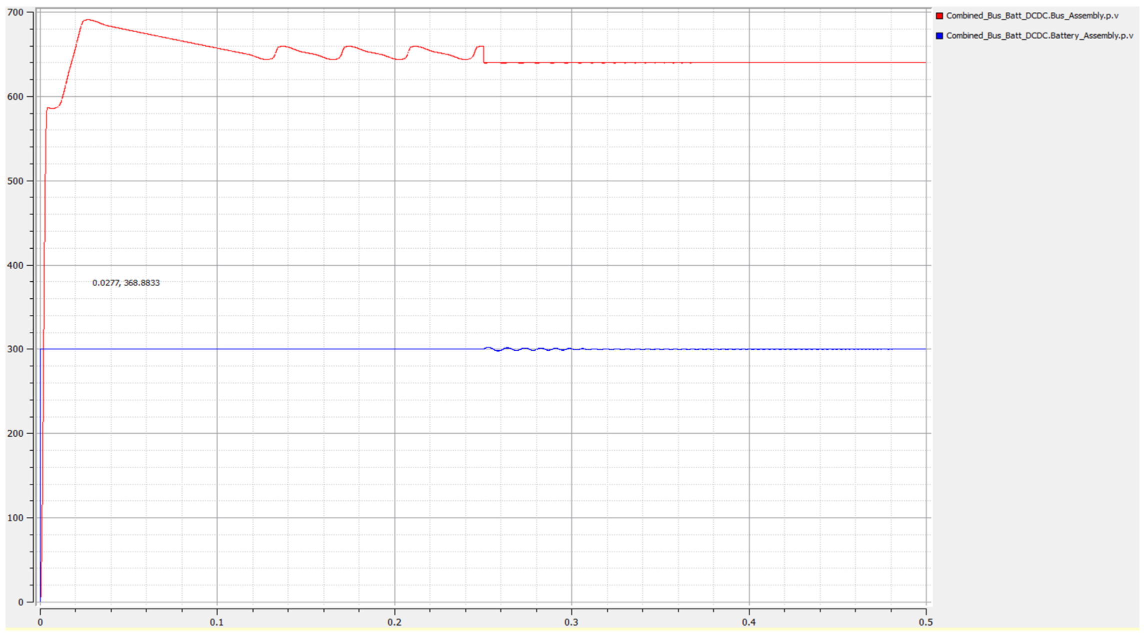


Figure IV.9: Hybrid-drive transition to battery charging: plot of bus and battery voltage.

current is zero at the time of a direction reversal, then little to no time is devoted towards transitioning the inductor current to the opposite direction. However, if boost mode had a high DC current in CCM, then it would create a nontrivial delay in order to reduce the current to zero, before switching to the buck mode DC value. A plot of the inductor current for this testbench is shown in Figure IV.10.

In the plot of inductor current we observe that the current is positive during boost mode, but frequently drops to zero during switching intervals. If we were to zoom closer, the solid areas on the plot would more accurately resemble quick transitions between the peak current and zero current. When the current drops to zero, the converter is operating in DCM. There are small windows of time wherein the inductor current does not drop to zero, indicating brief periods of CCM. These small windows of CCM coincide with steep rises in the regulated output voltage. If we look on the right side of the plot, we can witness the inductor ripple while in buck mode. Unlike boost mode, the inductor current stays mostly negative and never becomes positive. The buck mode current always drops to zero during each cycle, which means that the buck mode conversion is always in DCM unlike boost

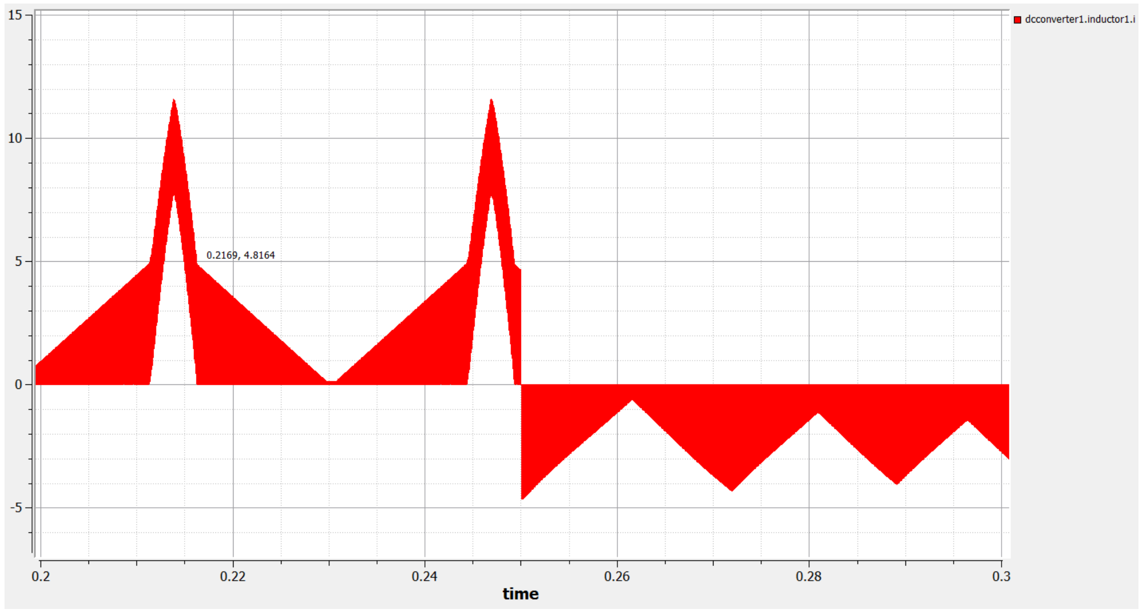


Figure IV.10: Plot of inductor voltage during a hybrid-drive transition to battery charging. The current is positive during boost mode, and negative during buck mode. The circuit operates mostly in DCM, with brief periods of CCM operation during boost mode.

operation.

If we focus on time = 0.25s on the plot where the switching event occurs, we see that the inductor experiences no delay in transitioning from boost mode to buck mode. The current is already at, or very close to zero amps so that no time intensive current swing is necessary to switch the operation of the DC/DC converter. If the transition had occurred during a period of CCM, a nontrivial period of change would be required so that the direction of inductor current could become reversed. However, the inductance in our inductor component is so small that this transition of a few amps occurs almost immediately. Still the fact that the converter operates mostly in DCM allows for the swiftest possible transitions from buck mode and boost mode, and vice versa.

IV.3 Simulation Results

We have organized, in Table IV.1 the results of 20 operating mode transitions to and from the five distinct states of the DC/DC converter system. These simulations cover all the possible mode transitions for the system model. For a given transition we list data pertaining to the regulated output of the DC/DC converter. So for a transition to braking or bucking, the signal attributes of V_{batt} are measured. Likewise, for a transition to forward boost mode, the attributes of V_{bus} are measured. The way that the table is organized is so that the vertical column determines the simulation's initial state, and the horizontal rows determine the simulation's final state.

If we look in Table IV.1 where the vertical Hybrid-Drive column intersects with the Load Drop-off column, then we see the simulation results for the results of the transition from Hybrid-Drive to Load Drop-off. In this transition, the set voltage does not change between states which is evident because $\triangle V_{bus}$ is zero, and also $t_{bus,\ rise}$ is zero . Rather, the controller maintains the same voltage level while enduring an increased ripple in the output due to the reduced resistance of the load. The ripple in the final Load Drop-off state is quite significant at 22.98V. This is much higher than a typical Hybrid-Drive ripple of about 14V. To get a sense for the typical Hybrid-Drive ripple, we examine the entire row of transitions ending in Hybrid-Drive. This row shows all transitions that terminate in Hybrid-Drive mode, therefore, this row shows the steady-state ripple inherent in Hybrid Drive.

The transitions from Battery Charging and Float Charging to Hybrid-Drive undergo indentical voltage adjustments, and maintain relatively similar transient behavior with modest ripple and quick settling time. Because $t_{bus,\ rise}$ is greater than $t_{bus,\ settle}$ in these transitions, we gather that the converter output is highly damped in Hybrid-Drive mode. This implies that the overshoot does not surpass 2% of the target value, and instead the output voltage has settled before reaching the full 0-100% rise.

The Battery-Only transition to Hybrid-Drive incurs a dramatic drop in regulated voltage of -140V. These leads to the longest rise and settling times of all transitions, at 0.4230 and 0.384 seconds respectively. Now if we examine the Battery-Only transition to Load Drop-off, we see another -140V voltage adjustment, yet the rise and fall times are significantly

Table IV.1: Voltage response of the DC/DC converter during operating mode transitions

Initial Mode $\Longrightarrow$		Hybrid-Drive	Load Drop-off	Battery Charging	Float Charging	Battery-Only
	$\triangle V_{bus}$	-	0	-30	-30	-140
	$V_{bus,\ overshoot}$	-	8.15	6.90	7.04	1.46
Hybrid-Drive	$V_{bus,\ ripple}$	-	15.56	13.79	13.92	2.92
	$t_{bus,\ rise}$	-	0	0.0679	0.0653	0.4230
	$t_{bus,\ settle}$	-	0.197	0.0364	0.0351	0.384
	$\triangle V_{bus}$	0	-	-30	-30	-140
	$V_{bus,\ overshoot}$	12.14	-	16.55	12.07	9.58
Load Drop-off	$V_{bus,\ ripple}$	22.98	-	33.09	24.13	20.85
	$t_{bus,\ rise}$	0	-	0.0203	0.0249	0.1128
	$t_{bus,\ settle}$	0	-	0	0	0.0848
	$\triangle V_{batt}$	0	0	-	12	0
	$V_{batt,\ overshoot}$	0	0.10	-	10.10	12.07
Battery Charging	$V_{batt,\ ripple}$	1.48	0.14	-	2.22	1.22
	$t_{batt,\ rise}$	0	0	-	0.0062	0
	$t_{batt,\ settle}$	0	0	-	0.0305	0.0427
	$\triangle V_{batt}$	-12	-12	-12	-	-12
	$V_{batt,\ overshoot}$	9.62	9.66	9.66	-	10.41
Float Charging	$V_{batt,\ ripple}$	1.76	1.79	2.03	-	1.79
	$t_{batt,\ rise}$	0.0104	0.0051	0.0075	-	0.0033
	$t_{batt,\ settle}$	0.0820	0.0677	0.0328	-	0.0443
	$\triangle V_{bus}$	140	140	80	80	-
	$V_{bus,\ overshoot}$	51.20	72.33	32.8	26.65	-
Battery-Only	$V_{bus,\ ripple}$	3.92	46.7	4.03	5.68	-
	$t_{bus,\ rise}$	0.0239	0.0207	0.0155	0.0157	-
	$t_{bus,\ settle}$	0.0879	0.0166	0.0826	0.0769	-

Table V.1: This is a table showing the results of all possible simulations to and from the five operating modes. The final mode is specified by the leftmost column and runs horizontally across the table. The initial mode of the transition is given on top by the vertical columns. The settling time is the time for the output to settle within 2% of the final voltage. For transitions to Load Drop-off, the settling time is determined when the bus settles within 5% of the final voltage due to large steady-state ripple experienced during Load Drop-off (larger than 2% of final value).

reduced. This reduction in transition time is due to the lessened load at the output of the converter leading to underdamping. So while an undesirable ripple is experienced in a transition to Load Drop-off, transitions to Load Drop-off occur comparatively quicker due to the load's effect on the boost conversion ratio

If we turn attention to the transition from Hybrid-Drive to Battery only drive, we see that the behavior is quite different from the reverse transition of Battery-Only to Hybrid-Drive. The transition to Battery-Only drive incurs a voltage increase of 140V, yet rises nearly 20 times faster than the reverse transition. This tells us that positive voltage adjustments occur much more quickly in our implementation than do negative voltage adjustments. This is due to 1) the PWM being bandlimited at D=0.02 prohibiting quick voltage drop-off, and 2) the DC/DC converter briefly entering CCM during positive voltage adjustments creating surges in voltage at the output. This dircetional incongruity in voltage adjustment remains an area of improvement to our system: the inductor could be swapped out, or the PWM could have its frequency, bandwidth, or precision increased to remedy this design issue.

A very common use scenario will be a Battery-Charging transition to Float Charging. This behavior would be expected after the battery becomes charged to full capacity. We see that this mode transition includes a -12V adjustment across the battery's terminals. Like the other transitions terminating in Float Charging, the signal rises quicker than in other modes, but settles in a comparatively slower time. This is because the 2% settling threshold is much narrower with a final voltage of 288V, compared to a voltage of 720V. Transitions to Float Charging experience the lowest ripple compared to all other transitions. That said the transition of Float Charging to Battery-Charging is nearly identical to the forward transition.

Some transitions, in particular the transitions terminating in Battery Charging, experience no change in voltage. This arises because the battery supplies 300V before the transition, so it requires no voltage adjustment over the mode change. In these simulations the rise and settling times occur at the instant of transition, or at 0 seconds. We measure the steady-state ripple of the battery voltage that is bucked down from the bus. If the ripple causes the battery voltage to unsettle, then $t_{batt,\ settle}$ becomes non-trivial.

CHAPTER V

Conclusion

In this paper we have presented model-integrated computing as an effective and efficient method for developing, maintaining and evolving large-scale cyber-physical systems. Cyber-physical systems−which are typically comprised of mixed models of computation−are fundamentally difficult to design because the consequence of many design decisions are not realized until late into the design process. This leads to a costly iterative process which often relies on fully deploying the system before verifying the system's functional and temporal correctness. The model based approach to system design and software development is desirable because it permits the automatic composition, analysis, and generation of simulation models which can provide valuable insights regarding the validity and behavior of various aspects of design.

Because of the diverse nature of cyber-physical systems a single modeling language cannot properly represent all incorporated domains of a system. Modeling languages routinely are catered towards describing a specific domain such as electrical dynamics, software, or signal flow. To decrease the costs of integrating domain specific modeling languages and associated simulation tools, the ESMoL tool suite provides a methodology for customizing and interconnecting models from disparate formalisms.

Through the integration language of CyPhyML, we are able to encapsulate models from multiple tool suites like Simulink or Modelica and compose them together through a unified semantics. In this paper we demonstrated the application of these techniques through the development of a bidirectional DC/DC converter circuit and accompanying software controller for a hybrid electric vehicle.

Our ability to successfully run simulations on and perform formal analysis proves the effectiveness of our Signal Flow DSML. Unlike the converter circuit, the computational vehicle controller component could not be appropriately modeled in Modelica, therefore, the controller required a modeling language capable of specifying synchronous, discrete-event semantics. The Signal Flow language possessed these characteristics, and our extension of Signal Flow ensured that the library contained the appropriate functional blocks for creating the controller. The CyPhyML integration language then allowed Signal Flow type components to be composed alongside Modelica components in our system model.

We integrated a software controller designed in Simulink together with a Modelica-based circuit model within GME−our application for manipulating CyPhyML models and underlying metamodels. We were able to test and simulate system-level behavior of the converter by leveraging CyPhyML's support for automatic synthesis of simulation models. Numerous testbenches, covering vital usage scenarios, were exported and simulated within Modelica's operational semantics. These dynamic simulations provided valuable feedback regarding the design of our converter model.

The most valuable information we gathered from the use case simulations was insight into the component parameters of the DC/DC converter circuit. Specifically, we witnessed how fluctuations in load resistance can drastically affect the conversion ratio and stability of the converter. From our data, we observed that all simulations which underwent a load drop-off experienced a large increase in steady-state ripple. In many cases, the ripple became so large that the t_{settle} metric became skewed since the voltage never settled within 2% of the target voltage. The increase in ripple is owed to the load resistance altering the conversion ratio, therefore, creating a circuit with less precise conversion at the same output voltage.

What remains is to update our model based on our knowledge feedback. For the case of a fluctuating bus load, it would be practical to adjust the converter's PWM to increase its granularity. It may even be possible to alter the behavior of the PWM specifically during drop-off mode since this mode is unique from other modes, requiring elevated voltage precision. Another area for improvement would be to consider replacing the converter's inductor, or bus capacitor specifically to compensate for the slow negative adjustment in bus voltage during boost mode operation. While positive adjustments of voltage—like transitions into battery-only mode—elapsed in less than 25ms, negative adjustments occurred slower by a factor of 16. This could be practically resolved by choosing a smaller bus capacitor, or by extending the duty range of the PWM to allow for a zero-valued duty ratio.

Future work on the DC/DC converter model would include to deploying the modeled code, and constructing the physical converter circuit. We could leverage our Signal Flow model to synthesize actual C-code that could be implemented within the controller's target hardware. The effectiveness of our model-baed approach could be tested in this way by generating controller code, building the circuit, and then performing dry-run tests of the DC/DC converter. Due to the relatively large voltages and currents within our circuit, a fault in operation could cause harm to the operator, or the destruction of circuit components. This highlights the benefits of employing modeling to perform tests of safety-critical system behavior.

While many designers make use of models during system design, our MIC approach embraces modeling as the central activity of design [3]. The support of mixed domains and multiple models of computation allows us to represent system components in a fashion that directly pertains to our complete understanding of the system and its dynamics. The upfront investment in modeling the system will be realized through shorter design cycles, and throughout the lifetime of the system in terms of maintenance, upgrade, and continued development.

BIBLIOGRAPHY

[1] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty. "Model-Integrated development of embedded software", *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145-164, Jan 2003. `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1173205&isnumber=26369`

[2] G. Hemingway, H. Neema, H. Nine, J. Sztipanovits, and G. Karsai. "Rapid Synthesis of High-Level Architecture-Based Heterogeneous Simulation: A Model-Based Integration Approach", *SIMULATION*, vol. 88, no. 1, pp.273-232, March 2011.

[3] G. Nordstrom, J. Sztipanovits, G. Karsai, A. Ledeczi. "Metamodeling - Rapid Design and Evolution of Domain Specific Modeling Environments", *Proceedings of the IEEE ECBS'99 Conference,* Nashville, Tennessee, pp. 68-74, April 1999.

[4] J. Porter, D. Balasubramanian, G. Hemingway, and J. Sztipanovits. "Towards incremental cycle analysis in ESMoL distributed control system models", *In Software Composition*, Springer Berlin Heidelberg, pp. 133-140, 2011.

[5] P. Barton. "Industrial Experience with Dynamic Simulation." *Massachusetts Institute of Technology.* 1997. `http://yoric.mit.edu/sites/default/files/BartonDynamicSimNotes.pdf`

[6] J. Contreras, J. Mara Ferrer."Dynamic simulation: a case study", *Hydrocarbon engineering*, vol. 10, no. 5, pp. 103-107, 2005.

[7] G. Hemingway, J. Porter, N. Kottenstette, C. vanBuskirk, G. Karsai, and J. Sztipanovits."Automated synthesis of time-triggered architecture-based TrueTime models for platform effects simulation and analysis", *Rapid System Prototyping,* Fairfax, VA, IEEE, pp. 1-7, Oct, 2010.

[8] G. Nordstrom, G. Karsai, M. Moore, T. Bapty, and J. Sztipanovits, ”Model Integrated Computing-Based Software Design and Evolution”, Conference on Life Cycle Software Engineering Technology for Modern Avionics, Missiles, and Smart Weapon Systems, Huntsville, Alabama, August 2000.

[9] D. J. Cooper. “Practical Process Control”, *Control Station Inc.* Tolland, CT, 2006.

[10] J. Porter. “The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design Part 1: Design Language, Modeling Framework, and Analysis”, *Vanderbilt University,* Technical Report ISIS-10-109, 2009.

[11] G. Simko, D. Lindecker, T. Levendovosky, E. Jackson, S. Neema, J. Sztipanovits. “A Framework for Unambiguous and Extensible Specification of DSMLs for Cyber-Physical Systems”, *ECBS*, Phoneix, AZ, IEEE, April 2013.

[12] X. Koutsoukos. “Introduction to Cyber-Physical Systems”, Security of Cyber-Physical Systems, Vanderbilt University, Feb 2013.

[13] National Research Council. “Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers”, *The National Academies Press*, Washington, DC, 2001.

[14] E. A. Lee, H. Zheng. “Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems”, In *EMSOFT,* Salzburg, Austria, ACM, 2007.

[15] D. Harel, A. Pnueli, “On the development of reactive systems”, *Logics and models of concurrent systems*, pp. 477-498, 1985.

[16] A. Benveniste, G. Berry, “The synchronous approach to reactive and real-time systems”, *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270, 1282, Sep 1991.

[17] S. Edwards, “Synchronous Reactive Systems and the SR Domain”, *Synopsis*, 2000. http://ptolemy.berkeley.edu/conferences/97/sedwards.pdf

[18] G. S. Fishman, “Discrete-event Simulation: Modeling, Programming, and Analysis”, *Springer*, New York, 2001.

[19] E. A. Lee, “Modeling concurrent real-time processes using discrete events”, *Annals of Software Engineering*, vol. 7, pp. 25-45, 1999.

[20] H. Olsson, H. Elmqvist, M. Otter, “Modelica: A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification Version 3.3”, May 2012. https://www.modelica.org/documents/ModelicaSpec33.pdf

[21] P. Fritzson, P. Bunus, “Modelica, a general object-oriented language for continuous and discrete-event system modeling and simulation”. *Proceedings of the 35th Annual Simulation Symposium*, San Diego, CA, April 2002.

[22] M. Belfiore, “Adaptive Vehicle Make DARPA’s Plan to Revolutionize Auto Manufacturing”, *Popular Mechanics*, Jan 2012.

[23] RDP Group, “How it Works : LVDT”, *RDP Electrosense*, Pottstown, PA, Sep 2006. http://www.rdpe.com/displacement/lvdt/lvdt-principles.htm

[24] Z. Lattmann, A. Nagel, T. Levendovsky, T. Bapty, S. Neema, G. Karsai, “Component-based Modeling of Dynamic Systems using Heterogeneous Composition”, *6th International Workshop on Multi-Paradigm Modeling,* MPM’12, Jan 2012.

[25] M. Hornauer, “MATLAB and Simulink Basics”, *Technical University of Munich,* Dec 2011. http://www.fsd.mw.tum.de/Homepage-Daten/TUM-GS-Kurse/Training_TUM_GS_Simulink.pdf

[26] The Mathworks, Inc., “MATLAB Solutions : Control Systems”, 2013. http://www.mathworks.com/

[27] W. Wolf, “The Good News and the Bad News : Embedded Computing Column”, *IEEE Computer*, vol. 11, pp. 104, Nov 2007.

[28] D. Negrut, “Advanced Computational Multibody Dynamics”, *University of Wisconsin, Madison*, Jan, 2010. `http://sbel.wisc.edu/Courses/ME751/2010/Documents/lecture0119.pdf`

[29] Z. Lattmann, A. Nagel, J. Scott, K. Smyth, C. vanBuskirk, J. Porter, S. Neema, T. Bapty, J. Sztipanovits, “Towards Automated Evaluation of Vehicle Dyamics in System-Level Designs”, *Proceedings of the ASME 2012 IDETC/CIEC*, Chicago, IL, vol. 2, pp. 1131, Aug 2012.

[30] V. Basili, “Software Modeling and Measurement : The Goal/Question/Metric Paradigm”, *University of Maryland,* CS-TR-2956, UMIACS-TR-92-96, September 1992.

[31] C. Hardebolle, F. Boulanger, “Exploeing multi-paradigm modeling techniques”, *SIMULATION*, Transactions of The Society for Modeling and Simulation International, vol. 85, pp. 688-708, Nov 2009.

[32] C. Kong, P. Alexander, “The Rosetta meta-model framework”, *Proceedings of the IEEE*, Engineering of Computer-Based Systems Symposium and Workshop, 2003.

[33] E. A. Lee, Y. Xiong, “A Behavorial Type System and its Application in Ptolemy II”, *Formal Aspects of Computing*, vol. 16, no. 3, pp. 210-237, 2004.

[34] C. Hardebolle, F. Boulanger, “ModHelX: A Component-Oriented Approach to Multi-Formalism Modeling”, *Models in Software Engineering : Workshops and Symposia at MoDELS*, Nashville, TN, pp. 247-258, Oct 2007.

[35] P.-A. Muller, F. Fleurey, J.-M. Jézéquel, “Weaving executability into object-oriented meta-languages”, *The 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS/UML)*, pp. 264-278, 2005.

[36] J. de Lara, H. Vangheluwe, “Using AToM3 as a Meta-CASE Tool”, *4th International Conference on Enterprise Information Systems (ICEIS)*, 2002. `http://www.cs.mcgill.ca/~hv/publications/02.ICEIS.MCASE.pdf`

[37] S. Taranovich, “Proper Litium-Ion Battery Charging and Safety”, *EDN Network: Power Management Design Center*, Jan 2013.

[38] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson, “Semantic anchoring with model transformations”, *ECMDA-FA*, Springer, pp. 115-129, 2005.

[39] K. Chen, J. Porter, J. Sztipanovits, S. Neema, “Compositional Specification of Behavioral Semantics for Domain-Specific Modeling Languages”, *Int. J. Semantic Computing*, vol. 3, no. 1, pp. 31-56, 2009.

[40] A. K. Jain, S. Mathapati, V. T. Ranganathan, V. Narayanan, “Integrated Starter Generator for 42-Volt PowerNet Using Induction Machine and Direct Torque Control,” *IEEE Transactions on Power Electronics,* vol. 21, no. 3, pp. 701-710, May 2006.

[41] RoboteQ, Inc., “Managing regeneration in RoboteQ controllers”, *RoboteQ, Inc.,* Application Note: AN70614, ed. 0.1, Jun 2007. `http://roboteq.com/files_n_images/files/apnotes/AN70614-Understanding%20Regeneration.pdf`

[42] A. Emadi, Y.-J. Lee, K. Rajashekara, “Power Electronics and Motor Drives in Electric, Hybrid Electric, and Plug-In Hybrid Electric Vehicles”, *Industrial Electronics, IEEE Transactions on* , vol. 55, no. 6, pp. 2237-2245, Jun 2008.

[43] W. Kester, J. Buxton, "Battery Chargers", *analog.com*, Mar 2009. `http://www.analog.com/static/imported-files/tutorials/ptmsect5.pdf`

[44] Robert Bosch LLC, "Chassis Systems Control : Regenerative braking systems", *Bosch Automotive Technology*, Hellbronn, Germany, no. CCA-201109-En, Nov 2011. `http://www.bosch-automotivetechnology.us/media/db_application/downloads/pdf/safety_1/en_4/CC_Regenerative_Braking_Systems.pdf`

[45] H. Yeo, D. Kim, S. Hwang, H. Kim, "Regenerative Braking Algorithm for a HEV with CVT Ratio Control During Deceleration", *INternational Continuously Variable and Hybrid Transmission Congress,* September 2004.

[46] R. Erickson, D. Maksimovíc, "Fundametnals of Power Electronics", *Kluwer Academic*, Norwell, MA, ed. 2, 2001

[47] J. Zhang, "Bidirectional DC-DC Power Converter Design Optimization, Modeling and Control", Ph.D dissertation, Virginia Polytechnic Institute and State University, 2008.