Functional Debugging of Equation-based Languages

Arquimedes Canedo[1] Ling Shen[1]

[1]Siemens Corporation, Corporate Technology, Princeton, USA,
{arquimedes.canedo, ling.shen}@siemens.com

Abstract

State-of-the-art debugging techniques for equation-based languages follow a low-level approach to interface users with the complex interactions between equations and algorithms that describe cyber-physical processes. Although these techniques are useful for understanding the low-level behaviors, they do not provide the means for creating a system-level understanding that is often necessary during the early concept product design phase. In this paper, we present a novel debugging technique for equation-based languages based on a high-level approach to facilitate the system-level understanding of complex cyber-physical processes. Our debugging interface is based on functional models that describe what the system does in a formal language that uses natural language elements to improve inter-disciplinary communication. Our novel technique, referred to as functional debugging, can be used in the context of the current systems engineering industrial practice in order to identify system-level problems and explore design alternatives during the early concept design phase. We present a working implementation of our functional debugger and we discuss the benefits of our approach using an automotive use-case.

Keywords Functional modeling, debuggers, equation-based languages, simulation, cyber-physical systems, concept design

1. Introduction

Product development, from consumer products to military systems, is a highly competitive area where companies are constantly challenged to meet quality targets, revenue targets, and launch dates for new and innovative products [2]. In order to reduce the product development cycle, companies use systems engineering methodologies that attempt to parallelize and detect errors in the design as early as possible. For example, DARPA's META-II project [59] has the qualitative goal to compress the system design, development, test, and evaluation of mission critical design applications by a factor of 5x or more by identifying system-level and component interaction problems early in the design cycle. Currently, most computer-based design tools are suitable for detail design and it is very difficult or impossible to effectively front-load the detection of system-level design flaws [10, 17].

Products characterized by a blend of multiple disciplines including mechanical, electrical, thermal, software, and control are often referred to as cyber-physical systems (CPS). CPS are often characterized by the use of dynamic architectures (e.g. based upon the availability of elements such as sensors) that produce online, emergent, and on-the-fly unprecedented behavior. Therefore, CPS design, analysis, validation necessitates a new systems science that encompasses both physical and computational aspects [1]. Object oriented equation-based languages are often used to describe CPS because they can be used to model the behavior of both continuous (physical-) and discrete (cyber-) processes. To facilitate physical modeling in terms of energy conservation principles, these languages are implemented as *declarative* programming languages that describe *what* the goal is. Debugging these programs is very challenging because during execution or simulation, these programs are highly optimized and transformed [44] into *imperative* programs that instruct the computer *how* to reach the goal. Unfortunately, these debugging techniques expose the user with the low-level details of the model and therefore, it is difficult to incorporate these techniques in tools for the early concept design phase.

In this paper, we introduce a new debugging technique suitable for the concept design phase. Based on the observation that functional models describe *what* the system is supposed to do, and models in equation-based languages describe *what* the cyber-physical process is, we provide a *functional debugging* interface that helps users understand complex processes in a high-level of abstraction. Our implementation couples a functional model (functionality) with an underlying simulation model (behavior). This enables, for the first time, a dynamic functional representation of the system that serves as a quick validation tool for new design concepts. The functional debugging technique can be integrated into the systems engineering process by reusing functional and simulation components and allowing the identification of system-level problems early in the design. Specifically, the novel contributions of this paper are:

- A model-based debugging methodology, referred to as *functional debugging*, that interprets the results of simulation models written in equation-based languages in a high-level manner and allows the identification of system-level errors and integration problems early in the design cycle.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.
 Proceedings available at:
http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084
EOOLT 2013 website:
http://www.eoolt.org/2013/

- The observation that declarative equation-based languages fundamentally describe *what* the system does and therefore can be naturally mapped to functional models that also describe *what* the system is supposed to do but in a higher-level of abstraction that is suitable for communication and design space exploration of new concepts.
- An implementation of the functional debugging methodology that, for the first time, provides a *dynamic* or *executable* functional model that effectively combines functionality and behavior in the same model.

The rest of the paper is organized as follows. Section 2 puts our work into context with an overview of the state-of-the-art in equation-based languages and their debugging techniques, and functional modeling. Section 3 introduces our new functional debugging approach and provides the details of our implementation. Section 4 presents how the functional debugger can be integrated into a systems engineering process with an automotive use-case. Section 5 summarizes our findings and provides the outlook for future work.

2. Background and Related Work

2.1 Physical Modeling with Equation-based Languages

In recent years, companies from all sectors are designing complex products through *physical modeling* – the combination of components that correspond to physical objects in the real world (e.g. pipes, motors, resistors, software). This approach is very attractive because reusable components encapsulate an associated behavioral description according to the laws of physics and principles of energy conservation. The interconnection of components in a model creates complete mathematical models that effectively combine different disciplines. Thus, by focusing the design on the structure of the system and automatically finding the equations that describe its behavior, physical modeling eliminates the need for manually finding mathematical descriptions of systems [51]. Equation-based languages such as Bond Graphs [12], Modelica [33], Simscape [28] have been developed to provide the syntax and semantics for physical modeling. Most equation-based languages are declarative programming languages that describe *what* the program should accomplish. It is the responsibility of the compilers and optimizers to transform equation-based declarative programs into an imperative program that specifies *how* to accomplish the goal as most numerical solvers require an imperative program to simulate the dynamic behavior of the system. Due to the extensive transformations that a declarative program suffers when converted into its imperative equivalent, *what the user sees (equations in the declarative model) is NOT what the user gets (code in the imperative simulation)*, and therefore it is very challenging to debug these applications.

2.2 State-of-the-art Debugging Techniques for Equation-based Languages

Debugging equation-based languages is a challenging problem that requires a combination of classical debugging techniques and other special techniques. In [44], the authors provide a comprehensive survey of the state-of-the-art in techniques for debugging declarative equation-based languages typically used in physical modeling. These debugging techniques are categorized as static (compile-time) and dynamic (run-time). Static techniques focus on tracing the complex process of symbolically transforming declarative code into highly optimized imperative code to provide explanations regarding problematic code. Novel and innovative static debugging techniques using graph-theoretic methods have been developed [9]. Dynamic techniques, on the other hand, are similar to classical debugging and focus on interactively inspecting the imperative parts of the model that relate to functions and algorithms typically used to describe control code and embedded software. Hybrid approaches [44] that combine static and dynamic methods are the most advanced debugging techniques for equation-based languages.

Although these techniques are invaluable for identifying errors in models and code during the detail design phase, they must focus on the low-level aspects of modeling and simulation. Integrating these debugging techniques to the first iterations of the systems engineering processes is difficult because a high-level of abstraction, rather than a low-level, is preferred during the early concept design phase of modern cyber-physical systems [25]. In this paper, we present a debugging technique that deals with the functional aspects of equation-based languages and presents to the user a high-level interface to complex cyber-physical processes to facilitate the conceptual design space exploration of complex products. In the following Sections we discuss how our high-level debugging approach and state-of-the-art low-level debugging techniques are complementary in a systems engineering context.

2.3 Functional Modeling

Functional modeling is a systems engineering activity where products are described in terms of their functionalities and the functionalities of their subsystems. Fundamentally, a functional model reflects *what* the system does and, therefore, we observe that functional models are strongly related to declarative equation-based languages. Because a Functional Model decouples the design intentions (functions) from behavior and/or structure (logical components[1]), it can be used as the basis for communication among engineers of different disciplines. Functional modeling reflects the design intentions that are typically driven by the product requirements and the human creativity.

Functional modeling is acknowledged by many researchers and practitioners to be a subjective process [17], therefore suitable for concept design. Defining a system in terms of its functionality[2] may seem simplistic and unnecessary but this is exactly what improves the systems engineering process by consolidating multiple engineering paradigms (e.g. electrical, mechanical, software, thermal engineering) into a unified system representation. By *making explicit the implicit knowledge of the engineers*, a functional model exposes the obvious facts about the system that people can easily understand, regardless of their domain of expertise. This improves the communi-

[1] Logical components (and models) are often used as the guidelines for the creation of simulation models.

[2] Functionality of a system is defined as its purpose, intent, or goal.

cation among different disciplines because it brings the minds of the domain experts and designers to a system-level abstraction that is facilitated by natural language. In this paper, we introduce a novel high-level debugging technique suitable for early concept design phases that uses functional modeling as a debugging interface for equation-based languages. Compared to existing research on functional modeling [17, 10, 46, 8, 25, 61], we are the first to demonstrate the use of functional models for debugging simulation models.

3. Functional Debugging

In this paper, we define *functional debugging of equation-based languages* as the mechanism by which states and variables of a running simulation are visualized through a functional model to create an implementation independent understanding of a cyber-physical process. As shown in Figure 1, a functional debugger relies on three components : a functional editor, a simulation model synthesizer, and a simulation runtime[3]. The functional editor is a visual programming environment for users to author functional models that describe *what the system does*. The functional editor is also used as the debugger user interface that allows users to visualize and interact with the simulation in a high-level of abstraction. Our implementation uses Microsoft Visio as the functional editor. The simulation model synthesizer is a computer program (automatic) or a simulation expert (manual) that takes a functional model as an input and generates a corresponding simulation model that *realizes* or *embodies* the system's functionality. This simulation model provides the executable semantics to the functional model. In addition to the simulation model, the synthesizer also generates a mapping model that associates functions to simulation components. Finally, the simulation runtime simulates the simulation model and calculates the dynamic behavior of the system. It is important to note that different simulation runtimes may be used to simulate the same functional model. For example, the thermal-vibration facet of a functional model may be simulated using a finite element analysis solver, and its 1D electro-mechanical facet may be simulated using Modelica or Simscape.

The functional debugger takes a functional model, a simulation model, and a mapping model as inputs. The mapping model specifies how functions and flows in the functional model associate to simulation components and effort/flow variables in the simulation model. This information is used during debugging (dotted lines in Figure 1) to relate the simulation output to visualization in the functional model, and to relate user interaction debugging commands to the running simulation. For interactive debugging, the functional debugger should be capable of controlling a simulation through pausing, stopping, resuming, advancing time to the next integration step, and querying simulation variables. The rest of this Section describes our implementation of the functional debugger architecture.

3.1 Functional Editor

Visual programming languages are suitable for authoring functional models [56, 21, 42, 48, 17, 66, 23] because a

[3] In this paper, we use *simulation runtime* and *simulation engine* interchangeably.

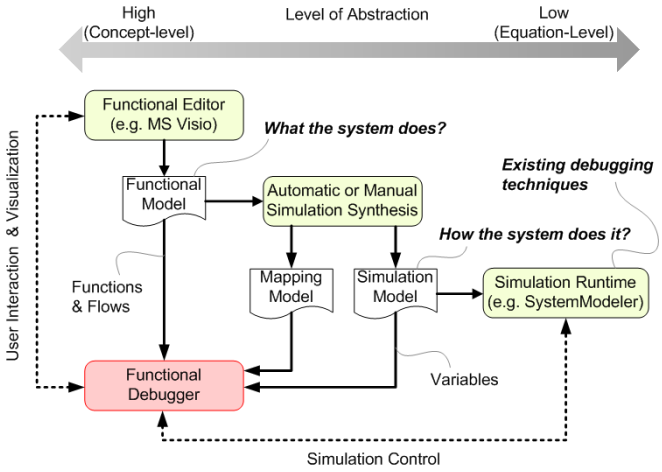

Figure 1. Functional Debugging Architecture consists of three main components: a functional editor, a synthesizer, and a simulation runtime. Different models are necessary for the functional debugger to relate functions to behavior.

Table 1. Functional modeling shapes in Visio stencil.

Visio Shape	Syntax
Function Block	Function
Material Flow	Material
Energy Flow	Energy
Signal Flow	Signal

diagrammatic representation facilitates the understanding of the system as a collection of functionalities interacting through the exchange of material, energy, and signals. Although a functional model can be also expressed textually, or as a design matrix [31], we believe that a visual functional editor improves the productivity of designers and our implementation provides an editor based on Microsoft Visio ActiveX control that can be easily embedded in other systems engineering tools. We have extended Visio with a C# implementation to improve the user-interaction and to manage the communication and data transfer between the displayed interface and the simulation runtime.

The functional modeling types are provided as shapes in a Visio stencil as shown in Table 1. We use the de-facto functional modeling syntax consisting of a block-flow diagram where blocks represent functions (process) that transform inputs into outputs (flows) [21, 42]. Blocks and flows use the Functional Basis syntax [56] to categorize functions into 8 categories and a total of 32 primitive functions, and flows into 3 categories (*material*, *energy*, and *signal* and a total of 18 flow subtypes. Constraining the vocabulary for functional modeling is beneficial for the systems engineering process because it normalizes the understanding and consistency of the models across the computer-aided tools and the organization. Although functional modeling is a highly subjective and creative process [17], the use of a constrained vocabulary does not affect the expressiveness of the functional models.

In the functional editor, a functional model can be refined into more specific descriptions in a process referred to as *functional decomposition*. For example, in the functional model of an automobile shown in Figure 2, the "`transport people`" function can be decomposed into sub-functions such as "`Store Chemical Energy`" and "`Convert Chemical Energy to Rotational Mechanical Energy`" implying the design of an internal combustion engine car. Furthermore, sub-functions can be decomposed to create a *functional decomposition tree* where the root node represents the top-level function and the leaf-nodes represent elementary functions such as "`Transfer Translational Mechanical Energy (TME)`".

Although our implementation uses the Functional Basis vocabulary, we use different semantics and we have added additional function types to facilitate the modeling of modern cyber-physical systems. For example, the original Functional Basis specifies that functional models are executed from left-to-right [56]. This causality rule, unfortunately, prohibits the coupling of functional models to acausal equation-based simulation languages because a change in the direction of energy flow during simulation is not expressible in the original Functional Basis semantics. Moreover, this causality rule does not allow for *feedback loops*, an essential construct for control theory modeling. To overcome these limitations, our functional editor allows acausal (left-to-right and right-to-left) execution semantics and the creation of feedback loops anywhere in the functional model as shown in the Third-level Functions in Figure 2. Moreover, we provide additional elementary functions for "`Control`" (function in black) and "`Sense`" (function in gray) to model cyber-physical control systems. Note that the functional modeling flows are represented by a directed arrow in Table 1. This is simply the syntax of the Functional Basis [56] and during debugging, the simulation semantics will affect the look and feel of these flows and functions. In other words, although the static functional model is constructed with directed flows, the dynamic functional model implies and reflects energy and material transfers in both directions and this also affects the functions' signatures.

3.2 Simulation Model Synthesis

The goal of the simulation model synthesis is to find components that fulfill the functionalities in a functional model. The synthesis can be performed manually by a simulation expert, or automatically by a synthesis tool. Automatic synthesis of functional models to simulation models is challenging because one function may be realized by multiple and different components, and one component may realize multiple functions. In other words, multiple valid simulation models exist for a given functional model, but only a few are useful for modeling the actual system. In our previous work [11], we introduced a *context-sensitive synthesis algorithm* that reliably generates high-quality simulation models from functional models. The synthesizer puts every function within a functional model into a context provided by its input and output flows, and using *engineering rules*[4] it correctly maps functions to the specified simulation components from reusable component libraries. Engineering rules and simulation component libraries are the means for capturing *engineering knowledge*. Due to the easy access to various simulation component libraries [34, 26, 36], our synthesizer currently generates Modelica code as an output. However, the synthesizer can be easily modified to emit and reuse components from other equation-based languages.

A simulation model consists of components with well defined interfaces, and each component may contain equations, variables, and algorithms. In order to create a correct mapping from functions to simulation components, the functional debugger must associate functions and flows in a functional model with components and variables in a simulation model. In the case of automatic synthesis, the output of an engineering rule is the mapping of functions and flows to components and variables. On the other hand, manual mapping requires the designer to make these relations by looking at both the functional and the simulation models and deciding how the two models relate. Either way, the functional debugger needs access to functional models, simulation models, and the mapping model.

3.2.1 Mapping of Functional Models to Simulation Models

It is possible to relate functional models (functions and flows) to simulation models (components and variables) because the concept of physical quantities exist in both models. Functional models specify *material*, *energy*, and *signal* flows and transformation functions operating on these flows. Physical-based equation-based languages, on the other hand, specify complementary *physical domains* such as *electricity*, *mechanics*, *software*, etc., and the physical behavior of components operating and governed by laws on these domains such as a *resistor*, *gearbox*, or *PID controller*. An important observation is that a single functional *energy* flow maps to a pair of conjugate variables in the simulation model that are used to accomplish *acausal* modeling[5]. These conjugate variables are known differently in different equation-based languages but have very similar semantics. For example, Modelica uses *potential-flow* [33] variables, Bond Graphs use *effort-flow* [12] variables, and Simscape uses *across-through* [28] variables.

Table 2 (adapted from [56]) shows the mapping between flow types (e.g. `electrical`, `magnetic`, etc. in Column 2) in a functional model to conjugate variables (Column 3) in equation-based simulation languages. The last two Columns shows some of the system-level equation-based languages (e.g. Modelica) and domain-specific equation-based languages (e.g. CAD/CAE) that are typically used to simulate physical systems. This table shows that functional models can be mapped to both system-level languages and domain-specific languages and therefore, functional debugging can be adapted to various equation-based languages. Notice that a single equation-based language is not sufficient to cover all the functional flow types. Even though it is out of the scope of this paper, we believe it is important to observe that functional debugging can be also used to comprehend multi-tool multi-language co-simulations of complex systems.

[4] Engineering rules are analogous to machine description files in a traditional compiler.

[5] Acausal modeling describes the behavior of components in terms of energy conservation laws [19].

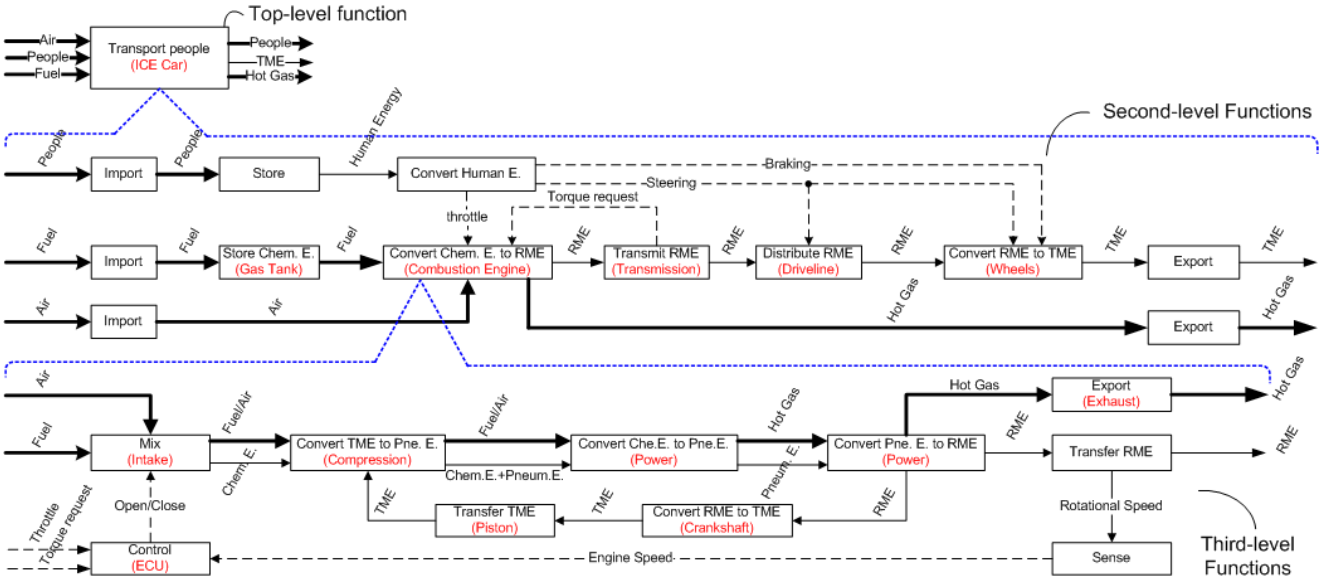

Figure 2. Functional model of an internal combustion engine car showing the functions associated with the main powertrain subsystems (in parentheses). Syntactically and semantically, our functional modeling approach handles feedback loops.

Our functional debugger implementation uses a data structure referred to as the Mapping Model (See Figure 1) to read the mapping information of functions and flows (Functional Model) to components and conjugate variables (Simulation Model). Although it is common that mappings are from *function(s)-to-component(s)* and *flow(s)-to-variable(s)*, other combinations are also possible including *flow(s)-to-component(s)* and *component(s)-to-variable(s)*. For example, a "`pneumatic energy`" flow in a functional model may be mapped to a "`pipe`" component, or to a "`pressure`" variable.

3.3 Simulation Runtime

A simulation runtime responsible for executing the simulation models, is the last component required for functional debugging. Although a simulation model is heavily transformed and optimized into a mathematical model for integration with numerical methods, the variables remain visible *during* simulation. Using the mapping model, the functional debugger can query the variables'status and values during simulation.

From the functional debugging perspective, there are two important requirements for the simulation runtime. First, in order to facilitate a natural human-computer interaction in the functional debugger, the simulation runtime must allow the synchronization of the simulation time with the real (human) time. Whenever the simulation time is faster than the real time, the simulation runtime must delay the execution of the simulation in order to synchronize the two times. In case that the simulation time is slower than the real time, the simulation runtime can adopt execution strategies similar to the ones used in hardware-in-the-loop simulations including fixed-step size solvers, loop tearing, or iterative limits. The second requirement demands the simulation runtime to be programmatically controlled by the functional debugger in order to start, pause, stop, and proceed to the next iteration step during the simulation. Currently, our functional debugger uses Wolfram's SystemModeler [64] as the simulation runtime and the next Section discusses the details of our implementation.

3.4 User Interaction, Visualization, and Simulation Control

The functional debugger consists of three applications as shown in Figure 3. The Functional Editor or Functional Debugger GUI (left) handles the user interaction events such as breakpoints and visualization requests on specific functions and flows. This C# application extends the functionality of Visio through the Visio Object Model [32] and allows the functional debugging specific commands and visualization such as stop, pause, restart, and perform the next iteration step; load functional, simulation, and mapping models; detect user events to debug specific functions and flows and to zoom in/out in the functional model hierarchy; manipulate the look & feel of Visio shapes representing the functional model to convey points of interest during the simulation. These points of interest can be pre-programmed by the user to monitor a range of operation of a subsystem, or built-in into our implementation (e.g. indicate when the energy flow changes direction). In our implementation, the simulation runtime (right) [64] uses an application-specific TCP protocol that allows a client application to control the simulation and set/receive simulation data. After the simulation runtime server has been initialized for control commands and data flow, this application streams data over TCP to the client after every integration time step. Through an initialization file, this application can be configured to maintain the simulation time and the real-time synchronized. The functional debugger application (middle) is the intermediary between the GUI and the simulation runtime. Its main responsibility is to retrieve data from the simulation and map it to the functional model in the GUI, and also to control the simulation according to the user commands.

Although we have developed an in-house implementation, each component of our functional debugger has an analogous technology that could be used to provide

Table 2. Relationship between functional models and equation-based languages based on flow types.

Functional Modeling		Equation-based Languages		
Flow Class	Flow Type	Conjugate Vars. (Effort/Flow)	System-Level Lang.	Domain Specific Lang.
Energy	Electrical	Electromotive Force / Current	[34], [28]	[45], [13]
	Mechanical (Rotational)	Torque / Angular Velocity	[34], [28]	[52], [57], [5]
	Mechanical (Translational)	Force / Linear Velocity	[34], [28]	[52], [57], [5]
	Mechanical (Vibrational)	Amplitude / Frequency		[52], [57], [5]
	Hydraulic	Pressure / Volumetric Flow	[35], [28]	[53]
	Pneumatic	Pressure / Mass Flow	[34]	[53]
	Thermal	Temperature / Heat Flow	[34], [28]	[52], [57]
	Electromagnetic	Intensity / Velocity		[4]
	Magnetic	Mag. Force / Mag. Flux Rate	[34]	
	Chemical	Affinity / Reaction Rate	[38]	
	Biological	Pressure / Volumetric Flow	[38]	
	Human	Force / Motion		
	Acoustic	Pressure / Particle Velocity	[24]	[55]
	Radioactive	Intensity / Decay Rate		
Signal	Status		[41], [29]	[37], [30], [16]
	Control		[41], [29]	[37], [30], [16]
Material	Human			[54], [20] [18]
	Gas			[52], [57], [5]
	Liquid			[52], [57], [5]
	Solid			[54], [20], [18]

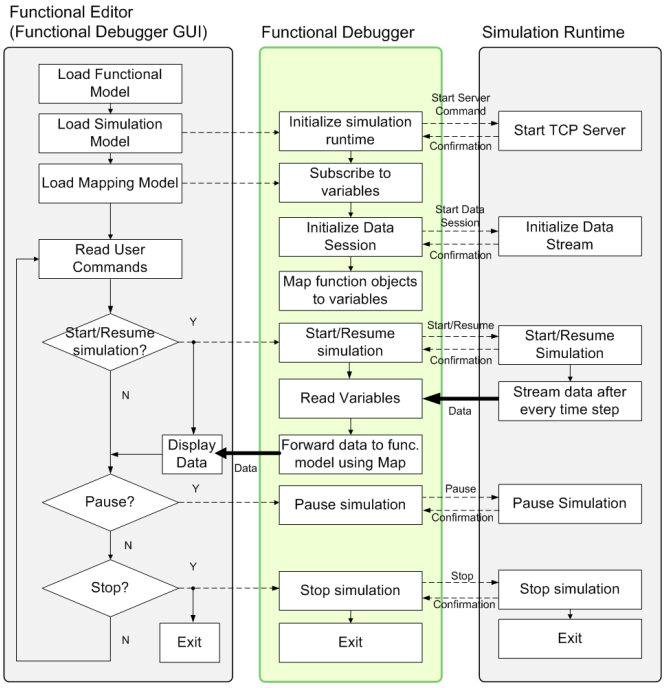

Figure 3. Control and data flow interactions between the functional editor, the functional debugger, and the simulation runtime.

the same functionality. For example, the functional editor could be implemented in SysML [39]. Several published investigations [66, 3] have shown how to create functional models in SysML. Similarly, the simulation model synthesizer could be realized by SysML4Modelica [43] or ModelicaML [50]. Open source Modelica runtimes [40, 22] could be used as the simulation runtime. And FMI [6] could be used as the communication and data transfer mechanism between the functional editor and the simulation runtime.

3.5 Industry Perspective

Concept design, despite being a critical design phase that determines 70-80% of the cost of a product [15], lacks the tool and methodology support that is available for detail design phases [49]. We strongly believe that the development of tools for conceptual design is mandatory to handle the complexity of large-scale cyber-physical systems [63, 47, 27, 58] where system-level optimization plays a critical role. Our functional debugger is a concept design tool that allows product designers to functionally understand the complex underlying cyber-physical processes of a system. Additionally, we see functional debugging as a complementary and orthogonal approach to existing debugging techniques that are employed during detail design. We also believe that functional debugging can be used as a tool to consolidate 3D CAD/kinematics with system-level simulation models. This would allow system designers to have an integrated and dynamic function-behavior-structure [60] view of the system with the capacity for testing and simulating design alternatives while reusing existing components.

4. Case Study: eCar Development

We evaluate our functional debugger with a common scenario in automotive development. In order to reduce risk and cost, automotive companies invest in the development of architectures that can be reused to produce different models of cars within and across brands [14, 7]. Therefore, it is natural that even radical new designs, such as an eCar, attempt to reuse an existing architecture and a set of compatible cyber-physical components. Functional models are used in this type of scenarios to understand the impact of major architectural changes[6] in the overall design. In summary, our objective is to demonstrate how the functional debugger supports a realistic conceptual

[6] An architecture is, after all, the allocation of functions (or functionality) to specific cyber-physical (logical) components (e.g. a gearbox, a wheel, an ECU).

design scenario where an eCar is developed while reusing, as much as possible, components of an existing architecture. Additionally, we show how our functional debugging approach is compatible and orthogonal to the existing debugging techniques for equation-based languages.

4.1 Baseline Architecture

We first created a baseline functional model of an internal combustion engine car shown in Figure 2. This baseline functional model describes the functionality of the automotive driveline industrial example in Modelica language published in [65]. The mapping of functions-to-components is indicated by the parentheses in Figure 2 and this represents the baseline architecture for our scenario. Using the baseline models as the starting point, the next step is to conceptually design an eCar with the help of functional debugging.

4.2 Concept Design Space Exploration

Conceptually, the simplest way to create an eCar from a conventional car is by replacing its internal combustion engine with an electric motor. In terms of functionality, the function "`convert chemical energy to rotational mechanical energy`" must be replaced with "`convert electrical energy to rotational mechanical energy`" as shown in Figure 4. This change also implies new functionality where the "`electrical energy`" flow is "`stored`" (e.g. in a battery).

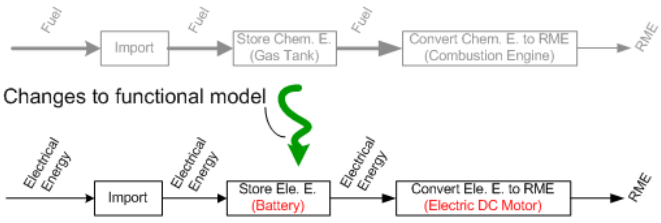

Figure 4. Changes to the functional model in Figure 2 to convey the new design intentions of an e-Car. Fuel containing chemical energy is replaced with electrical energy. This implies the use of a Battery and an Electric DC Motor instead of a Gas Tank and a Combustion Engine.

To mimic the reusability aspect in the current system engineering practice, we created new engineering rules to convert the newly introduced eCar functionality into existing simulation models of a DC motor and a battery developed in-house [62]. As a result, the synthesizer creates an aggregated simulation model that replaces the internal combustion engine component from the baseline simulation model with the battery and DC motor, but reuses the rest of the simulation components in the baseline. The coupling between the DC motor and the baseline drivetrain is possible because the two have a compatible interface and this allows the aggregated simulation model to be correctly generated and compiled using SystemModeler.

This workflow shows that a simple change in the functional model can be used to generate new simulation models that allow the designer to understand and quantify how a change in functionality of an existing architecture has an impact in the overall system-level design. The relation of functions to components, or mapping model, was created by the engineering rules and the analogy between functions and system-level equation-based languages discussed in Table 2. Therefore, at this point in time, the three input models to the functional debugger are available: an eCar functional model (Visio), a simulation model (Modelica), and the mapping model (data structure described in Section 3.2.1). The next step is to run the eCar simulation model under the functional debugger to identify any possible system-level problems created by the architectural change.

4.3 Functional Debugging

We use the New European Driving Cycle to test the eCar simulation model in the functional debugger. Figure 5 shows the functional debugger under four modes of operation: (a) Acceleration, (b) Cruise, (c) Deceleration, and (d) Idle. During acceleration in Figure 5(a), the functional debugger shows that the main energy transfer in the power train, indicated by the direction of the flows, is from left-to-right starting from the "`convert electrical energy to rotational mechanical energy`". While the Modelica simulation explains the physical behaviors, the functional debugger helps a non-expert to understand that rotational mechanical energy (RME) is functionally correct. During cruise in Figure 5(b), the functional debugger shows that there is an equilibrium of energy transfer in the powertrain and this is indicated by the bi-directional flows. During deceleration in Figure 5(c), the functional debugger shows that RME flow is from right-to-left. In addition, this functional debugging snapshot shows that the function being performed by the "Electric Motor" component changed to "Convert RME to electrical energy". This insight is very important for the systems engineer because it shows that the newly introduced electric motor is performing two functions and this can be used to validate the requirements. It is also important to note that this additional functionality can be used to recharge the battery while the eCar decelerates. During the idle mode in Figure 5(d), the functional debugger shows the case when the clutch is disengaged and the electric motor and the transmission are physically decoupled, and the functional debugger eliminates the flow connecting these two components. This causes the functionality of the electric motor to change to "`convert electrical energy to thermal energy`".

The snapshots in Figure 5 illustrate how the functional debugger can help the concept level designer to create a mental high-level picture of the system and conceptually understand how a functional and architectural change affects the rest of the system. It also allows them to visualize potential new innovations such as regenerative breaking, and visualize the energy, material, and signal flows through the system. Functional debugging can be easily integrated to the current systems engineering processes and reuse the existing and legacy simulation, functional, and architectural models. Another important feature is that functional debugging allows any non-technical person to easily understand the cyber-physical process at the functional or conceptual level.

Iterative design is a very important aspect of the systems engineering process. Although the results shown in Figure 5 are functionally correct as the system does what it is supposed to, the systems engineer must verify that

Figure 5. Visualization of an eCar simulation through the functional debugger on different modes: (a) acceleration, (b) cruise, (c) deceleration, (d) idle. Notice the energy flows across the functional model on the different modes, and the mapping of multiple functions to a single component (e.g. Electric Motor).

the eCar is reaching its performance targets by enabling the display of numerical values of the conjugate variables of the simulation in the functional model. Enabling the numerical values in the functional debugger reveals that although the system is functionally correct, the eCar never accelerates to even 5% of the desired speed. Since the eCar concept includes a newly introduced component, the electric motor, the first guess would probably be that the test needs a stronger motor. However, after simulating the eCar with a stronger electric motor, the results are still unfavorable. These quick iterations that use the functional debugger as a visualizer for the underlying simulation provide valuable information to the systems engineer about the system-level integration problems on the new concept at a level of abstraction where they can reason about the possible problem, but without being concerned about the details of the cyber-physical implementation.

Typically, this situation would lead the systems engineer to report and discuss the problem with the multi-disciplinary teams in charge of the transmission, the software, and the electro-mobility. Using the common language of functionality, engineers can communicate at a high-level of abstraction and then translate these insights to their domain of expertise. At this point in time, the domain experts would perform detail design iterations on their subsystems and this is where existing debugging techniques for equation-based languages are very useful for identifying the root cause of the problem. In this example, the problem is in the control gains in the controller software at the transmission control unit, and it required an expert to use the existing debugging techniques to find the solution. Because functional debugging is used early in the concept design phase, and its purpose is to communicate potential problems to the systems engineer using a high-level of abstraction (functionality) rather than a low-level of abstraction (behavior), we argue that it enhances the systems engineering and it is complementary and orthogonal to the existing debugging techniques for equation-based languages.

5. Summary

With the objective of supporting the early concept design phases with computer-based tools, we introduced a new methodology referred to as *functional debugging* that builds a functional view of an underlying cyber-physical process described in equation-based languages. Our implementation couples functions and flows in functional models with conjugate variables in simulation models, and this mapping enables a high-level view of *what* the system does. In a systems engineering context, our functional debugger can be used as a rapid prototyping tool for new concepts to identify system-level integration problems. Through an industrial use-case, we have shown that functional debugging can be a valuable tool for an iterative design process that involves the coordination of multiple disciplines. Additionally, we have shown that functional debugging is compatible with existing low-level debugging techniques for equation-based languages. Our future work will include the implementation of functional debugging for domain-specific equation-based languages.

Acknowledgments

The authors would like to thank Eric Schwarzenbach from Princeton University and Georg Muenzel from Siemens Corporate Technology for their support during the research and development of the functional debugger.

References

[1] Foundations for innovation: Strategic R&D opportunities for the 21st century cyber-physical systems – connecting computer and information systems with the physical world. Technical report, NIST, 2013.

[2] Aberdeen Group. System design: New product development for mechatronics. January 2008.

[3] A. A. Alvarez Cabrera, M. S. Erden, and T. Tomiyama. On the potential of function-behavior-state (FBS) methodology for the integration of modeling tools. In *Proc. of the 19th CIRP Design Conference âĂŞ Competitive Design*, pages 412–419, 2009.

[4] ANSYS. HFSS. `http://www.ansys.com`.

[5] Autodesk. Simulation Software. `http://www.ni.com/labview/`.

[6] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th Modelica Conference*, pages 105 – 114, 2011.

[7] Manfred Broy, Mario Gleirscher, Peter Kluge, Wolfgang Krenzer, Stefano Merenda, and Doris Wild. Automotive architecture framework: Towards a holistic and standardised system architecture description. Technical report, TUM, 2009.

[8] Cari R. Bryant, Robert B. Stone, Daniel A. Mcadams, Tolga Kurtoglu, and Matthew I. Campbell. Concept generation from the functional basis of design. In *Proc. of International Conference on Engineering Design, ICED 2005*, pages 15–18, 2005.

[9] Peter Bunus. An empirical study on debugging equation-based simulation models. In *Proceedings of the 4th International Modelica Conference*, pages 281 – 288, 2005.

[10] A.A. Alvarez Cabrera, M.J. Foeken, O.A. Tekin, K. Woestenenk, M.S. Erden, B. De Schutter, M.J.L. van Tooren, R. Babuska, F.J.A.M. van Houten, and T. Tomiyama. Towards automation of control software: A review of challenges in mechatronic design. *Mechatronics*, 20(8):876 – 886, 2010.

[11] A. Canedo, E. Schwarzenbach, and M. A. Al-Faruque. Context-sensitive synthesis of executable functional models of cyber-physical systems. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[12] F. E. Cellier. *Continuous System Modeling*. Springer-Verlag, 1991.

[13] E. Christen and K. Bakalar. Vhdl-ams-a hardware description language for analog and mixed-signal applications. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 46(10):1263 –1272, oct 1999.

[14] Jeffrey B. Dahmus, Javier P. Gonzalez-Zugasti, and Kevin N. Otto. Modular product architecture. *Design Studies*, 22(5):409–424, September 2011.

[15] D. Dumbacher and S. R. Davis. Building operations efficiencies into NASA's Ares I crew launch vehicle design. In *54th Joint JANNAF Propulsion Conference*, 2007.

[16] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, jan 2003.

[17] M.S. Erden, H. Komoto, T.J. Van Beek, V.D'Amelio, E. Echavarria, and T. Tomiyama. A review of funcion modeling: approaches and applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22:147–169, 2008.

[18] G. S. Fishman. *Discrete-Event Simulation - Modeling, Programming, and Analysis*. Springer, 2001.

[19] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica*. IEEE, 2004.

[20] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris. *Fundamentals of Queueing Theory*. Wiley, 4th edition, 2011.

[21] Julie Hirtz, Robert B. Stone, Simon Szykman, Daniel A. McAdams, and Kristin L. Wood. A functional basis for engineering design: Reconciling and evolving previous efforts. Technical report, NIST, 2002.

[22] JModelica. `http://www.jmodelica.org/`.

[23] Hitoshi Komoto and Tetsuo Tomiyama. A framework for computer-aided conceptual design and its application to system architecting of mechatronics products. *Comput. Aided Des.*, 44(10):931–946, October 2012.

[24] H. Kuehnelt, Thomas Baeuml, and Anton Haumer. SoundDuctFlow: a Modelica library for modeling acoustics and flow in duct networks. In *Proc. of the 7th Intl. Modelica Conference*, pages 519–525, 2009.

[25] Tolga Kurtoglu and Matthew I. Campbell. Automated synthesis of electromechanical design configurations from empirical analysis of function to form mapping. *Journal of Engineering Design*, 19, 2008.

[26] Lawrence Berkeley National Laboratory - Modelica Buildings Library. `http://simulationresearch.lbl.gov/modelica`.

[27] Insup Lee, Oleg Sokolsky, Sanjian Chen, John Hatcliff, Eunkyoung Jee, BaekGyu Kim, Andrew L. King, Margaret Mullen-Fortino, Soojin Park, Alex Roederer, and Krishna K. Venkatasubramanian. Challenges and research directions in medical cyber-physical systems. *Proceedings of the IEEE*,

100(1):75–90, 2012.

[28] MathWorks. Simscape. http://www.mathworks.com/products/simscape/.

[29] MathWorks. Simulink. http://www.mathworks.com/products/simulink/.

[30] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[31] Oregon state university, design engineering lab, design repository. http://designengineeringlab.org/.

[32] Microsoft. Visio. http://msdn.microsoft.com.

[33] Modelica Association, Modelica. https://modelica.org/.

[34] Modelica Association, Modelica Standard Library. https://modelica.org/libraries/Modelica/.

[35] Modelon. Hydraulics Library. http://www.modelon.com/products/modelica-libraries/hydraulics-library/.

[36] Modelon - Vehicle Dynamics Library. http://www.modelon.com/.

[37] National Instruments. LabVIEW System Design Software. http://www.ni.com/labview/.

[38] E. L. Nilson and P. Fritzon. BioChem - A Biological and Chemical Library for Modelica. In *Proc. of the 3rd Intl. Modelica Conference*, pages 215–220, 2003.

[39] OMG Systems Modeling Language (SysML). http://www.omgsysml.org/.

[40] OpenModelica. https://www.openmodelica.org/.

[41] Martin Otter, Karl-Erik Årzén, and Isolde Dressler. Stategraph—A Modelica library for hierarchical state machines. In *Modelica 2005 Proceedings*, 2005.

[42] G. Pahl, W. Beitz, J. Feldhusen, and K.H. Grote. *Engineering Design - A Systematic Approach*. Springer, 3rd edition, 2007.

[43] Christiaan J.J. Paredis, Yves Bernard, Roger M. Burkhart, Hans-Peter de Koning, Sanford Friedenthal, Peter Fritzson, Nicolas F. Rouquette, and Wladimir Schamai. An overview of the SysML-Modelica transformation specification. In *INCOSE International Symposium*, 2010.

[44] Adrian Pop, Martin Sjolund, Adeel Asghar, Peter Fritzson, and Francesco Casella. Static and dynamic debugging of Modelica models. In *Proceedings of the 9th International Modelica Conference*, pages 443 – 454, 2012.

[45] Thomas L. Quarles. *Analysis of Performance and Convergence Issues for Circuit Simulation*. PhD thesis, EECS Department, University of California, Berkeley, 1989.

[46] Venkat Rajagopalan, Cari R. Bryant, Jeremy Johnson, Daniel A. McAdams, Robert B. Stone, Tolga Kurtoglu, and Matthew I. Campbell. Creation of assembly models to support automated concept generation. *ASME Conference Proc.*, 2005(4742Xa):259–266, 2005.

[47] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM.

[48] S.D. Rudov-Clark and J. Stecki. The language of FMEA: on the effective use and reuse of FMEA data. In *AIAC-13 Thirteenth Australian International Aerospace Congress*, 2009.

[49] Alberto Sangiovanni-Vincentelli. Quo vadis SLD: Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[50] Wladimir Schamai, Peter Fritzson, Chris Paredis, and Adrian Pop. Towards unified system modeling and simulation with ModelicaML: Modeling of executable behavior using graphical notations. In *Proceedings of the 7th Modelica Conference*, pages 612 – 621, 2009.

[51] Peter Schwarz. Physically oriented modeling of heterogeneous systems. *Math. Comput. Simul.*, 53(4-6):333–344, October 2000.

[52] Siemens. NX. http://www.plm.automation.siemens.com.

[53] Siemens. Simulation & Testing SIMIT. http://www.siemens.com.

[54] Siemens. Technomatix. http://www.plm.automation.siemens.com.

[55] MCS Software. Actran Acoustics. http://www.mscsoftware.com/product/actran-acoustics.

[56] Robert B. Stone and Kristin L. Wood. Development of a functional basis for design. *Journal of Mechanical Design*, 122(4):359–370, 2000.

[57] Dassault Systemes. SolidWorks. http://www.solidworks.com/.

[58] Janos Sztipanovits, Xenofon Koutsoukos, Gabor Karsai, Nicholas Kottenstette, Panos Antsaklis, Vijay Gupta, Bill Goodwine, John Baras, and Shige Wang. Toward a science of cyber-physical system integration. *Proceedings of the IEEE*, 100(1):29–44, 2012.

[59] Serdar Uckun. Meta II: Formal co-verification of correctness of large-scale cyber-physical systems during design. Technical report, Palo Alto Research Center, 2011.

[60] Y. Umeda, H. Takeda, T. Tomiyama, and H. Yoshikawa. Function, behaviour, and structure. *Applications of artificial intelligence in engineering V*, 1:177–194, 1990.

[61] Thom J. van Beek, Mustafa S. Erden, and Tetsuo Tomiyama. Modular design of mechatronic systems with function modeling. *Mechatronics*, 20(8):850 – 863, 2010.

[62] A. Votintseva, P. Witschel, and A. Goedecke. Analysis of a complex system for electrical mobility using a model-based engineering approach focusing on simulation. *Procedia Computer Science*, 6(0):57 – 62, 2011.

[63] Wolfgang Wahlster. Industry 4.0: From smart factories to smart products. In *Forum Business Meets Research BMR 2012*, 2012.

[64] Wolfram. Systemmodeler. http://www.wolfram.com/system-modeler/.

[65] Wolfram. Systemmodeler. http://www.wolfram.com/system-modeler/industry-examples/automotive-transportation/.

[66] Stefan Wölkl and Kristina Shea. A computational product model for conceptual design using SysML. *ASME Conference Proc.*, 2009(48999):635–645, 2009.